

**PANELS, TOOLS, AND VIEWS:
A FRAMEWORK FOR VISUAL PROGRAMMING**

BY

ALLAN K. SPALE
B.S., Elmhurst College, 2000

PROJECT

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2002

Chicago, Illinois

ACKNOWLEDGEMENTS

The author would like to thank his project advisor, Andrew Johnson, for helping him accomplish his project. The author would also like to thank Tom Moher and Robert V. Kenyon for their analysis of the author's work and providing important advice that will serve the author well for the future. The author would also thank his family and friends for supporting him during the time of this project.

TABLE OF CONTENTS

<u>CHAPTER</u>	<u>PAGE</u>
1. INTRODUCTION.	1
1.1 Background.	1
1.1.1 Visual Programming for Professionals and Non-Professionals Today.	1
1.1.2 Direct Manipulation.	2
1.2 Problem Statement.	4
1.2.1 Programmers Need Better Tools To Perform Programming Tasks.	4
1.2.2 Non-Programmers Need the Ability To Program.	6
1.2.3 The Need for a Consistent Visual Programming Metaphor.	7
1.3 Purpose of the Project.	8
 2. HISTORY OF VISUAL PROGRAMMING.	 10
2.1 Paper Notations.	10
2.2 Definitions.	10
2.3 Sutherland's <i>Sketchpad</i> : the Pioneering System.	11
2.4 Metrics For Classifying Visual Languages.	12
2.4.1 Graphics as the Primary Means of Programming.	13
2.4.2 Graphics That Accompany Conventional Programming Languages.	13
2.4.3 Visual Expressions Used for Nonprocedural Programming Languages.	14
2.5 An Overview of General Visual Programming Systems.	15
2.5.1 Syntax-Directed.	15
2.5.2 Specification-Directed.	15
2.5.3 Visually Transformed.	16
2.5.4 Forms-Based.	16
2.5.5 Visual Environments.	17
2.5.6 Flow- and Constraint-Based.	17
2.6 Demonstrational Interface Systems.	18
2.6.1 Description of Programming By Example / Programming By Demonstration and Their Systems.	18
2.6.2 Description of Demonstrational Interfaces and Their Systems.	21
2.6.3 Reuse in Programming by Example Systems.	24
2.6.4 A Description of the Ideal Programming by Demonstration System.	25
2.7 A Survey of Selected Visual Programming Systems.	26
2.7.1 <i>Pygmalion</i>	27
2.7.2 <i>Graphics-Based Program Support System</i>	29
2.7.3 <i>Tinker</i>	31

TABLE OF CONTENTS (continued)

<u>CHAPTER</u>	<u>PAGE</u>
2.7.4 <i>ThingLab</i> and <i>ThingLab II</i>	32
2.7.5 <i>PIGS</i>	34
2.7.6 <i>Pict/D</i>	34
2.7.7 <i>Rehearsal World</i>	35
2.7.8 <i>SmallStar</i>	36
2.7.9 <i>PECAN</i>	38
2.7.10 <i>ThinkPad</i>	39
2.7.11 <i>PLAY (Pictorial Language for Animation by Youngsters)</i>	40
2.7.12 <i>Chimera</i>	40
2.7.13 <i>Fabrik</i>	41
2.7.14 <i>Geometer's Sketchpad</i>	42
2.7.15 <i>Triggers</i> and Future Work Based on <i>Triggers</i>	45
2.7.16 Geographic Information Systems.	47
2.7.17 <i>Stagecast Creator</i>	49
2.7.18 System by Beaumont and Jackson.	51
2.7.19 <i>ICE</i>	52
2.8 Future Directions in the Field of Visual Programming.	52
2.8.1 Summary from "The Future of Visual Languages".	52
2.8.2 Summary from "Historical Role and Capacity of Visual Languages".	55
3. A CONCEPTUAL FRAMEWORK FOR VISUAL PROGRAMMING.	58
3.1 Explanation of Model-View-Controller.	58
3.2 Background of the Creation of Panel-Tool-View.	59
3.3 Description of Panel-Tool-View.	61
4. PROTOTYPE PROGRAM: "PANELS, TOOLS, VIEWS".	64
4.1 Previous Systems Related To and Inspiring "Panels, Tools, Views".	64
4.2 System Design.	66
4.2.1 General Introduction to "Panels, Tools, Views".	66
4.2.2 System Overview.	69
4.2.3 Object Editor.	70
4.2.4 Code View.	72
4.2.5 Instruction Panels.	75
4.2.5.1 Math.	77
4.2.5.2 Compare and Branch.	77
4.2.5.3 Move.	78
4.2.5.4 Move Characters.	79
4.2.5.5 Section Comments.	80

TABLE OF CONTENTS (continued)

<u>CHAPTER</u>	<u>PAGE</u>
5. DISCUSSION OF “PANELS, TOOLS, VIEWS”.....	81
5.1 General Comments.....	81
5.2 Suggestion for Improving the Code View in “Panels, Tools, and Views”.....	82
5.2.1 Using a Textual Representation in the Code View.....	82
5.2.2 Organization of the Pseudocode.....	84
6 FUTURE DIRECTIONS OF PANEL-TOOL-VIEW.....	88
6.1 Creating a Panel-Tool-View Development Framework.....	88
6.1.1 Coding Framework.....	89
6.1.2 Debugging-runtime Framework.....	94
6.1.3 Possible Contributions.....	98
7. CONCLUSION.....	100
APPENDIX: “Panels, Tools, Views: A User Manual”.....	103
CITED LITERATURE.....	158
BIBLIOGRAPHY.....	162

LIST OF ABBREVIATIONS

AC	Abstract Code
ACTTL	Abstract Code To Target Language
AI	Artificial Intelligence
API	Application Programming Interface
ARE	Abstract Runtime Environment
ARETRP	Abstract Runtime Environment To Runtime Panel
ASCII	American Standard Code for Information Exchange
CP	Coding Panel
CPTAC	Coding Panel To Abstract Code
CRT	Cathode Ray Tube (i.e. monitor)
GIS	Geographic Information Systems
GRR	Graphical Rewrite Rules
GUI	Graphical User Interface
HCI	Human-Computer Interaction
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
NS	Nassi-Shneiderman
NSD	Nassi-Shneiderman Diagram
OO	Object-Orientated
P-T-V	Panel-Tool-View
PAC	Program Abstract Code

LIST OF ABBREVIATIONS (continued)

PBAE	Programming By Analogous Example
PBD	Programming By Demonstration
PBE	Programming By Example
PTV	The program “Panels, Tools, and Views”
PV	Program Visualization
PWE	Programming With Example
RE	Runtime Environment
RETARE	Runtime Environment To Abstract Runtime Environment
SGML	Standard General Markup Language
SV	Software Visualization
TL	Target Language
UML	Unified Modeling Language
VP	Visual Programming
VPT	Visual Programming Tool
WYSIWIG	What You See Is What You Get
XML	eXtensible Markup Language
Y2K	Year 2000, A.D.

1. INTRODUCTION

1.1 Background

1.1.1 Visual Programming for Professionals and Non-Professionals Today

Most people who write computer programs today primarily use text-based editors. Some visual programming tools (VPTs) are used, but they are typically confined to some aspects of system design or in rapid application development (usually with prewritten components). VPTs may be one of the methods of programming more efficiently. They may be able to allow the user to focus on the logic of the program and reduce programming syntax errors. Ultimately, VPTs may help people not trained in the areas of information technology be able to write programs.

Although visual programming has had limited success in the professional arena of people in information technology, visual programming has made more progress in the area of graphical user interfaces (GUIs) found in many modern programs today. For instance, most people take for granted office automation products today; however, they were among the first beneficiaries of using direct manipulation as the primary means of interacting with the program. Word processing used to involve line editors with difficult-to-remember keyboard commands and HTML-like syntax to create “nicely formatted” documents. Creating spreadsheets would involve writing programs to complete the calculations. Now, word processing is done with WYSIWIG editors that have spell checking and grammar checking. Today, spreadsheets act as a programming language with the ability to calculate data with a variety of mathematical and

financial functions, create graphs from the data, and forecast future data by changing some datasets (Shneiderman, 1983).

1.1.2 **Direct Manipulation**

The conceptual foundation for these advancements is direct manipulation. Before beginning with the history of visual programming, it is fitting to start with one of the foundational papers from which modern user interfaces are constructed—the concept of direct manipulation. Shneiderman (1983) described the concept of direct manipulation as having a variety of characteristics:

- The object focused upon by the user should have its representation displayed continuously.
- The combination of a GUI and input devices replace complex syntax.
- “Rapid, incremental, reversible operations” visibly impact the object of interest.
- The “spiral approach to learning...permits usage with minimal knowledge...After obtaining reinforcing feedback from a successful operation, users gracefully expand their knowledge of features and gain fluency” with other features.

Direct manipulation systems are successful in part because the interface presented focuses on the semantics of operations instead of the syntax of operations. User behavior is characterized in part with a syntactic/semantic model. Syntactic knowledge is system dependent and arbitrary. This knowledge tends to be memorized in a rote manner “and [is] easily forgotten unless frequently used.” Semantic knowledge “is well-structured, relatively stable, and

meaningfully acquired.” It remains in the long-term memory where it is organized from “high-level program domain concepts down to numerous low-level program details.” (Schneiderman, 1983).

Although a command language is used in the following example given in the paper, one could also apply this example to a textual programming language. When novices use a programming language, they closely relate the syntax and semantics of the language. For these users, the syntax of the language assists the user in remembering the semantics of the language. By reviewing methods mentally or by using a manual, this process “act[s] as stimuli for recalling the related semantics. Each [method] is then evaluated for its applicability to the problem.” As the level of the user’s experience increases, the user becomes more distant from the syntactic details of a language and begins thinking in “higher level semantic terms” (Schneiderman, 1983).

It is important to think about the interface and display presented in a direct manipulation system. Users who have problems depicted in spatial or graphic manner may not improve performance. From one study, it was determined “that the content of graphic representations [was] a critical determinant of their [usability].” Another problem was the learning of “the meaning of the components in the graphic representation”. Similarly, care must be taken to ensure that the graphic representation ensures the proper meaning. Finally, the graphics representation may use much more screen space than another concise notation (Schneiderman, 1983).

If direct manipulation systems are successfully deployed, its users will experience many benefits. These benefits include the following:

- Novices can attain the skills to operate the system at a basic functionality level without a steep learning curve. Learning typically occurs by seeing a demonstration of system tasks.
- Experts can complete a wide number of tasks rapidly and may even be able to define new functions.
- Intermittent users retain concepts for operating the system.
- Progress in accomplishing a goal is visible and changes can be made to accomplish a different goal, if desired.
- Users experience less stress using a system because actions are reversible and the system design is understandable.
- Users experience confidence and mastery of the system because they control the system through initiating actions and can predict system responses.

Direct manipulation is successful in part because of the semantic/syntactic model. The user interacts with the object of interest so that user actions occur in the “high-level problem domain”. Furthermore, these actions are “closer to innate human capabilities [since] action and visual skills emerged well before language in human evolution.” (Schnedierman, 1983).

1.2 **Problem Statement**

1.2.1 **Programmers Need Better Tools To Perform Programming Tasks**

The methods of programming continue to change over time. Ten years ago, object-orientated programming was means of programming. With that paradigm shift, programmers

had to reorganize their methods of procedural programming to accommodate this change. Today, programmers have to contend with the impact of the Internet and distributed systems. Not only is object-orientated programming still crucial, but now programmers have to think about multiple instances of objects running at the same time and making the state of objects persistent over time for use at a later time. Within each paradigm, there are programming languages. Programming languages come and go, but there is an investment in time and resources to use these programming languages not only for an individual who makes a career out of programming but the businesses that create software. It is fairly certain that future paradigm shifts in programming with whatever languages used to create this software will add more complexity. Thus the programmer will have to continue to learn, adapt to, and harness all of these things in order to create the applications of tomorrow.

End-users have become more demanding than ever in what features their software should have. The mantra of “faster, cheaper, and better” used throughout the industry drives a need for code with a minimal number of errors while taking advantage of the new hardware available. Yet, the software has to be completed in a short amount of time before the next version comes out. In addition to this issue, there are always a segment of users that keep old versions of software until the computer it uses becomes useless. The Y2K crisis was the perfect example of this where business held on to programs on old systems until it became absolutely necessary to migrate to new software on a new system. As a result, these users want to have their software run on the platforms that they use despite how old the software is and what conflicts it may cause running with other software.

1.2.2 Non-Programmers Need the Ability to Program

Ordinary people, the non-programmers of the world, feel swept up by the pace and complexity of technology. The underlying reason for this is that people feel controlled by technology instead of controlling the technology themselves. Computers should empower, yet, too often, they intimidate. It is a very odd phenomenon that people blame themselves when the technology on computers does not meet their expectations. Who blames themselves when a major household appliance breaks? People do not usually say, “I am not smart enough, so I cannot use the television or the microwave.” People can control and understand these other appliances, but there are people who cannot control their computers. This phenomenon is not limited to the people who are computer-illiterate. There are also people who would like to do additional things with their computers, but do not want to invest the time and effort to do these things. This not only applies to individuals but also to people who are at an institution facing these same issues. People do embrace technology, but as with anything else, it is evaluated for its positives and negatives. Because computer technology provides such a number of positives over negatives, people will continue to want more powerful and feature-filled technology.

The basic problem is the human-computer interface. Products that sell well tend to be easy to use. Who wants to buy a hard-to-use automobile or a difficult-to-maintain lawn mower? There are some computer applications described earlier that have created good human-computer interfaces— word processors, spreadsheets, video games, etc. End-user programming is a challenge, though. When people think of programming a computer, they probably think of typing in series of cryptic commands and dealing with strange codes, and it does not have to remain this way. Abstractions should exist that provide the user enough ability to program for a

particular domain, which do already exist to some degree. It is a matter of finding a metaphor and visual interface that will inspire people to want to make an effort to learn.

People who do not write programs as a career need to control the technology around them. More electronic devices are being released, and people want to them to interconnect and different tasks that may not be available. Other end-users who have mastered the interface for an application have a sense of accomplishment when they finish some work, but sometimes they are left with the feeling, “I wish that the program could do...”. If an individual wanted to tinker with a program without becoming a programming expert, that individual should have that ability. As much as people have hobbies like play video games, paint, work on automobiles, or garden, why is it that people who do not program for a living say that one of their hobbies is writing computer programs and not be looked at strangely?

1.2.3 **The Need For a Consistent Visual Programming Metaphor**

It is important to have widely-used metaphors to allow portability of task knowledge to different domains. Currently, the metaphor for general-purpose programming is the text editor. This is the universal medium by which people write programs. The universal medium that people utilize to use the computer operating system is some sort of visual desktop metaphor or a text-based command-line interface. With the power of these agreed-upon metaphors, people are able to cross different domains to accomplish tasks. The programmer does not need to feel anxiety when programming in Java, C#, or Python because the text editor will always be there. People who use different computer platforms can feel at ease that there will be a desktop

metaphor or command line with other supporting programs in order to help them find files and configure the computer.

Likewise, it seems necessary to create a general-purpose visual programming metaphor beyond the text-editor. Just like with the use of a text editor for writing programs, this visual programming metaphor should have a consistent look and task set used for creating programs. Through its application, people who write programs will not have to invest as much time and money learning a set of languages only to have them become less and less used over time. One visual IDE could be used for a set of visual languages just as one IDE is used for many text-based languages. It will not matter what language is used because the task of programming is focused on creating the logic of the program successfully instead of struggling with the syntax of the programming.

1.3 **Purpose of the Project**

The author believes for the reasons stated previously, that there exists a general framework for visual programming that could be applied to general-purpose programming. After providing a history of visual programming and explaining how this general framework shares a similarity with the model-view-controller programming design pattern, the paper will describe the actual framework for visual programming called panel-view-tool.

In addition to providing a framework for visual programming, the author has written a program in Java (version 1.2.2) that illustrates this general framework for visual programming. The program enables a programmer to write a program in an assembly language. The running

and testing of the program must be done with an accompanying program since this was outside of the scope of the project. Accompanying this program is a user manual was also created that appears separately from this paper.

2. HISTORY OF VISUAL PROGRAMMING

2.1 Paper Notations

For the remainder of this paper, the first published year of a system found in the author's research will be surrounded by square brackets. This year should be considered an approximation since the year listed may not be the exact year the first publication of the system occurred since the author did not carefully research this aspect of the researched system. Additionally, for systems where only an implementation date was listed, the system implementation date along with the word "implemented" followed by the year of the implementation will appear surrounded by square brackets. Additionally, system names will be italicized.

2.2 Definitions

It is best to begin by properly defining the terminology used in this paper. Myers (1990) quotes a reference in his paper when defining programming as "a set of statements that can be submitted as a unit to some computer system and used to direct the behavior of that system." He also defines Visual Programming (VP) as "any system that allows the user to specify a program in two (or more) dimensional fashion...includes conventional flow charts and graphical programming languages. It does not include systems that use conventional (linear) programming languages" as well as "most graphics editors, like [Sutherland's] Sketchpad." Myers differentiates VP from the term Program Visualization (PV) that he defines as involving "...the program is specified in the conventional, textual manner, and the graphics is used to illustrate

some aspect of the program or run-time execution.” Today, this older term is now likely referred to as software visualization.

Programming has evolved over many decades utilizing the recent technology available. In the beginning, programming was accomplished using wires and a patch panel. Then programming could be performed using punch cards or a Teletype machine. Finally, the CRT and interactive editing arrived to provide the foundation for how programmers write programs today. Before focusing upon more modern visual editing, it is important to mark what could be considered as the beginning of the era of visual programming.

2.3 **Sutherland’s *Sketchpad*: the Pioneering System**

Ivan Sutherland was working on his Ph.D. thesis at the Massachusetts Institute of Technology and created what is considered a pioneering “constraint-based graphical communication system.” (Ambler and Burnett, 1989). The name of the system was *Sketchpad*, and a paper was published about its design in 1963. Instead of using instructions to generate computer graphics, a user utilized a light pen to directly manipulate graphics on the screen. A data structure contained information about different aspects of the drawing that was used to satisfy geometric constraints, permit copying, and operate on subparts of the drawing. Controls on the screen permitted various operations to be performed on parts or all of the drawing. Through constraint satisfaction, the user had “the ability to specify...mathematical conditions on already drawn parts on his drawing which will be automatically satisfied by the computer to make the drawing take the exact shape desired.” Usually, a “one-pass method” was implemented, which was similar to Moore’s algorithm, to satisfy the constraints. However, if

this was not sufficient, *Sketchpad* resorted to mathematical relaxation techniques. Some of its uses included pattern generation, geometric animations, animated cartoons and artistic drawings, scaled drawings, electrical circuit diagrams, and bridge force drawings. It would be many years before something ambitious would be attempted and published again (Sutherland, 1990).

2.4 **Metrics for Classifying Visual Languages**

There are a number of visual languages used for programming. However, each visual language has its own strengths and weaknesses. In order to effectively evaluate a visual language, one survey paper on visual programming languages by Shu (1990) described various metrics. These metrics create the “profile of a language” which characterizes the language in a three dimensional framework. Graphically, it may be represented by the surface determined by the relative measures of the language on three axes (Shu, 1990).

The assessment of a programming language will use the metrics of language level, language scope, and the visual expression of the language. The language level is “an inverse measure of the amount of details that a user has to give the computer in order to achieve the desired results.” The language scope has a “depicts how much a language is capable of doing”. The range of values starts at “general and widely acceptable” to “specific and narrowly acceptable”. The visual expression of the language is “the meaningful visual representations (for example, icons, graphs, diagrams, pictures) used as language components to achieve the purpose of programming.” (Shu, 1990).

2.4.1 Graphics as the Primary Means of Programming

From these metrics, Shu (1990) defines three categories used to describe different visual programs. The first category has “graphics...deliberately designed to play the central role in programming.” These systems include: *Pygmalion* [1975], *Xerox Star* [1982] (serves as the basis for a future work called *SmallStar* that will be described in more detail later by Halbert (1993)), *Rehearsal World* [1984], *Pict* [1984] (described in more detail later on as *Pict/D*), *VennLISP* [1986 or earlier], and *Jacob’s State Transition Diagram Language* [1985]. These systems are created for novice users who are either office workers, typically using *Xerox Star*, or new programmers, who would be likely to use *Pict*. *Xerox Star* is similar to the pervasive iconic operating system interfaces of today. However, complicated procedures must be done in a textual command language. *Pict* allows all programming to be done graphically and with a pointing device, which could be thought of as a primitive mouse today. Scoping is permitted, and color is important. On the graph, *Xerox Star* has a low language level and scope but a high level of visual expression. In comparison to *Xerox Star*, *Pict*, has an equal value for scope but has higher levels than *Xerox Star* of language level and language scope.

2.4.2 Graphics That Accompany Conventional Programming Languages

The second category has “graphics incorporated into the programming environment as an extension to conventional programming languages.” In this category are the following systems: *Diaz-Herrera and Flude’s PASCAL/HSD* [1980], *PIGS* [1983], *Belady and Hosokawa’s “Visualization of Independence and Dependence for Program Concurrency”* [1984]. *PIGS* is an acronym that stands for “Programming with Interactive Graphical Support”. In this system, “NS diagrams [serve] as executable program control constructs...that support[s] program

development and testing within the same environment.” The graphical representation extends the language. Specifically, NS diagrams are used with Pascal language for “structured control constructs of logic flow.” The system is capable of interpreting programs placed into NSD chart. The system also displays the execution sequence and supports interactive testing and debugging. *Belady and Hosokawa’s “Visualization of Independence and Dependence for Program Concurrency”* utilizes “special notations to expose the sequencing [of programs] and [their] concurrency information”. Here, the programmer provides a two-dimensional chart that indicates the order of program execution as well as what statements may be executed in parallel. This information can be provided to the compiler (Shu, 1990).

2.4.3 **Visual Expressions Used for Nonprocedural Programming Languages**

The third and final category that encompasses the above metrics is defined as containing “non-procedural programming languages using tables or forms as visual expressions”. Additionally, the graphical representations are purposely created as a crucial aspect of the language. Furthermore, “...the language cannot function without the graphic representations. Many table- and form-based languages belong to this category.” Included in this category are the following systems: *QBE* [1981] and *FORMAL* [1985]. *QBE* uses templates of tables to query a database by having the user filling in codes in fields of records. These combinations of codes in various fields of different tables will determine the database operations. *QBE* has a high language level limited to non-procedural languages, a small scope still greater than *Xerox Star* and *Pict* with a limitation to flat tables, and a low level visual expressiveness that is greater than *PIGS* and *Belady and Hosokawa’s “Visualization of Independence and Dependence for Program Concurrency”* but less than *Xerox Star* and *Pict*. *FORMAL* has the user fill in the template of the

program through the specification of its data source. Through the use of this template form, the user can specify matched fields from two input source and conditions for selecting the input data and the ordering of output data. *FORMAL* enables the user to construct complex database queries while relying on the compiler to write the algorithms to perform the actual database query operations. The formatting of the heading of the tables can also be performed with some detail. *FORMAL* has high language level even greater than *QBE*, a moderate-level language scope still greater than *QBE*, and a level of visual expression equivalent to *QBE* (Shu, 1990).

2.5 **An Overview of General Visual Programming Systems**

2.5.1 **Syntax-Directed**

From Sutherland's *Sketchpad*, people have been continuing to search for ways to simplify the tasks related to programming. Some programming systems used syntax-directed editing. These systems provided immediate feedback of syntactical errors and may have permitted the programmer to utilize language syntax templates. The *Cornell Program Synthesizer* [1981] had the programmer give language-specific commands and fill in templates. *Aloe*, an editor used in *Gandalf* [1986] which was another software development, was similar to the *Cornell Program Synthesizer* through its use of syntax templates, but it also used token replacement in the templates (Ambler and Burnett, 1990).

2.5.2 **Specification-Directed Editing**

Other systems went beyond templates and created specification-directed editing. This type of system used additional rules "on the structure of programs and enforc[ed] these rules through the editor." Two examples of this system are *Use.IT* [1976] and *PegaSys* [1986].

Use.IT guided the user through entering formal specifications “using a structure-based editor that enforces decomposition [typically of functions] based on provably correct design axioms that limit interaction between modules.” *PegaSys* graphically depicted formal dependency diagrams based on user entered design specifications. Editing occurs with “system imposed syntactic and semantic constraints.” (Ambler and Burnett, 1990).

2.5.3 **Visually Transformed**

Another set of systems used visually transformed programming where the diagram is the only program representation, and the editing of programs is done directly on the diagram. In *Pict/D* [1984], the programmer manipulates flowcharts and can create new icons that substitute for positions of the flowcharts. In the system, icons replace keywords. Another system is *PECAN* [1985] which depicted Pascal programs using Nassi-Shneiderman (NS) diagrams and flowcharts, whose representations were published in 1983 (Ambler and Burnett, 1990).

2.5.4 **Forms-Based**

Another visual language type utilizes the forms-based paradigm, which also is known as a “generalization of spreadsheet programming.” The visual cell matrix prevents the programmer from having to consider “variables, declarations, and output formatting.” The program called *Forms* [1987] extends the spreadsheet metaphor. A “basic sheet” is a form, corresponding to a piece of paper on which one can place cell matrices called objects. Each cell is only evaluated once. A “[c]ell expression” can reference any cell(s) in any object within the containing form or within other forms (Ambler and Burnett, 1990).

2.5.5 Visual Environments

Besides visual languages, there are also visual environments. *PECAN* [1985] and *VP* [1982] are visual environments. Visual environments utilize “graphical techniques in a software environment that supports the program or system development.” *PECAN* is a “family of program development systems that support multiple views of the user’s program.” The internal representation of the program is an abstract syntax tree. Concrete views are updated as changes occur. The “...views can be representations of a program or of the corresponding semantics.” *PECAN* also updates its views during run-time. It provides “visualization of a program and its run-time environment.” *SDMS* [1980], or Spatial Data Management System created by the Computer Corporation of America, provides “visualization of data or information.” The system stores information in relational databases and has its information presented through a spatial framework. It uses direct manipulation on the “graphical view of the database” to facilitate information retrieval. A system called *PV* is capable of “support[ing] manipulation of static and dynamic diagrams of computer systems; manipulation of program and document text; creation and traversal of multidimensional information space; and reuse and dissemination of tools, which is made possible by a library of diagram and text components.” (Shu, 1990).

2.5.6 Flow- and Constraint-Based

Flow- and constraint-based languages typically have a visual “front end” over a non-visual language or have a visual environment in which there is “no strictly textual equivalent”. One type of visual language utilizes data flow that has diagrams in which functional modules are connected by paths of inputs and outputs. This paradigm omits translation to text, which is typically done as a separate design step. Another type is the constraint-based paradigm

characterized by *Sketchpad* from 1963 and *ThingLab* [1986]. *ThingLab* is based on logic programming. A solution “is the set of values that simultaneously satisfies all constraints.” The paper will describe these systems in more detail at another point (Ambler and Burnett, 1990).

2.6 **Programming by Example / Programming by Demonstration Systems**

2.6.1 **Description of Programming By Example / Programming By Demonstration and Their Systems**

A paradigm for a type of visual programming is called Programming by Example (PBE).

It has the following three characteristics:

- The programmer specifies a program solely from input-output pairs.
- The programmer performs the steps of the algorithm on a set of examples. Then, the system attempts “to infer the general program structure.”
- The programmer specifies all aspects of the program, “but the [programmer] can work out the program on a specific example.” Commands execute normally, while the system remembers them for later reuse.

This may also be referred to as Programming by Demonstration (PBD) (Myers, 1990).

Another aspect of visual languages is that PBD involves a visual process that lacks a textual equivalent. A person directly manipulates graphics instead of text in order to demonstrate to the programming system what the program should do. Three examples of this type of programming include *ThinkPad* [1985], *Rehearsal World* [1984], and *Pictoral*

Transforms (PT) [1988]. *ThinkPad* is a “declarative, graphical PBD language and environment.” Manipulating diagrams of data structures demonstrates operations on data and maps to Prolog code. Running and debugging code still must occur in the Prolog environment. *Rehearsal World* is based on the SmallTalk programming environment and uses a theater metaphor to primarily help non-programmers be able to write programs. “The basic components, called performers, interact with each other on the stage (the screen) by sending cues. The screen is the stage upon which the performers (objects) perform the actions the user has taught them for a particular production (program).” *PT* allows the user to describe visual representations of data and “manipulat[e] them to develop program algorithms.” The user “design[s] graphical objects and use[s] [them] to demonstrate” how algorithms work. A picture is a “collection of graphical objects.” A film is a “sequence of manipulations performed on a picture.” (Ambler and Burnett, 1990).

One of the survey papers written by Myers (1990) classified programming systems in the following manner: VP, not PBE, batch; VP, not PBE, interactive; VP, PBE, interactive; and VP, PBE, batch. Not PBE and batch systems include: *Grail* [1969], *GAL* [1984], *AMBIT/G* [1968], *AMBIT/L* [1971], *QBE* [1977], and *FORMAL* [1985]. *Grail* utilized computerized flowcharts that were compiled. The flowchart boxes contained “ordinary machine language statements.” *GAL* implemented NS flowcharts compiled into Pascal. *AMBIT/G* and *AMBIT/L* used “symbolic manipulation programming using pictures. Both programs and data were represented diagrammatically as directed graphs, and the programming operated on pattern matching.” *QBE* “allows [programmers] to specify queries on a relational database using two-dimensional tables (or forms).” Examples are really variable names typically used in most conventional

programming languages. *OBE* is an office automation system whose ideas are extended from *QBE*. *FORMS* is a “forms-based database language...which explicitly represents hierarchical structures.”

VP, not PBE, and interactive systems include: *Graphical Program Editor* [1966], *PIGS* [1983], *Pict* [1984], *PROGRAPH* [1983], and *State Transition User Interface Management System (UIMS)* [1985]. The *Graphical Program Editor* is likely the first VP system, according to the paper written by Myers (1990), because he did not consider Sutherland’s *Sketchpad* since he viewed it as a graphics editor. Programs are represented “somewhat like hardware diagrams”. *PIGS* represents programs using NS flowcharts. *Pict* uses flowcharts whose flowchart symbols contain color icons. *PROGRAPH* describes programs in a functional data flow language that is concurrent. *State Transition UIMS* uses a state transition diagram editor for specifying the user interface graphically although it should be noted that this does not implement a modern GUI interface.

VP, PBE, and batch systems include a system by Bauer [1978]. This system uses visual programming since the “[programmer] can specify the program execution using graphical traces”. Programs are generated from input-output pairs, data structures, and algorithm specifications for a set of programs. The system executes the program on example data and infers “where loops and conditionals should go to produce the shortest and most general program that will work for all of the examples.” (Myers, 1990).

VP, PBE, and interactive systems include the following: *AutoProgrammer* [1976], *Pygmalion* [1977], *ThingLab* [1986], *SmallStar* [1981], and *Rehearsal World* [1984]. *AutoProgrammer* is similar to Bauer's system [1978] and can inference. *Pygmalion* is "one the seminal PBE systems." The programmer writes a program by manipulating icons. The emphasis is on the interactions of the programmer in the environment as opposed to writing a text program that tells the computer what to do. *ThingLab* "allows the [programmer] to describe and run complex simulations easily." It permits the definition of new constraints among objects graphically. One uses existing objects to make new objects in SmallTalk. *SmallStar* is an office automation tool that has the ability to record macros. No inferencing occurs in *SmallStar*. Users must explicitly add constraints, variables, and control structures. If the system should remember operations performed by the user, the user must make sure to set the system in program mode while performing the actions in the system's interface. *Rehearsal World* uses SmallTalk and implements a rehearsal metaphor for the purpose of "allow[ing] teachers who do not know how to program to create computerized lessons easily". Users can generate code using PBE, but sometimes the users must write code when creating a new performer if PBE is not sufficient (Myers, 1990).

2.6.2 **Description of Demonstrational Interfaces and Their Systems**

Myers (1993) wrote a later article that further described PBD and PBE systems under the category of demonstrational interfaces. He defined demonstrational interfaces as permitting the user to act upon objects related to concrete examples, typically using direct manipulation, which creates an abstract program. As a result, the user does not have to learn the programming language. Myers continues by indicating that these interfaces may be programmable or not

programmable. Additionally, the systems providing demonstrational interfaces may or may not have the capacity to infer the user actions. If the system is programmable and uses inferencing, Myers classifies the system as PBE. If the system is programmable and does not use inferencing, he classifies the system as Programming With Example (PWE) since the user is required to specify all aspects of the program even though the programmer still uses one or more specific examples during the programming process.

To further illustrate the given classifications, an example of a system or systems will be placed into each of the eight categories. To simplify repetition of terms, the focus will be upon demonstrational systems versus not demonstrational systems beginning with not demonstrational systems. Myers (1993) classifies standard direct manipulation interfaces as not intelligent and not programmable. He classifies natural language interfaces as intelligent and not programmable. The UNIX shell is programmable and not intelligent. A system named *Programmer's Apprentice* [1988 or earlier] is programmable and intelligent.

As for demonstrational systems, there are many more examples of systems given that the article written by Myers (1993), which appears in a book Cypher (1993) that focuses upon systems that are intelligent or not intelligent, programmable, demonstrational systems. *Macro Makers* and *Mondrian* [implemented 1991] are not intelligent, not programmable systems. *Eager* [implemented 1990], *Peridot* [implemented 1987], and *MetaMouse* [implemented 1988] are programmable, intelligent systems. *Predictive Calculator* [implemented 1982], *MacDraw*, and *Microsoft Word* are intelligent and not programmable systems. *Pygmalion* [1977], *Rehearsal World* [1984], *Tinker* [implemented 1979], *Tango* [1991 or earlier], *Chimera* [initially

implemented in 1987; constraints added in 1991], *Finzer's (Geometric) Sketchpad* [implemented 1991], and *TRIGGERS* [implemented 1991] are programmable and not intelligent systems. Because the conceptual VP framework discussed in this work focuses upon this category of system, many of the VP listed here will be described in more detail later.

Myers (1993) lists the advantages and disadvantages of demonstrational interfaces. The positive aspects of demonstrational interfaces include “providing abstractions concretely” through the use of permitting programmers to write a program based on directly manipulating example objects. Another positive aspect are allowing programming to be opened up to nonprogrammers who have domain-specific knowledge through the use of domain-specific interfaces. Control constructs in the program may be added directly by the nonprogrammer or through system inferencing. One final benefit is ease of use that comes through the recording of macros that may or may not use inferencing. This inferencing process may allow for the creation of proper constraints among objects in a program.

There are also some negatives for demonstrational interfaces. Many systems have not implemented demonstrational interfaces, so it is uncertain as to what applications should use demonstrational interfaces should be used and even if these interfaces are really useful at all. Additionally, there is no one set of user interface elements for using demonstrational interfaces. Another difficulty is that if this type of system uses inferencing, the system may incorrectly infer some aspect of the program. Some final negative aspects include difficulties with giving useful feedback, in providing program editing and debugging features, and in building the

systems that implement demonstrational interfaces. At the time of this article, Myers (1993) considered these difficulties as open research topics.

2.6.3 **Reuse in Programming by Example Systems**

Reuse is a major issue with PBE systems. Users must “leap cognitively between two levels of representations”: the GUI level and the program level. The difficult mapping of these levels is the “PBE representation chasm.” Analogies permit new construction of knowledge from understood knowledge. PBAE, or Programming By Analogous Examples, closes this chasm. Macro mechanisms provide the recording and static playback of user text. From the GUI level, macros are not editable once they are recorded. Users can only make modifications on the programming level, such as in Microsoft Word where macros are supported (Repenning and Perrone, 2001).

AgentSheets [1995] uses PBE and Graphical Rewrite Rules, known as GRR. “These rules declaratively describe spatial transformations with a sequence of two or more situations containing objects...[O]ne or more actions [are] capable of transforming one situation into another.” (Repenning and Perrone, 2001). To help achieve reuse, users need to add semantic annotation. With semantics, analogies can be drawn that permits generalization but limits having to consider abstract concepts. To facilitate this end, some structure exists in base icons that permit syntactic or visual rewrite rules to transform the base icons into their multiple forms. In an example from this article, a street and train track can be transformed from their base icon syntactically, i.e. its appearance, and semantically, i.e. the ability to connect other icons.

Another reuse mechanism is inheritance. Even though “class [hierarchies] may be ontologically sound, [they introduce] serious problems [for the end user]” (Repenning and Perrone, 2001). With inheritance, users would have to do the following: think abstractly, visually recognize an abstraction, and visually create an abstraction. Additionally, overgeneralization may occur, but using constraints could prevent this problem. For example, streets and train tracks are semantically equivalent since vehicles use them as a means of traveling from one place to another; however, only cars should drive on streets, and trains should only ride on tracks.

2.6.4 **A Description of the Ideal Programming by Demonstration System**

PBD usually eliminates the need for the user to learn a programming language. Instead of complete reliance on machine learning, the “user must be allowed to provide information beyond examples the give the system information about the program’s internal workings”. “Plain demonstrations” occur when the system relies on inference along with the generalization of parts of specific examples. Instead of having this scenario, the system should attempt to find the “hidden state” of the program from the user instead vice versa (McDaniel, 2001).

Two methods exist to perform this inquiry. The first is the passive-watcher method where either the system can infer something from user actions, or the user does the task without assistance. Microsoft Word with its “auto-correct, auto-indent”, etc. approach. However, this method is unable to be used for general programming. The second method is where the user gives an example of the before state and the after state while the system “infer[s] the constraints between those changes” which results in code generation (McDaniel, 2001).

The ideal PBD system would provide an advanced program editor and “allow[s] the user to specify and abstraction in the way that is most comfortable for the individual.” The user can give the system “hints” that “provide extra channels for the user to represent things that the system would find too difficult to infer on its own.” This can be done by “creating special objects and programming widgets [in addition to] using selection techniques for pointing out objects at key times.” These techniques come from systems not using PBD or inferring code (McDaniel, 2001).

Another inclusion into the ideal PBD system would be a secondary drawing area not appearing in the application’s interface used for these objects. This idea was used in *Rehearsal World* [1984] and *Gamut* [1999]. Also, one could also mix visible and invisible objects as is done with the “guidewires” in *DEMO II* [1992]. In the ideal PBD system, the “a behavior of a [program] widget becomes similar to the operations one performs in a programming language.” A heavy reliance of program widgets “can cause the same problems as writing code textually.” (McDaniel, 2001).

2.7 **A Survey of Selected Visual Programming Systems**

Now that some different classification systems have been described, a number of different visual programming systems will be described in more detail. The selection process for whether to include these systems was based on meeting one (or both) of the following criteria: the systems either were an important contribution to the field of VP or served as an incremental contribution to the area of VP; or, some portion of the system served as an inspiration or was

used directly in PTV. The presentation order of the systems is chronological based on a paper was published on the system (unless otherwise noted).

2.7.1 *Pygmalion*

Pygmalion [1977] was a system developed “*whose representational and processing facilities correspond to and assist the mental processes that occur during creative thought.*” (italics from the paper). This environment “has no representation for *telling* a program anything” about computation but only for performing computations. It “is an interactive ‘remembering’ editor for iconic data structures exhibited graphically on the display screen.” (Smith, 1990).

Its philosophy was constructed according to the following design principles:

- Present a visually orientated system.
- Abstract concepts are manifested graphically and directly manipulated through various interactions.
- Data structures and routines may be partially defined, but, upon use, the user will complete the definition.
- Multiple levels of detail are permitted (particularly data structures) under control of the user.
- Icons act as representations for data structures or a program part.
- Basic control flow and hierarchy units are provided to the user (i.e. conditionals, recursion, subroutines, classes, etc.). A basic icon shape is also provided.
- Programming is done incrementally and interactively.

- Information may be displayed in a temporal order.
- The system will have context dependent modes within the system and the user's program.
- *Pygmalion* is a general purpose programming language.

In *Pygmalion*, “icons provide the mechanism for storing and retrieving information and for representing procedures.” Symbolically, the icon represents the attributes of a program. Icons also “provide an alternative representation which stimulates creative thought in the programmer.” Smith (1990) also had a response to the current paradigm of using text editors for composing programs. He believed that traditional text editors wait for an operation, perform the operation, “display the result”, and wait for a new operation. If the editor “remembered” operations, then the result is a program that “correctly edited text once.” Expanding this idea to performing operations on data structures with visual representations brings about a more powerful result.

The system consists of a visual environment design space and runtime environment. The display mode is used “for communicating the semantics of operations visually” with a remember mode “for writing programs.” Performance of the program and creation of icons is done in the design space. Visually, icons represent a “class”. The visual environment has an icon text menu, a pointer for icon interaction and structure manipulation, icons for value of the pointed-to icon, icons with recently remembered operations, and icons for the SmallTalk evaluator (which the system is based in). Performance of the program and creation of the icons is done in the design

space. The system design creates the following characteristics (all quotations in this area come from Smith (1990) unless otherwise noted):

- “The results of operations are immediately visible and mistakes are immediately correctable...Operations are concrete rather than abstract...Data and program are visually represented.”
- “The standard mode for writing programs is what other systems consider to be debugging mode”, i.e. viewing “the step-by-step executions of operations on actual data.”
- “The human programmer is considered to be part of the execution of the program.”
- “No additional medium besides the computer is used in designing software.” A later paper by Smith (1993) describes the system as equivalent to an “electronic blackboard.”

The system seeks to improve the tools of programming but not automate the programming process.

2.7.2 **Graphics-Based Program Support System**

The *Graphics-Based Program Support System* [1978] addresses the problem that “...more than half of total [programming] effort expended goes to defect removal activities in the forms of testing and post-release defect repairs.” To remedy the situation, a Programming Support System was created “to help people produce correct programs that are self-documenting and easily read and understood by others”. According to Frei et al. (1990), it is necessary to

develop techniques that improve the specification, production, and documentation of computer programs in the ways:

- Specify programs that illustrate their structure and logic through charting techniques.
- “[D]raw and edit these charts” with “an interactive graphics system.”
- “[T]ranslate charts into executable code”.
- “[Provide] self-documentation as a by-product of the program development process”.
- “[Provide] better, interactive diagnostics and program development aids than is currently the case using a program interpreter.”
- “[Evaluate] the Programming Support System in real applications.”

The *Graphics-Based Program Support System* uses Nassi-Shneiderman (NS) diagrams that support “structured programming”. A program statement consists of a rectangular figure. Assembling these rectangular figures together represents the program. The contents of each rectangular figure may contain code “written in the base language...or other rectangular figures representing structured statements.” “NSDs...specify the control flow”, while the contents of these constructs indicate the actions that the program executes. The NSD contains a “data definition header”. The data definition has a “diagram name, a comment about its function, and a definition of the local variables, and parameters used by the diagram.” Other NSDs may contain “embedded language statements.” (Frei et al., 1990).

This system contains many different tools needed to create, edit, and execute NSDs that include the following: an editor, interpreter, preprocessor, compiler, “a dialog component for

answering questions about NSDs and system commands...[and] a set of utility routines.” The user uses the editor for graphically creating an NSD as well as adding and modifying the text of the structure and program. The question-answering mode is a help system that queries if various NSDs are available. A preprocessor translates diagrams into a language that will later compile the code. “Interpretive execution of NSDs will be provided to facilitate the development and debugging of programs.” This includes stepping through code, stopping execution on breakpoints, pausing interpretation, viewing a variable display, and “resetting program variables” (Frei et al., 1990).

2.7.3 ***Tinker***

Tinker [implemented in 1979] is based upon Lisp and is used by beginning programmers. *Tinker* uses programming-with-example to “define conditionals and recursive programs.” (Cypher, 1993). Programmers must resolve ambiguities and provide an object for a function argument “which has the appropriate data description.” It is not “a purely direct-manipulation interface” since the programmer must have knowledge of how to use Lisp. The Object List contains a history list of “objects paired with their corresponding data descriptions.” For performing an action, the programmer types the function name and selects function arguments from the Object List having the correct relation. Incremental program development is supported such as demonstrating “a simple case” and later demonstrating “more complicated situations.” (Lieberman, 1993).

Tinker is “the most procedural general PBD system” that can create conditionals when given multiple examples.” Providing the appropriate examples is important for useful code

generalization since key relationships must be shown. To decide which generalizations to use, the user must “define an expression to separate the cases”. The state of the blocks is displayed graphically. As the programmer types Lisp expressions, *Tinker* executes the program. “The Object List can be edited or the resulting Lisp definition can be edited as text.” (Lieberman, 1993). Additionally, “multiple examples define conditionals and recursive programs”, and there is no inferencing used in the system (Maulsby and Turransky, 1993).

2.7.4 *ThingLab and ThingLab II*

ThingLab [1981] was influenced by *SketchPad*. It is built as an extension to SmallTalk-76. The system focuses on “the programming language aspects of a simulation laboratory.” “[C]onstraints are used to specify the relations that must hold among parts of a simulation” which will be maintained by the system despite modification. The user specifies a constraint using “a rule and set of methods”. *ThingLab* uses a rule for testing whether a constraint is satisfied; if not, an error is given. Methods provide other means to satisfy the constraint. If the constraints are not met, a relaxation technique occurs that “approximat[es] the constraints on a given value” in some manner. This process continues until all constraints are satisfied, an optimum situation for meeting the constraints is reached, or failure occurs (Borning, 1990a).

The organization of the simulation is done through SmallTalk’s class hierarchy whereby an “object is an instance of a class”, methods of a class or its superclass are the only permitted operations on an object, and object data is internalized. *ThingLab* extends this idea with prototypes by allowing object data to have default values before the programmer sets its values, and defining a class by example using the prototype ability of the class (not always possible if

this ability is not available). Furthermore, the idea is extended through multiple superclasses and “the use of paths for symbolic references to subparts”, and “a part-whole hierarchy with an explicit, symbolic representation of shared substructure”. This hierarchy organization results in intricate relationships of constraint satisfaction. Code of existing classes can be viewed by selecting the appropriate items in the browser. Changes can be made to the code where appropriate (Borning, 1990a).

Simple graphics tools are provided for drawing items and connecting items and manipulating the diagram. The manipulation of the diagram causes recalculation in order to satisfy constraints and makes the simulation interactive. The drawing and constructing of diagrams take place in the space below the SmallTalk browser. Other regular interaction expected within the SmallTalk environment is likely to also be supported since *ThingLab* extends SmallTalk.

ThingLab II [1986], supports multiple views according to an abstraction mechanism. The constraints express the relations amid the alternate views. “Each of the alternate views may also have internal constraints, and an object must obey all the constraints specified for all its alternate views.” A relation exists between the views and the part-whole hierarchies. What could be compared to “an enclosing container object” would contain the object with “all its alternate views” and also allow the container object to own “[t]he constraints relating the views” (Borning 1990b).

2.7.5 **PIGS**

A programming system called *PIGS* [1983] continues the idea presented by Frei et al. in the *Graphics-Based Program Support System* where a single environment provides “development, debugging, testing, documentation, and maintenance of programs”. Programs are created and edited graphically using NSD. *PIGS* provides an interpreter that directly executes the chart form in the NSD program. The system also permits the setting of breakpoints and graphical step-by-step program execution. “On-line debugging and testing facilities are available which allow the user to examine and modify the program [being executed].” *PIGS* may permit the recursive calling and direct invocation of NSD modules.

2.7.6 **Pict/D**

Pict/D [1984] has no underlying textual language. Programmers compose their programs, using only a joystick, by placing icons into a flowchart creates programs. The icons can consist of system-defined functions or can be defined by the user. *Pict/D* permits recursion. The programmer may define icons. System functions are accessed through various icons. Numerical input is performed using an on-screen numeric keypad. *Pict/D* uses auditory cues to communicate system state, such as the rejection or acceptance of commands, to the programmer. Color is also used. Program execution involves the animation of the current execution step in the diagram. The system is considered a prototype and is not intended for general-purpose programming.

2.7.7 *Rehearsal World*

Rehearsal World [1984] is “an environment for developing educational software.” In this system “...only things that can be seen can be manipulated.” *Rehearsal World* is based on the SmallTalk language and was designed for educational curriculum designers. The system “extend[s] the [SmallTalk] object-message metaphor to a theater metaphor in which the basic components of a production are performers...that interact with one another on stage by sending cues.” The design environment is dynamic and interactive because one can practice using a performer before including it on the stage and watch the performers execute the production on its own or in a debugger. One can also learn SmallTalk through using the Performer Workshop that contains a limited SmallTalk browser and restricted abilities to create new primitive performers and cues. (Finzer and Gould, 1990). The following are the steps for creating a program:

- Create an empty stage
- Display the desired performer troupes (they contain similar functionality)
- Select a performer in a troupe and observe responses to available cues
- If the performer is appropriate, copy it onto the stage and resize it as needed
- Repeat until all performers are present
- Start having the system “record” the actions of the performers based on user interaction or other performers by demonstrating the actions within the system
- If the recorded actions are adequate, store the production on disk

For items not to be seen by the user, they can be kept in the “wings”, a hidden area not seen by the user. Designers can quickly understand the system and design small programs of some complexity within some time (Finzer and Gould, 1990).

2.7.8 *SmallStar*

SmallStar [1984] is a classic PBD system that “introduces the notion of programming in the user interface.” Data descriptions of objects are used by the PBD system to “refer to objects in the users’ demonstrations” (Cypher, 1993). The target audience for *SmallStar* is users of an office information system called *Xerox Star* who do not program.

The programmer creates a program by opening a program icon and “pressing the Start Recording command in the resulting window.” *SmallStar* does not record the “simple selections of Star objects.” Actions are displayed using icons and text in a transcript part of this window. In order to better illustrate why an action was taken, the programmer should edit the “operand in the program”, which has received the action, “[press] the ‘Properties’ key”, and change the properties appropriately (Halbert, 1990).

By performing the following similar measures, one could perform actions on a set of documents. First, the user performs the actions on a document that would be in a certain set of documents. Next, a user selects these actions that will be repeated. Then, the user selects “‘Repeat...’ from the pop-up menu in the program window.” Finally, the user selects the criteria for repeating by copying one of the document descriptions into the “-fill-this-in-” section of the repeat line. For conditional statements, the user will do the following: perform the actions that

would occur when the condition is true, highlight these lines in the program window, “[use] the pop-up menu in the program window...[to wrap] the statements in an ‘if’ statement”, select the predicate “-fill this in- = -fill this in-“, and open the property sheet to specify the operands of the condition (Halbert, 1990).

When programming has been completed, the user presses the “Stop Recording” command and chooses a “Run” option to run the program. Single-stepping option for the program running is available. Error messages are displayed when countered and indicate the cause of the error (Halbert, 1990).

2.7.9 PECAN

PECAN [1985] “provides views of the program, its semantics, and its execution.” Internally, *PECAN* represents a program as an abstract syntax tree. The system renders this format into different visual forms like: “[a] syntax-directed editor...[an] NS structured flowchart...[a] module interconnection diagram [for program organization].” Each view can be modifiable or not modifiable. Modifiable views allow programmer to perform manipulations that adjust the abstract syntax tree. Since these actions affect the abstract syntax tree, all other views will be modified. *PECAN* permits program execution by illustrating the current state of the program in the appropriate views.

There are a number of views available in the system. Among the program views are the syntax-directed editor and the uneditable NS view. The syntax-directed editor contains a traditional editor and an editor based on the abstract syntax tree that allows syntax tree traversal

and template-based programming. The semantic views include a symbol table, data type (currently edited data type only), expression (currently edited), and flow graph. The execution views include a view of the current frame of the program stack, including the “variables in that frame, and their values”, and involve the updating of the other views. Specifically during program execution, the current statement is highlighted in the syntax-directed editor and flow graph (Reiss, 1990).

2.7.10 *ThinkPad*

ThinkPad [1985] “model[s] graphical programming by demonstration.” The underlying language of the system is Prolog. Data structures are drawn graphically by the user and given appropriate names and types. Constraints on a data structure may be specified so that there is an “[encapsulation of] the semantics of manipulating substructures.” Many facets of the data structure definition utilize constraints including: for strong typing provisions and enforcement; for denoting “the internal and graphical hierarchy of the data structure”; and for expressing “dependencies and relationships among fields of a structure.” Constraints may be used to control the graphical representation as well as the data representation of the data structure. Multiple views, including graphics and constraints, exist for data structures and are organized onto the graphical representation called a form. The environment consists of icons to call system components, and a Data Editor. The Data Editor contains: a Data Window for the data object and constraints, a Type Display for types of data structures in the Data Window, and a Prolog Display for showing code from creating the data structure (Rubin et al., 1985).

“Functions on [data structures] are...defined by editing [their graphical] representation”. Each function has a name, arguments, and result types. Functions are also “defined as a sequence of constrained transformations from the data structures that represent the input to the data structure that represents the result.” A function should be demonstrated for each case. A “case is based on a set of constraints...[which] can be based on all of the parameters to the function.” (Rubin et al., 1985).

2.7.11 **PLAY (Pictorial Language for Animation by Youngsters)**

PLAY (Pictorial Language for Animation by Youngsters) [1986] utilizes a drama metaphor in a system designed as an educational program for teaching children how to interact with computers and program computers. Users can be in different modes: playgoer (watches a production), director (changes part(s) of a production), or playwright (writes a production). A production consists of data objects including: script, characters (appearance and sequential movement images), and backgrounds (backdrop for the scene). Editors are available for backgrounds, characters and their movements, and script.

The script consists of iconic sentences (like a comic-strip) that illustrate in a somewhat restricted manner the sequence of actions in the production. By pointing to a place in the script, the new action will be inserted after the selected icon. When viewing a play as a playgoer, the current part of the script being performed is highlighted. The play can also be started or stopped at any point. Children who tried the system enjoyed seeing the work they made despite the time and effort in creating the production. The interaction with the system occurs through selecting on-screen system icons and keyboard commands.

2.7.12 *Chimera*

Chimera [initially implemented in 1987; constraints added in 1991] is a part of the group of “[g]raphical editing and interface editing” programs that does not utilize inferencing. Specifically, Kurlander (1993) focuses upon the search and replace utility of *Chimera* for graphics called *MatchTool2*. Graphical objects can be drawn or copied into the canvases of the search or replace part of the window. Additionally, one can select which graphical attribute of the search drawing should be matched and what graphical attributes from the “replace” drawing should replace those graphical attributes of the matched drawings (Kurlander, 1993).

Besides using search and replace for graphical attributes only, one can use “constraint-based search and replace”. Constraints can exist both in the search graphic and replace graphic. “Constraints in the search pattern indicate which relationships must be present in each match, and those in the replacement pattern indicate which relationships are to be established and which are to remain unaltered.” These search and replace rules of either type may be gathered into sets of rules that can be archived, “applied to a static scene, or can be expanded dynamically as the scene is drawn and edited.” Constraints used “include distance, slope, angle, and others”. This system uses no inferencing and is used by end-users. Its purpose is to “automate repetitive search and replace tasks.” (Kurlander, 1993).

2.7.13 *Fabrik*

Fabrik [1988] is “an experimental interactive graphical environment designed to simplify the programming process by integrating the user interface, the programming language and its

representation, and the environmental languages used to construct and debug programs.” *Fabrik* targets casual and novice programmers. The system uses data-flow graphs to organize “function icons, called components”. “[C]onnection points or pins” connect components within the graph. Pins can be input, output, or bi-directional. Bi-directional pins have their direction chosen depending upon what pin connects to them. By creating a data flow graph and enforcing input-output rules on pins, “every *Fabrik* program is always syntactically correct.” The Parts Bin contains all available components. Components of related functionality are grouped in folders (Ludolph et al., 1993).

To create a program, the programmer selects components from the Parts Bin and places them into the work area called the Construction Space. Once different components are connected, the programmer enters appropriate information. If there is a visual component for display, it can be designated as the user frame. Thus, when the program runs, only the visual component is displayed. Existing programs can be modified for a specific use. Programming With Example (PWE) is used when drawing graphical elements in the Draw Component. This activity will also generate a data flow diagram that can be added to by the programmer. Since the system is interactive, any changes to part of the graph will cause values to be recalculated. Likewise, error reporting occurs after performing an illegal action.

2.7.14 *Geometer’s Sketchpad*

Geometer’s Sketchpad [implemented 1991] is a system that focuses on having students who have knowledge about geometry write programs by sketching. The interrelationships within portions of the drawing serve as the parameters and inputs to a program. By manipulating the

drawing, the input from the program will generate additional output. Constraints may be visible or hidden. Components of a drawing that are hidden permit the “synthesis of meta-constraints— composite behaviors linking visible parental objects to child objects.” “[M]eta-constraints afforded by duplicating constructions (with intermediate hidden objects) allow precisely the sort of scalability associated with modular code and static scoping.” Relationships between components in a sketch are not dependencies but are truly relationships since the system’s primitives allow “the inverse of their function” to be “a function as well.” Thus, the output of a sketch can serve to “work backward” from its input (Jackiw and Finzer, 1993).

Geometer’s Sketchpad is dynamic and interactive. When sketches have invalid relationships, these relationships may literally disappear. To find the error, one may manipulate the sketch to determine what state set is valid or invalid. The reversibility of the relationships produces what could be thought of as a two-way (backward or forward) stepping-through-code system. One can also query a sketch part to reveal relationship inadequacies. The sketch is generated in a textual script that may be recorded. The scripts “[a]t any point...may be ‘played’ onto new objects, recreating the system of constraints.” Writing a loop in the script and having objects in a script refer themselves allows for the creation of fractals. Additional program constructs include variables and procedures with parameters (Jackiw and Finzer, 1993).

2.7.15 **Triggers and Future Work Based on Triggers**

Triggers [implemented 1991] is a system designed for researcher evaluation that permits Macintosh users to create macros based on demonstrated “condition-action rules” focused upon pixel patterns. *Triggers* does not use inferencing methods and was created because data in a

program is not directly accessible to the users of the program. In order to record a condition, the user specifies various areas of the screen where pixel patterns should be searched. The user records a macro, which in this context is the recording of the trigger, by normally interacting in the program's interface using the keyboard and/or the mouse. In order to control when the system executes the rule, the user can set trigger flag that determines whether or not rule executes. A user can use multiple trigger rules and interdependent triggers to create more complex actions. Trigger rules have the ability to continuously execute without intervention or to have single-stepped execution. Rules are editable on a limited basis and have their "condition-action steps [displayed] as icons." A user can reorder rules and steps "by dragging icons." The user can also insert steps into already existing rules (Potter, 1993).

One can watch program execution by allowing searched areas of a screen to become highlighted. Also, icons in a "condition-action rule" can be highlighted during program execution. The system allows the pausing of the program execution. Some examples of using this system are for automating repetitive steps, graphical search and replace, and floating menus. Some limitations are mapping device-level programming to user goals, less efficient execution than by using higher-level constraints, and dependence on temporal interface features (such as size, placement, etc). Another limitation is that the user interface devices cannot be shared efficiently, so automated tasks cannot easily run in the background (Potter, 1993).

A paper by St. Amant et al. (2001) expands upon the ideas in the original *Triggers* paper. PBE systems have the "data description problem" that present the issue of how a system is able to realize the user intent of an object selection or manipulation. This problem affects the

system's ability to generalize user actions. To remedy this problem, the paper introduces a visual generalization system that attempts to process visual information through the user interface where user tasks are performed (St. Amant et al., 2001).

Many benefits result from this generalization system. First, if the system is not tied to specific code or an API, the system could have wider applicability. Second, “[F]unctional and visual consistency” across programs might permit flexibility within the PBE system, such as by using PBE for different web browsers. Third, visual information that is generally relevant to the application could be made accessible to the PBE system (St. Amant et al., 2001).

There are challenges also presented by the introduction of a generalization system. The first challenge is the task of image processing since real-time visual analysis of a screen is possible with high-end machines. The second challenge is information management that poses the question: “How can a system process low level visual data to infer high-level information relevant to user intentions?” For example, the system could be confused as to whether a user is clicking a rectangular area or pressing a button. Another challenge is system brittleness. Another scenario is if the visual interface changes from what the PBE system expects, how will the system react to these changes? An example involves how the system would differentiate between a banner ad containing GUI widgets and the GUI widgets that are actually related to a program (St. Amant et al., 2001).

The foundation for this work is *Triggers*. A “trigger is a condition-action pair” whereby a “user defined a trigger by performing a sequence of steps in an application.” In an enhanced

system, annotations may also be added where appropriate. Once defined, the user can execute the trigger. In *Triggers*, the system can be used to extract pixel patterns. For example, the task of moving up to a higher-level directory in a web browser would be dependent upon finding the pixel pattern of the directory divider, which is a forward slash (St. Amant et al., 2001).

The ideas in *Triggers* are found in other programs. One similar program is Yamamoto's *AutoMouse* [1998]. Another system that conceptually extends *Triggers* to allow for visual generalization is Xettlemyer and St. Amant's *VisMap* [1999]. *VisMap* "is a programmable set of sensors, effectors, and skeleton controllers for visual interaction with off-the-shelf applications." Sensors create "structured representation of visual interface object" from analyzing display pixels in image processing. An effector generates mouse and keyboard gestures to manipulate objects. To improve upon the concept of visual recognition in PBE, the addition of visual grammars may be necessary (St. Amant et al., 2001).

2.7.16 **Geographic Information Systems**

Another system described is for end-user programming is Geographic Information Systems or GIS. The purpose of the system described in this paper is to help end-users use GIS to help make decisions about their neighborhood. Faculty members, who were serving the residents, could not use GIS because of its complexity. They needed graduate students to simplify the GIS interface. These people acted as the human interface between the system and the faculty. This occurred because GIS systems required users to know technical terminology and concepts of the GIS knowledge domain as well as have a mental model of the software architecture, i.e. the sequence of tasks and the data representation, and "the system had no record

of how a display of information on a map was created.” (Traynor and Williams, 2001). According to Traynor and Williams (2001), there were four goals for the new system:

- “[P]resent the HCI in terms of the user’s task”
- “[P]rotect the user from needing technical expertise [of domain-related fields].”
- “[P]rotect the user from having to know about the software architecture.”
- “[P]rovide a program representation of the steps for creating an information display.”

PBD requires an understandable program representation that should “view it while it is being constructed, [in order] to check whether the software is making correct inferences”, read and write the program representation to disk, permit its on-demand execution, and allow its editing “for performing similar tasks”. A system named *C-SPRL* was created that has the following components: a neighborhood map serving as a GUI component, GUI menus and forms, recording and editing modes, a sequential comic-strip panel of the program with before and after images of a user command, hierarchical formatting (ex. high schools are a type of school), and multiple data displays (ex. information about the map and the map itself) (Traynor and Williams, 2001). Traynor and Williams (2001) state that there is not a large number of symbols used in the system. *C-SPRL* allows six classes of queries permitted in the system:

- Displaying “one or more objects”
- Displaying “one or more objects” that may or may not have “one or more features”
- Displaying “one or more objects” that may or may not have “one or more features”

- Displaying “one or more objects” that may or may not have “one or more features” and may or may not have “one or more attributes”
- Displaying “one or more objects” in the range of a specified distance “of one or more objects”
- Displaying “one or more attributes for one or more objects”

2.7.17 Stagecast Creator

Stagecast Creator, referred to as *Creator* for the remainder of this section, had the purpose of attempting “to make computers more useful in education.” Through years of research, it seemed necessary to create an environment that did not use a textual programming language. Instead, the environment, “focus[ed] on simulations” because they have a power to teach effectively, with “programming by demonstration and visual before-after rules.” (Smith et al., 2001).

Development in *Creator* is designed to permit teachers and students “to construct and modify simulations through programming.” The textual language remains as a barrier as preventing novices from programming. In the system, the syntax is “a list of tests and actions in a rule.” Additionally, program is restricted to domain-specific concepts. The physical system and user goals are separated by the “‘Grand Canyon’ gap between human and computer”. The user has to cross the “gulf of execution” in getting the system to act in accordance with the user’s specified goals. The computer has to cross the “gulf of evaluation” where the computer attempts to perform the user’s tasks according to the user’s expectations (Smith et al., 2001).

In *Creator*, the goal is to bring the system closer to the user by choosing analogical over “Fregean” representation. Aaron Slowman defines an analogical representation as “the structure of the representation gives information about the structure of what is represented”. Gottlob Frege, inventor of predicate calculus, has a representation where the relation “between parts of a configuration [is] the relation between ‘function-signs’ and ‘argument signs.’” Furthermore he states, “[t]he structure of such a configuration need not correspond to the structure of what it represents or denotes.” For example, there is a map going from point A to point B. A predicate calculus program would tell how to get from point A to point B. More understandability results from the analogical representations. Fregean tends to be “general and powerful.” (Smith et al., 2001).

Additionally “[e]ducational psychologist Jerome Bruner...asserted that any [knowledge domain] can be represented in three ways” which are enactive, icon, and symbolic.” All three of these representations are crucial to creative thinking. Enactive representations permit the direct manipulation and drag-and-drop capabilities of images. “[V]isual before and after rules and the domain of visual simulations” are iconic. The use of variables in simulations is symbolic (Smith et al., 2001).

Many tests were conducted on with students using prototypes of *Creator*. The subjects of the study were children, seen as novice users, who created running simulations with moving interacting objects.” It was found that there was “no gender bias”. This same system was tested with high school students. They became better programmers if they began by using *Creator*

prototypes first. As a result, “*Creator* shifts the language design emphasis from computer science to human factors.” (Smith et al., 2001).

2.7.18 **System by Beaumont and Jackson**

This is a system that had similar functionality to PTV. The goal of this system was to provide a visual interface to the Motorola 68000 processor instruction set (no official system name was provided) in a paper by Beaumont and Jackson (1997). Despite the lack of low-level programming by most programmers, it still has use today if a chip does not have a compiler from a language of a high-level available. Other reasons for low-level programming include improving system performance and accessing microarchitectural level functions unavailable in a high-level language.

Instead of creating a “universal intermediate machine code languages and notations...a set of lower level primitives [are defined] which can be used as building blocks to create higher level operation symbols.” The code is generated for the Motorola 68000 microprocessor. The main objective is to present “a concise visual form [that reduces] the potential for errors.” (Beaumont and Jackson, 1997).

Each low level instruction has a relation to a template, which appears with a large icon and spaces for the operands, if any, as well as for the instruction. An appropriate instruction icon is placed in its location in the template upon the selection of the instruction. The template also displays the states of the “status” registers. The programmer is able to understand the action of the instructions through data-flow arrows present in the diagram. Also included in the system

are labels that illustrate which entities in the interface can be attached in order to form a syntactically correct low-level command. To allow for syntactically correct commands, icons involved become inactive if they are inappropriate for a particular instruction (Beaumont and Jackson, 1997).

Instructions are divided into logical groupings with each instruction represented with a single icon. Input labels that represent an address or register appear to the left of the icon. These input labels are the data to be manipulated. The output labels appear to the right of the icon. These output labels indicate “where the manipulated data will be located” (Beaumont and Jackson, 1997).

“Parameters are constructed first by selecting the relevant addressing mode and then inserting the relevant constants, or register names into that addressing mode.” There are building blocks that correspond to a particular addressing mode where every block maps to an entity related to composing a parameter. In the lower right corner in the area of the addressing mode is its label that links “the addressing mode with the instructions.” In order to “distinguish between parameters that represent an actual number from those that represent a value in memory...[an] indirect icon appears at the top of some addressing modes.” (Beaumont and Jackson, 1997).

The visual representation of a register in this system is similar to that found in architecture books. A label may appear next to its register. With a “group of registers, the alphabetic part of the label is written above the registers and the numeric part next to the registers.” To divide the register into logical parts, vertical lines and an accompanying

description may appear. Numbers appear in the area below the register to indicate the sizes of register parts or the size of the entire register (Beaumont and Jackson, 1997).

The interface consists of four regions: “instructions, templates, addressing modes, and register description.” A user selects an item from instruction group menu and then selects the related instruction icons appearing “in a scrolled window at the bottom of the interface.” The Motorola “68000 instruction set requires seven templates” each of which can be chosen from the pressing the “‘template menu button’ in the [center] of the interface.” Once the user presses the button, the appropriate template will appear. Other buttons appear in the template area to aid in configuring and editing the panel. A user can select one of the eleven addressing modes that appear in the topmost region for use in a template. The registers appear on the right area that can be selected by the user in order to have it placed into the template (Beaumont and Jackson, 1997).

2.7.19 **ICE**

An excellent example of the ideal being sought through the work described in the paper is with the *ICE* program. Children used a paint program that served as a GUI abstraction for writing Logo programs. Children could choose from a variety of tools on the screen and then use the mouse to draw what they wanted. Depending upon the editing mode, the Logo turtle, which indicates the location of the cursor relevant to the generated Logo program, the turtle would either follow the mouse pointer while something was drawn or would wait until the child issued a signal to have the turtle follow a straight line from its previous point to the current point. *ICE* recorded the actions by having a scrollable “comic book” history that used icons found in the

user interface. It is also possible using *ICE*, as is in *Logo*, to define procedures, which would allow previously drawn shapes to be repeated in the drawing. Because of its success, the author of the paper wondered whether the abstraction provided an entry point of programming that was too low that would prevent children from planning their programs. However, the author, decided that some planning would be performed as projects become more complex (Sheehan, 2000).

2.8 **Future Directions in the Field of Visual Programming**

2.8.1 **Summary from “The Future of Visual Languages”**

Although there has been progress over the years in the area of developing better visual languages, there are many challenges that still remain. To determine the future of visual languages, a panel discussion on this topic will be summarized. Not all of the discussed material was relevant to the area covered by PTV, so those viewpoints were omitted. To simplify citation references, all quotations in this section originate from the panel discussion in this paper: (Chang et al., 1999).

S.K. Chang announced that research on visual languages should spread outward from programming languages to visual communication with consideration given to human-computer interaction. Specifically, additional research should occur in multimodal interfaces, visual and multimedia computing, visual programming in the large, foundations, and applications. For foundations, S.K. Chang wanted consideration of visual and spatial reasoning as well as other categories of languages such as multidimensional and multimodal. In the area of applications, he wanted focus on distance learning design, visualization of systems and information, and visual interfaces.

Stefano Levialdi discussed how the success of applications is heavily determined by its interface to its intended class of users. To improve visual communication of the interface to these users, there are a handful of items that he believes deserve consideration as described below:

- People should study metaphors that map actions and objects in the user's domain as well as the visualization of objects.
- The user should be fully involved in interface design from the start.
- Projects should be refined through "frequent testing, validation, and assessment".
- Difficulties should be presented during the initial stages of the development process.
- One should find an optimum balance between mental workload and visual information.
- It is necessary to create a "construction of a human-computer dialogue" which will guide and not dictate program execution.

Kim Marriott stated that the area of visual language theory began over thirty years ago. Although a majority of the goals in this research area have been achieved, the area of visual language theory has been enlarged to include "all types of visual notations...and how they are used." As a result, the following issues still need to be addressed:

- "The dynamic and interactive aspects of visual languages"

- An improved understanding of how “one visual formalism [is] more suitable than another for a particular application domain and when is a visual notation superior or inferior to a purely textual notation.”
- “A better understanding of what makes one diagram better than another at communicating a particular message.”
- How to map formal semantics to a visual language.

Margaret Burnett stated that “[visual programming languages] (VPLs) are about multidimensionality” and include “the use of icons, ...spatial relationships,...[and] time.” To produce better visual programming languages, three objectives exist:

- To empower a group of people with programming capabilities
- To increase the level of correctness in performing programming tasks
- To increase the rate that the user completes programming tasks

Designers of VPLs use the following attributes in their languages:

- Concreteness of various programming aspects.
- A reduced “distance between a goal and the actions required of the user to achieve the goal.”
- Explicitness of semantics
- Immediate user feedback of these “semantic effects of program edits.”

In addition to studying multiple programming dimensions, “serious attention s needed to the [human-computer interaction] and cognitive aspects of visual programming.” With the above attributes, the focus remains— “which language attributes help humans program”?

2.8.2 **Summary from “Historical Role and Capacity of Visual Languages”**

The author would like to share a summary of a paper by Yukio Ota entitled “Historical Role and Capacity of Visual Language”, whose paper describing the importance and future of visual language, seems to parallel the promise and importance of visual programming discussed in this work.

Ota (1999) believes that the time for visual languages is now. There are a variety of serious global problems that affects us. In order for an individual to have an awareness and understanding of the problems, the information medium used to define the problem should become more universal. Advances in transportation and communication have increased global collaborations that make the language barriers more evident. People living in this current age have an unprecedented amount of information to absorb and digest. The solution to this problem is visual language.

Written and spoken languages are defined by culture and location. They are “individualistic and restrictive”. Written and spoken languages are “reasonable, analytical...and time-sequential requiring education and study for mastery” and “restricted by the logic of the sender”. Visual languages allow people to understand each other regardless of group affiliation, do not require an education and study for mastery, and are “sensual, comprehensive, and

simultaneously understandable.” Visual language “perceives and expresses the world as it is”, thus removing the need to switch between the abstract and concrete ideas and vice versa. People “use the symbols...as clues to identify, recognize, judge, and evaluate the environment as information and then act.” Signs have two important characteristics. First, “signs are information elements and are considered to be the mark of things with meaning and situations.” Based on a person’s value standards, the individual selects signs. It is the combination of signs that form the information. Second, signs have signals that are “[perceived] and [reacted] to, as well as symbol systems as represented by languages.” (Ota, 1999).

The use of visual languages creates a paradigm shift. The study of “the relationship between meaning and form, and...[gaining] the ability to see and understand instead of read and understand, were not done in school education.” The education of an individual in technical arts and fine arts, where both fields managed color and forms, remained restricted by the “creation and appreciation of beauty or the field of cultivation of artistic sentiments.” Visual language has its foundation in daily experiences with priority given to the subjective view of the individual. Visual language is the means through which one can “interpret and give meaning to a situation.” (Ota, 1999).

Ota (1999) lists “the basic concepts of visual language which reflect the actual conditions of daily experiences”:

- “Make it possible to structurally and time sequentially express composite intellectual information.”

- Its design should appeal to both reason and feelings.
- Within the information medium, any form of information transmission means is used, such as “movies, cartoons, photographs, symbols and signs”.
- “Such information media can freely carry out metamorphosis such as analysis and synthesis and time sequential development.”

Such an environment produces interactions “between the will to transmit and the will to know, and it will probably function as...the situation in which the interface between information and information harmonizes and assimilates with the environment image. It is easy to understand and can be ignored.” This environment has been referred to as “sign-less-sign[s]”. Such an environment can be thought of as an individual’s home where this individual understands everything. Movement in the environment is easy “and becomes spontaneous and transparent...The gap between the information environment and everyday environment disappears, and they willfully merge to become one.” (Ota, 1999).

3. A CONCEPTUAL FRAMEWORK FOR VISUAL PROGRAMMING

3.1 Explanation of Model-View-Controller

Model-view-controller is a computer programming design pattern. Design patterns are designed to provide heuristics for developing a solution to a programming problem. In this case, the design pattern consists of three parts. From this point, a discrete unit used in the system that implements the solution to the problem will be referred to as a subsystem. The use of the term “object” may not be appropriate since a subsystem, or logical part of the system, may consist of many objects.

The model is underlying representation of the subsystem. It stores the data related to the subsystem as well as implementing the logic, or functionality, of the subsystem. A programmer will interact with the model through its functionalities. An end-user interacts with the model using the controllers. An end-user perceives the state(s) of the model through the view(s).

A view provides a representation of the model. There can be more than one view for a model depending upon what is useful in a particular solution. The view does not have to be visual since there might be better ways to represent the model through the uses of other modalities such as sound.

The controller serves as a mapping to some model functionality. As a result, there may be more than one controller pertaining to a model. There can be more than one controller for a model, but there does not have to be one controller for each model functionality. Furthermore, it

may be prudent to have more than one controller connected to one model functionality (or possibly vice versa). As with the view, the controller is not limited to a visual GUI widget. It may also take the form of another input modality.

3.2 **Background of the Creation of Panel-Tool-View**

There were many reasons for creating the conceptual framework of *panel-view-tool* because of two inherent difficulties associated with writing a program. First, the time and effort needed to learn new programming languages and their related APIs is becoming more difficult with the proliferation of new languages catered to have specific strengths over existing languages. Second, there are people with knowledge of their particular domain who need to write programs to complete various tasks. However, they are unable to do so because they either do not have the time to learn or are intimidated by learning to program.

In order to address these difficulties, a higher-level abstraction was created that would hide some of the prior knowledge one must have when writing a program. This would seem to be a natural evolution. The first computer programming language was assembler. That dealt with the actual hardware of the computer. Eventually, a higher-level abstraction appeared related to with the higher-level languages in use today that also include libraries of functions. The next step, which has already been taken by many companies and other research groups, is a visual abstraction. My advisor, Dr. Andrew Johnson commented that this evolutionary step is analogous to how operating system commands were abstracted away with the visual desktop metaphor introduced by systems similar to *Xerox Star* and popularized in the Macintosh and Windows operating systems.

With the choice of a visual language came the issue of finding the correct representation. This is very important because a poor visual metaphor would make the language abstraction less desirable to use or even unusable. For most popular programs that utilize a visual interface, the predominant property is a visual interface but also a direct manipulation interface. If one combines these two concepts together, one would likely achieve a desirable, productive interface to an application. Examples of this combination of interface characteristics include office suite and graphics software.

The organization I used can best be described as a *panel-tool-view* conceptual framework. This is based on model-view-controller design pattern (whose similarities to *panel-tool-view* will be presented in more detail at a later point). This conceptual framework began to take form in part when I was taking a class in my undergraduate computer science class involving algorithms and data structures. In the fall of 1998, my professor at Elmhurst College in Elmhurst, Illinois, Dr. John Jeffrey, made an analogy about how to conceptualize OO-programming. He compared the OO-programming methodology to a vending machine. My explanation of his analogy extends his basic idea. The outer part of the vending machine was the public interface. The wiring and internal control characteristics inside the vending machine were the private methods. The internal state of the machine was the private data. The front panel of the vending machine with the various buttons and slots were the public methods. Of course there were other parts of an object that were not mentioned, but this essentially provides the basic idea of what the analogy involved.

3.3 Description of Panel-Tool-View

I extended this metaphor to a general-purpose programming conceptual framework called *panel-tool-view*. The *panel* is the vending machine outer panel that is accessible to the person using the vending machine. There are different ways to display the data of the object related to the *panel* as well as other objects in the program. These are rendered in different *views*. Finally, there may be ways to manipulate objects on a *panel*. Whether these items are GUI controls related to a panel or certain functionalities provided through a mouse cursor from selecting some menu or pressing a button, these items are *tools*. To summarize, the *panel* is the organizational structure, or container, given to the user to interact with an object. There are *views* related to the *panel* that present information about the object or different parts of the program. Finally, there are *tools* related to the panel that allow the user to directly manipulate objects.

In relation to the model-view-controller pattern, there is not a perfect correlation. The *view* and *tool* parts of in this conceptual framework are analogous to the *view* and *controller* of model-view-controller. The *panel* is more of a container view that contains the controllers (*tools*) and the views. This conceptual framework had more of a focus on creating a visual design pattern that would serve the purpose of creating an interface where the programmer would not be limited to only a text editor.

To actually place the conceptual framework of *panel-tool-view* completely within the context of the model-view-controller, additional clarification is required. In this context, the model is the *abstract code* that should be generated for this *panel* that will be added to the *abstract program*. This model is named *code generator*. The mapping of the abstract program

to a specific programming language is considered to be a view typically exported to a file for compilation by the compiler of this programming language. As a result, many *panels* with zero or more *views* and one or more *tools* may be mapped to an appropriate model of the *abstract code*. Another model, named *translator*, exists whose sole purpose is to determine if the input entered into the editable *views* and/or through the use of the *tools* of the *panel* or through the use of the other *tools* will produce a valid mapping to the *abstract code*. If a valid mapping does not exist, errors must be issued. Ideally, the *translator*, that contains the current configuration of the panel, responds immediately to changes and updates the state(s) of the *views* and/or *tools* in order to prevent the user from entering a situation where an error would occur either by issuing an error immediately or disabling certain *views* and *tools*. In this context, one *translator* model maps to one *panel-tool-view*. Many *translators* can map to one *code generator* since there can be more than one *panel* of the *panel-tool-view* that can exist for a set of abstract codes.

For example, a set of abstract codes can relate to using the programming interface for a network-based socket. There can be many *panels-tools-views* that may serve as the programming interface to the abstract programming concepts of a network-based socket. Because each of the *panel-tool-view* programming interfaces is different, there are different requirements needed to enter valid input. As a result, a *translator* is required for each *panel-tool-view* in order to properly map valid input from the *panel-tool-view* into the format required for the *code generator*. Once the *code generator* has valid data, functions can be issued from this model to generate the abstract code.

Ideally, a programming environment that used *panel-tool-view* should allow for the creation of higher-level abstractions. The series of abstract codes generated by the *code generators* from the actions of a programmer, who used many *panels-tools-views*, can map to a higher-level function. This higher-level function and other higher-level functions can then have their related abstract codes mapped to a *code generator*. From there, new *panels-tools-views*, each with its own *translator*, can serve as the programming interface for this new set of higher-level functions. This process can be analogous to the creation of a library of functions or the creation of a class in a traditional programming language. One paper that might describe a somewhat similar approach in creating higher-level visual abstractions is by Jung et al. (2000). A similar idea is mentioned in an article from *Dr. Dobb's Journal* called "Getting Skinned" by the magazine's editor, Michael Swaine (2001), though the tone of the article seems more fictitious than real. The article talks about someone who files a patent for a text editor that employs plug-in skins to create an alternate form of the original text entered into the editor even though there are more frivolous skins like the columnist skin and the marketing skin. This could be considered similar to marking a document in SGML, like the abstract codes mentioned in the panel-tool-view framework, and then having different outputs created for HTML or for a printer.

4. PROTOTYPE PROGRAM: “PANELS, TOOLS, VIEWS”

4.1 Previous Systems Related To and Inspiring “Panels, Tools, Views”

The design of PTV evolved based on studying the work of others in the area of VP. Although some aspects of the design of PTV were already determined before studying previous work in VP, the studying of the work contributed to adaptations of the earlier designs of PTV. The following is a list of contributions from previous systems (and similarities of PTV to existing systems):

- *Pict/D* used flowchart representation of a subprogram using icons. There was a plan to allow macros or subprograms to be written and represented by a system-based icon, not a user-defined icon like in *Pict/D*, but this feature was not implemented in PTV.
- The important contribution of *PECAN* was its presentation of multiple code views. In PTV, there initially was plans for three separate views of the program code: visual, assembly, and pseudo-code (English). Eventually, the assembly language was dropped completely as a view and became an “exported view” that only appeared in a file. The role of the pseudo-code view was lessened from an autonomous view to popup text in the visual code view.
- *Rehearsal World* organized similar functionality into troupes, even if there was only one function in the troupe. This system was very influential to PTV because the panel organizational structure is similar to the troupe organization in *Rehearsal World*. If the user was permitted to create higher-level panels from existing code

generated from panels, then the manner of copying performers from a troupe to the stage would be even more applicable to PTV. Additionally, interacting with the troupes generated code in SmallTalk which is directly applied to the generation of code that occurs (in an abstract sense since assembly code is generated in a separate step) in PTV.

- *PLAY* presented iconic sentences for a visual representation of a program. It allowed a “play” mode that permitted movement in iconic sentences to display the forms with their respective objects and settings. Forward and backward movement was allowed. This work was applied to PTV through the use of iconic sentences displayed in the visual code view. If the programmer wanted to see the panel related to the iconic sentence on a particular line, the programmer could select this option from the menu while right-clicking on the line. The appropriate panel would then display the configuration that created the line of visual code. If the programmer has selected the overwrite mode, making changes and reapplying the line to the visual code view is analogous to editing a line in a text editor.
- *Pygmalion* supported two modes of interaction—display and remembering. The remembering mode allowed functions to be created. A similar feature was planned but never implemented for PTV in order to allow the regular usage of the panels for writing a program to be applied to a macro instead of the actual program. These macros could then later be used in manner similar to a “library of functions” to eliminate some redundancies in coding.
- *Fabrik* utilized data flow charts onto rectangular design areas called diagrams. Pins determined the direction of the data flow from various items in the diagram, and icons

were used to represent functions. *Fabrik* immediately reported if an error occurred. It utilized popup menus and allowed the editing of already existing components (functions). PTV applied the features in *Fabrik* by using data flow charts were used as the visual representation in some panels. PTV also implemented popup menus as the only manner in which to interact with the menu system. PTV occasionally provides immediate feedback on errors (although most wait until the programmer has pressed the “Apply” button in the panel) but usually disables certain options so that the errors do not occur in the first place. Again, since macros were not implemented, the modification of subprograms is not possible in PTV; however, the user can modify previously saved programs written using PTV, but only one program can be edited at one time.

- The design of *ThingLab* initially was considered for a brief time for PTV, but in the end, it did not really impact PTV.
- *ThinkPad* also was influential for PTV. *ThinkPad* uses forms for organizing data structures, and PTV uses a similar organization for functions. The programmer in *ThinkPad* manipulates the items in the forms to generate Prolog code, and PTV performs this same functionality. *ThinkPad* displays a code window, as does PTV.

4.2 **System Design**

4.2.1 **General Introduction to “Panels, Tools, Views”**

PTV is a direct manipulation, form-fill-in application. The first main item of interest is the menu usage. As is common with most applications today, popup menus tend to be context sensitive. Even though PTV also utilizes context sensitive menus, it was decided to have the

menu that would usually appear at the top of a program to be completely contained in the popup menu. This design choice was made to enable the menu to be readily accessible where the programmer is working instead of having to move the mouse to the top of the screen in order to select something from the menu. One program that known by the author which does this is a graphics program by the name Suzie345.

Another feature of PTV is the method of writing a program. Except where it is necessary for the programmer to type something in, the important parts in the process of writing the program occur by using the GUI. The programmer clicks in the operand area of an instruction panel and selects the object to place in the operand by using the popup menu. If additional information is required, a dialog box appears. The programmer might also have to press a function button when using the multifunctional Math Panel and the Compare and Branch Panel. To create a line of code, the programmer has to press the “Apply” button, which is found in every panel, that translates the configuration of the panel into a line of code once the panel has the proper configuration of operands and other items.

It is also easy to edit a line of code. The programmer simply moves the cursor in the visual code view area to the line that the desired line to edit and changes the editing mode to overwrite. By right clicking in the code view area, the popup menu appears. From the “View Panel” menu, the programmer should select “View Panel for This Instruction”. This causes the correct panel for this line of code to appear with the configuration displayed in the line of code. Then, the programmer makes the changes to the panel and presses the “Apply” button. This action results in the changes being made to the line of code.

It should be noted that PTV is simply a compiler that generates assembly language code. PTV does not provide a complete programming solution that would include a program editor, runtime environment, and debugger. The intent of PTV is to demonstrate the concepts of the panel-tool-view conceptual framework for visual programming. The ASSIST/I program used to run the program is separate from the PTV application.

Another point that must be made is that the PTV framework is not necessarily limited to visual programming. Visual programming is an alternative to traditional text based programming. For people that are visually impaired, visual programming is more detrimental than helpful. For this reason, PTV may be extended to a non-visual medium whether it is tactile or auditory because the conceptual framework is generic enough to be applied to any combination of the senses. The panel is merely a container views and a workspace to use tools. The views may be rendered in any appropriate manner that is useful to the programmer. Tools should be provided that operate directly or indirectly on the view related to the panel or some other view. The panel-tool-view conceptual framework is embodied in most applications today that provide a multitude of tools to work on some internal model rendered through a variety of views. As a result, this framework seeks to provide a more modern user interface onto the task of programming in order to allow the programmer to focus on coding the logic of the program instead of struggling with the task of coding.

4.2.2 **System Overview**

The panel-tool-view conceptual framework was applied to a subset of an assembly language that was used on an outdated IBM mainframe. This was done for two reasons. First, assembly language involves very simple program interactions that permit simple mapping of a group of instructions to a panel. Additionally, this particular assembly language was the only one known by the author. Second, if this was a successful implementation, then this interface may scale well to a higher-level computer language. The drawback of using an assembly language is that the panels are organized by similar groups of functions instead of being organized using an OO programming style. Additionally, the abstraction provided by PTV may be insufficient to warrant its usefulness.

In this program, there are six panels, three views, and two tools. The six panels include the following instruction groups: math, move, move characters, compare and branch, comments, and declarations editor. Two of the three views are related to the code of the written program, while the other view is the list of declarations. The tools are simply a mouse pointer used in a typical manner in the GUI and popup menu. In order to write a program using the application, the user directly manipulates the visual representation in the panels. The manipulation of the instruction panels will affect the code view. The manipulation of the declarations editor panel affects the declarations list view that appears in the same panel. Once the user is satisfied with the program, the user can compile the program to an assembly language file. The assembly language file could also be thought of as an exported view of the written code. The assembly language file can then be executed through the use of a Windows-based emulator of the IBM mainframe called ASSIST/I. In addition to compiling the code to a file, the code created in the

application can be saved or loaded at any time. The program is stored as a serialization of the object declared in the application.

4.2.3 **Object Editor**

The declarations panel, known as the Object Editor, consists of a table acting as the view of objects already declared. Each column describes a property of an object, while each row represents all of the properties of one object. The first column is an icon visually depicting the structure and type of the object. This will be the representation of the object in other panels in the program in addition to its name along with an array index, if appropriate. The next column is the name of the object. Since the internal representation of the code and the declarations are tied to the generated assembly language file for the purpose of demonstrating an example of the panel-tool-view framework, the name of the object is confined to the subset of legal names determined by the scope of this work. In this case, the name must begin with a letter and be followed by any combination of letters and digits with a restriction of only using uppercase letters. The third column is for the object type. There are three permissible types: integer, character, and label. The label type is not an object like an integer object. The label type was used as a matter of notational convenience in the internal declarations representation in the application. The label is used in program instructions for branching purposes. The fourth column in the declarations list view is the restriction type that can either be a constant or variable. The fifth column represents the object structure. This object property will permit the object to have a single value or a one-dimensional array of values. The sixth and final column lists the size of the object. As a general rule, there is no enforced limit on how many objects may be declared.

In the bottom half of the Object Editor panel is the editing form in which the programmer can create new objects, modify existing objects, and delete existing objects. There are some fields appearing in the editing form that are not in the table view above and vice versa. In the editing form, a comments field used to describe the object declaration and a value field given to assign value(s) to the object having a restriction of constant. Absent from the editor form is an area for object icon and total object size; although array size is available if the programmer selects the 1-D array structure for the object. All fields are text-based except when there is a restricted range of values as in type, restriction, and structure. The comments field is an optional field, while the value field is a required field only if the object has a restriction of constant.

Object declaration validity only occurs when the programmer presses one of the editing form action buttons. The available form action buttons are “Add”, “Modify”, “Delete”, “Clear”, and “Help”. Values are entered in the following manner. Single character values are surrounded by single quotes and may use a forward slash character for “escape sequences”. However only printable characters may be used with escape sequences, and the programmer may not use numerical ASCII codes and Unicode codes. Character arrays have all characters appearing inside of double quotes. Again, “escape sequences” are permitted. The programmer should enter a single integer value with a negative sign for negative values and no sign at all for positive values or zero. Integer arrays must begin and end with opening and closing curly braces, respectively and have values inside the braces separated by commas, except for the last value. If the programmer used objects in various lines of code and later changes or deletes these objects, this panel will apply the changes to the appropriate lines of code. Legal name changes will not

cause errors, but changes made to the properties of the object may cause an error in line(s) of code. It is the responsibility of the programmer to correct any errors in the code.

4.2.4 **Code View**

The other two views in PTV are the visual code view and the pseudo-code view. The pseudo-code view is embedded within the visual code view. If the programmer's mouse hovers an empty area of a line of visual code, the pseudo-code appears as a text popup. The pseudo-code view is an alternate view of the visual code, and both are dependent upon the internal representation of the code hidden from the programmer in PTV. The programmer can make changes to code through actions performed in panels or if the programmer decides to cut, copy, or paste parts of the visual code. The visual code view is actually a table with only one column. However, this column is divided into different areas, and each will be described, as it appears reading from left to right. Although objects are used in the code, they will be referred to as operands whose term is more pertinent since the underlying language is assembly-based.

The first area is for a label. The label serves as an identifier for a line of code used when branching. It is not required for a line of code to contain a label. A label uniquely identifies a line of code; therefore, it cannot appear in multiple lines of code.

The second area contains the panel name and its icon. If a panel has only one function, this area of the code will completely describe the function performed in the code; otherwise, the specific function from the panel will appear in another area of the visual code.

The third area is an operand with its icon and text name. It will only be used for the Compare and Branch Panel. This operand represents the left operand for the compare portion of the Compare and Branch Panel.

Continuing, the fourth area is an optional function icon. In PTV, it is only used for panels that have multiple functions that include the Compare and Branch Panel and the Math Panel. Although the function icon does not have a text description below it, hovering over the function icon will produce a popup text that provides the description of the function.

The fifth area has a meaning that is dependent upon what panel was used to create the instruction. If the Compare and Branch Panel was used, this area represents the right operand of the comparison; otherwise, it is the source operand for the line of code. In this paper, the source operand is defined as the operand passed to a function. For a line of code generated by the Move Characters Panel, the numerical range of array indices of characters being moved appears in a second line below the name of the operand. The first number is the starting array index, and the second number is the ending array index. The range represents the contiguous region of characters used in the move characters operation. Since this operand is the source operand, this region of characters will be copied into and replace an equally sized contiguous region of characters in the destination operand (located in another area). In this paper, the destination operand is defined as the operand that receives the results from a function. For operands that are arrays, a second line appears below its name indicating its array index. In the case of an operand used in the move characters panel, a numerical range appears below the name of the operand.

The sixth area has varying meanings but always is an operand. When used with the Compare and Branch Panel, the operand is the label of the line of code to execute. As stated above, if the panel is Move Characters, this area becomes the destination operand receiving characters to be copied into its specified array range. Similar to other panels, this area simply is the destination operand that receives the data from the function given in this line of code. The ambiguity here relates to a conflicting representation of control flow versus data flow. For the majority of the panels that generate lines of code, the visual representation in the code is data flow based. That is, data flows from one operand to another operand. In the case of the Compare and Branch Panel, the representation is control flow because the result of a comparison operation determines where execution control will be given. This conflict was permitted to remain in order to retain visual consistency.

The final area is the comments area. Because of lack of space, an icon appears here indicating that there are comments related to this particular line of code. If there are no comments for this line of code, the comments icon will not appear. To view the comments, the programmer simply moves the mouse to hover over the comments icon, and the comments appear as popup text. It may be more difficult to see longer comments because they appear on a single line, but most of the text of the comments should be visible.

The visual code view can be directly manipulated and can manipulate other panels. Basic editing functions (i.e. cut, copy, and paste) on a single line of code or contiguous lines of code can be performed using the editing instruction found in the menu. Other capabilities that determine how a new line of code is placed into the existing code are the editing modes and the

cursor location. There are two editing modes that are purposely analogous to text editing modes—insert and overwrite. Just as in a text editor, an instruction will be placed in the current line of code where the cursor is and the previous line of code shifts downward. While in insertion mode, the cursor appears as a single line that is visible below the icons and text of a line of code. In overwrite mode, a newly added line of code replaces the original line of code. The cursor appears as a rectangle surrounding the line of code in this mode. Similar rules apply to cutting and pasting a range of rows and attempts to follow the conventions found in text editors. It should be noted that empty instruction lines are not permitted. Additionally, adding a new line of code in overwrite mode in the last blank line of code is not permitted. Regardless of mode, the cursor highlights the background of the line of code. This background color will blend in with the default alternating gray background color of the line of code used to improve readability or if the line is highlighted in a shade of red to indicate an error.

4.2.5 **Instruction Panels**

All of the panels share a general structure no matter what group of instructions it is capable of generating. The first commonality is a label area. This is used to indicate that this instruction can be branched to after a compare instruction. Each label object may only serve as the branch point once. Another commonality is the comments area. The programmer may type short comments that give any information about the purpose of this line of code.

The third commonality is the action buttons labeled “Apply”, “Clear”, and “Help”. Pressing the “Apply” button causes the contents of the panel to be converted into a line of code that appears in the visual code view. Only valid instructions are generated, so any errors will be

reported at the time the “Apply” button is pressed, and the programmer will be able to fix the contents of the panel so that the generated line of code will be valid. However, it should be noted that attempts are made to disable a number of possibilities that would cause the programmer to create an illegal line of code. The “Clear” button removes all comments, operands, and the selected button (if applicable) from the panel. Pressing the “Help” button displays a window that contains information about how to use the current panel.

Another commonality is the operand areas. Right clicking in this area will allow the programmer to select the operand from the menus to place in this area. The operand will then appear with its type and structure icon along with its name. The border of this operand area represents the operand’s restriction. A solid border indicates a constant or literal, while a dashed border represents a variable. Almost every panel contains arrows. These arrows represent the data flow in the function. The exception is the Compare and Branch Panel in which the arrow represents control flow. A line emanates from the source operand. An arrowhead points to the destination operand. Sometimes, a line has a solid rectangle on an end near a smaller panel with buttons on it. These indicates that the source operand will be transformed by the function, selected in the smaller panel with buttons, and have the function output applied to the destination operand. Because of the simplicity of the functions that map to the assembly language, this representation may be too literal, but the representation may be simple enough not to be easily misunderstood. It should be noted that the only operand that may receive a label outside of the designated label area is the compare and branch panel.

4.2.5.1 **Math**

The Math Panel has a source operand on the left and a destination operand on the right. Between the two operands is a smaller panel depicting all of the possible functions. In this panel, the allowable math operations are the following: addition, subtraction, multiplication, division, and modulus. Only one operation button may be pressed at a time. According to the data flow diagram presented on the panel, the source operand has a math operation applied to it after which it is applied onto the destination operand. The destination operand contains the result of the entire operation. Acceptable data types and structures of this operation are single integers and registers. The destination operand may not have a constant restriction or have a literal integer value.

4.2.5.2 **Compare and Branch**

The Compare and Branch Panel is the only panel that depicts both data flow and control flow diagrams. The diagram is similar in structure to the Math Panel with some differences. There are two source operands and a button panel between them. Arrows emanate from the left and right source operands and point to the button panel. The button panel contains the following comparison operators: less than, less than or equal to, greater than, greater than or equal to, equal to, and not equal to. The symbols used for the comparison operands are identical to the syntax used by most programming languages today. One final button is named “Branch Always” that does not perform any comparison but simply accepts only a label operand that represents the line of code that will be branched to. The icon for this function button is an arrow pointing to a flag. The two operands are compared to one another. If the comparison is true, then program execution will jump to the next line of code marked by the label operand pointed to by the arrow

from the button panel. If the comparison is false, then execution continues with the next sequential line of code. The right comparison operands may be a constant, variable, or literal. The left operand may be a constant or variable but not a literal. Both operands must have the same type and structure.

4.2.5.3 **Move**

The Move Panel depicts an arrow from the source operand pointing directly to the destination operand. No functional transformation, as seen in the Math Panel and the Compare and Branch panel, occurs except with the single function of the panel that moves the contents of one operand into another operand.

Although there is not a group of functionalities to select, most are applied implicitly based on the type and structure of each of the operands. This is the only panel that permits a restricted set of type conversions. These conversions include the following:

- Integer, register → integer, register, character array (number converted to its text equivalent)
- Single character → single character
- Character array → integer, register

The Move Panel also allows functionalities that permit data to be displayed on the screen or input from the keyboard. These functionalities utilize two special system defined objects called “KEYBOARD” and “DISPLAY”. In general, an object with any type and structure may

be displayed to the screen. For this to occur, the programmer must place object for displaying in the source operand, while placing the “DISPLAY” object in the destination operand. Additionally, keyboard input may be stored in an object of any type and structure. To accomplish this task, the programmer must place the “KEYBOARD” object in the source operand, while placing the object to store the keyboard input into the destination operand. For any combination of objects, the destination operand cannot be a constant or a literal.

4.2.5.4 **Move Characters**

The Move Characters panel is similar to the Move Panel but only permits operations on single character objects or character array objects. The programmer selects the appropriate object and then enters the contiguous array of character array elements. As a result, the range of characters from the source operand is copied into the specified range in the destination operand. It should be noted that the number of characters moved from one object to the next must be equal. The panel prohibits moving a single character object to another single character object since this same functionality is permitted with the Move Panel. If the programmer chooses a single character object, only one character may be copied into a character array or vice versa because of the restriction of having the same number of characters moved between objects. Since the Move Characters panel prohibits the ability of moving data between two single character objects, it is recommend using the Move Panel to accomplish this task. Another restriction on operands is that the destination operand cannot be a constant or literal.

4.2.5.5 Section Comments

The final panel available for use is the Section Comments panel whose purpose is to allow a place to insert a block of comments to describe some section of the code. These comments will appear in the visual code view as well as in the generated assembly language file. The text area placed in the line of code appearing in the code view displays approximately three lines for displaying the comments; however, if carriage returns were heavily used, not all of the text from this panel will appear. Additionally, the ability of popup text appearing when the programmer hovers over the comments icon is not available here since the area provided for in the visual code typically should be sufficient. One other note is that the visual code generated for the Section Comments panel will differ in appearance from the other visual code created from the panels because the only information to be presented is the comments (i.e. no operands or other information except for the panel icon and name appear). Since this panel only generates comments, hovering the mouse in the background of the visual code for this line of code will not produce the popup pseudo-code text. Just as in other traditional programming languages, the comments will not affect the execution of the code in any way.

5. DISCUSSION OF “PANELS, TOOLS, VIEWS”

5.1 General Comments

An attempt was made to accomplish the lofty goal of trying to make PTV an effective alternative to using a text editor to program for an assembly language; however, the reality of this goal was not realized. Plans were made to conduct a user study that evaluated this particular claim, but the committee witnessing a demo of PTV was not convinced that this version of PTV accomplished this difficult goal. Other comments about PTV included the following:

- Having too low of an abstraction level to make the need for VP evident
- Creating an interface that seemed more cumbersome and time-consuming to use than a traditional text editor
- Providing a visual code view whose visual representation did not seem to add any additional meaning to an equivalent assembly language text statement

The author designed PTV primarily as a prototype to showcase the possibility of utilizing the panel-tool-view framework and make a system that was more usable than a text editor so that using a panel-tool-view framework would become the primary user interface implementation for the task of general-purpose programming. After listening to the comments made when performing a demonstration of PTV, the author continues to believe that improvements could be made that would allow some future programming tool to be written that would utilize the conceptual framework of panel-view-tool. The reasoning for this belief is based upon, as Shneiderman (1983) indicated in his direct manipulation paper, the importance of finding the

correct visual metaphor in order for the potential of this design pattern to be realized. There have been successes for end-user programming, which in this context could be considered as a domain-specific visual programming, in the area of spreadsheets, graphics, computer-aided design, and word processing, just to name a few examples. These domains have found the correct user interface in order to allow the users to accomplish tasks within their domains while minimizing the effort of using the interface. In the same manner, the author believes that there is some set of panels-tools-views that will permit general-purpose programming to occur across many programming domains and will present a consistent programming manner, which will supplant the text editor. Regardless of whether the panel-tool-view framework is used or some other visual framework for VP, the author believes that it is not a matter of *if* it will happen, but *when* it will happen.

5.2 **Suggestion for Improving the Code View in “Panels, Tools, Views”**

5.2.1 **Using a Textual Representation in the Code View**

Displaying code in a visual format may not be suitable for the general-purpose programming context because the use of a visual metaphor may be more suitable for portraying simple ideas instead of complex concepts. This may be most effective with international signs and symbols that have some kind of common elements or exhibit among them a gestalt principle. The seed for the idea began when having a conversation recently with a friend, Mubeen Ahmed, about the VPT program, and the concepts involved with panel-tool-view. He remarked that he would prefer to specify a program to the computer using words (English). It is known that this was attempted before with a programming language such as COBOL. Today, specification languages are still in use today but tend to be used by major companies, to the best of the

author's knowledge. Based on this conversation, it seemed that unambiguous specification is important to the programming process. As Mubeen wanted to use a natural language to write a program, other people such as those in Gilnert (1990) and Lieberman (2001) want an application that interprets what the programmer is trying to do in an application's user interface and have the application write the program based on PBE and PBAE concepts. As a result, using all text or all visual symbols (or even mostly inference/AI) may not be the best way to display the code one would write in a VP environment. Perhaps it is better to use the strengths of each code presentation for more specific purposes.

The author remains convinced that the panel-tool-view framework is strong for program specification. Trying to use some related visual metaphor for a literal transcription of the actions of the programmer, i.e. the program code portrayed graphically, probably is not a good idea. As Shneiderman (1983) warns in his seminal paper, a visual presentation can take up much more screen space than another text. Moreover, the visual presentation may not be more comprehensible than its textual equivalent. This was very evident in the criticism of the Code View for the PTV program. However, the Code View did encompass a combination of views. If the programmer hovered the mouse over different areas of a line of visual code, a text message would popup either describing the data object or the line of code in text (English). Based on this reasoning, it seems appropriate to use a pseudocode text representation as the primary representation in the Code View. This is not to say that other views of the code may not be more appropriate as the complexity of the program grows— perhaps software engineering-related views such as a UML diagram or metrics showing the coupling and cohesion in program modules.

5.2.2 Organization of the Pseudocode

The Code View of the PTV program would render the internal representation of the program code into a pseudocode view. It seems that this would be successful because the programmer would tend to be comfortable with text descriptions as long as the descriptions use vocabulary that the programmer would be accustomed to. Even if the programmer is not completely familiar with the word order and usage produced from a particular line of code, patterns would emerge as the programmer would see identical word order and patterns for functions used in different locations and begin to understand the meaning of the pseudocode.

Just switching to a primarily text representation of the code is alone not the answer. There are additional enhancements that the primarily text representation of the Code View could provide. Though the text is much denser than visual code, interspersing small icons by the name of the data object used in a function or an icon for the function name might improve quick readability, improve the ability to detect patterns in the code, and assist with the overall understandability of the code.

One other enhancement would be dealing with control flow in the code by relegating any code in a control construct such as an *if-then*, *switch*, *do-while*, *while-do* into separate areas from the main area of the code. Only within this section, the term “main area” is used. In this manner, the control construct appears with its conditional in the main area of the code along with an icon, such as a solid-colored diamond (with reference to a decision flowchart symbol) followed by some unique identifier (which is not bound to code but used only for the ability to locate it in the code). For instance, if there was a control construct on the screen, this

line of code could be preceded by a diamond, followed by its unique identifier, and ending with the actual condition while hyperlinked in order to reference the code that would execute if the conditional for this particular code construct was true. In a paper copy of the code, an approach similar to flowcharts and tax forms could be taken. The code not related to the blocks of code in control constructs for a program file would first be printed in its entirety. Then, on the trailing pages, the blocks of code appearing in control constructs would appear. Each control construct on this page would appear with its solid-colored diamond, a unique identifier-- either some numerical ID or a short text phrase that would describe the purpose of the condition--, and reprint its conditional such as `if (flag==true)`. For an `else` condition that appears after one or more `if` statements, all of the `if` statements could be listed, along with their particular identifiers, followed by the phrase “are false”, followed by the block of code related to the `else` statement.

There were many sources of inspiration for this particular solution to presenting code in a text view in the manner described above. The original idea was based on a comment made while demonstrating the PTV program. It was thought that the linear presentation of the code was a poor choice since being aware of a branch that occurred from a conditional in the program hardly would be evident. One of the possibilities recommended was that the code parts related to each branch could be placed side by side, which is the convention used in NSDs. Thinking about the representation further resulted in the idea for hyperlinking to the code related to the control constructs, so as to allow side-by-side comparisons of the code on the screen. The author is almost certain that there was a web site and possibly a newspaper article that described this idea of hyperlinking within program code, but there was not a successful attempt to know for certain

who exactly the person was. The inspiration for referencing a code portion on another page comes from the usage of jumping around in assembly language code with some text-based line identifier, referencing other flowcharts not on the paper using a flowchart symbol, and thinking about how the federal 1040 tax form serves as the main filing form with other appropriate tax forms plugged into various lines of the 1040 form.

The particular confidence in presenting the written code in a text format comes from a few different sources. First, this is the standard method to create code, so the text presentation of the code would be a more acceptable means instead of a visual presentation of the code that would at best be viewed very skeptically. Next, when someone provides directions to go somewhere, such as how to travel to a location, or for preparing food, text is the primary means of explaining the task with pictures serving as a secondary means to communicate an aspect of the task. Finally, it seems that people understand how conditionals work in their own life experiences but not in the programming experience. When someone offers to pour another person a glass of some drink, the one who pours may say some expression as “say when” which would translate to “I will continue pouring until you say the word ‘when’.” This is a `do-while` condition in real life. Furthermore, people are able to make a selection from a list of choices, such as in the case with shopping (a salesperson may offer many colored shirts to buy from which the customer selects one). Therefore, as long as the programmer can understand where “forks in the road” are occurring in the program and compare the different paths, the different paths of the program will be understandable without having to directly use a visual metaphor to describe the control construct. Furthermore, the pseudocode view of the program code might

serve more like reading documentation about what the code does; however, the author views the purpose for documentation as not only explaining *what* was done but *why* it was done.

6. FUTURE DIRECTIONS OF PANEL-TOOL-VIEW

6.1 Creating a Panel-Tool-View Development Framework

The purpose of the P-T-V framework is to alleviate the difficulty of specifying an algorithm and its related data structures to the computer. Instead of using the ubiquitous text editor, the author prefers to have a programmer use panels that will represent a set of concepts in a knowledge domain. This could be thought of as a mini-compiler where the panel is like a very, very narrow language that uses a small number of symbols to allow the programmer to utilize the functions related to the panel. By making the tools and views of the panel intuitive, the programmer can perform the task of programming without being bogged down in the details of the task of programming. If necessary, the panel could even guide the programmer to avoid coding in a manner that would harm the efficiency of the program, i.e. providing tips for guiding the programmer to optimize the program.

However, to be effective in its extensibility, the P-T-V framework must be applied to itself in order to become truly powerful. That is, an environment must be created that will allow panels with their tools and views to be created for abstracting anything. To be that powerful, a development framework must be created which accounts for the two most important tasks when programming— coding and debugging. Additionally, each of these tasks must have at its core a manner in which to describe the data for coding and debugging in a platform- and language-independent manner. This framework will include and extend the ideas from section 3.3. Additionally, each layer of the framework as well as the other components in the panel-tool-view framework will be described as objects with respect to OO methodologies.

This idea is not without precedent. NetBeans from Sun Microsystems (2002a, 2002b) provides a framework for building a generic desktop application whereby plugging in modules within this framework will give the specifics needed to create a specific environment for an application. The NetBeans architecture embodies the concepts of the P-T-V framework and provides the ideas for what is described in the following sections. However, NetBeans provides a framework for its implementation in Java (for coding and debugging-runtime) and is bound by the environment described through the NetBeans framework, though NetBeans is very extensible. The following development framework based upon P-T-V could at least be equally extensible.

6.1.1 **Coding Framework**

The means to create any code relies on *panels* and also serves as the top layer of the coding framework. Each *panel* has *tools* and *views* and even other *panels*. *Tools* and *views* as well as other *panels* are independent of their containing *panel* since the containing panel merely is where the *tools* and *views* are located. One could organize the previously created *tools*, *views*, and other *panels* into some sort of hierarchy so that they could be found easily during the process of creating new *panels* or changing aspects of existing *panels*. For the purpose of this part of the framework, the *panels* will be referred to as *coding panels*, or CP, since the presentation may be different for the process of coding than the process of debugging. For instance, the text editor serves merely as an editor for specifying the code, but, during the process of debugging, the currently executed line is highlighted. The CP serves as the first layer of the coding framework.

The next layer is named *Coding Panel To Abstract Code*, or CPTAC. Each CP has a CPTAC, and this is a one-to-one correspondence. Using the P-T-V framework, the CPTAC takes the state of the CP and translates one of its possible *Abstract Code* representations, or ACs. Because there would tend to be many different functions provided by the panel, there is a one-to-many relationship between a CP and ACs. More details about the AC will be given later. The CPTAC would consist of a *translator* specific to the *coding panel* as well as a *code generator* that would take the data related to the *translator* and store it into the AC representation. All of the ACs generated from all the actions performed by the programmer using the CPs consists of the *Program AC* or PAC.

The AC is the core of the coding framework. Each CPTAC has a one-to-one correspondence to an AC. However, all ACs will have the exact same encoding format despite the differing knowledge domains that they represent and the varying levels of abstraction that they represent. In other words, there should only be one *AC specification* for all ACs. The AC could be thought of as an SGML, a markup language, preferably in XML because it is a subset of SGML and a standardized markup language that can describe every aspect of coding from class definitions to simple data declarations. Just as the CP consists of *panels*, *tools*, and *views*, the AC should consist of text or binary data surrounded in appropriate XML tags related to the AC specification.

The AC specification should account for all possible coding paradigms, the major ones being OO and logic or functional (typically used for AI applications with languages such as LISP and Prolog). Additionally, it should be able to handle scripting abilities that would relate to

configuring an application (more about this later). One AC *specification* will allow for the extensibility that the coding framework of P-T-V needs to enable programming for any domain. As for defining the AC *specification*, the author is not certain at this time what the ideal representation should be. With reference to the idea given in the article by Swaine (2001), “an algorithm-representation language like Donald Knuth’s Mix” was used in the text editor skins. Perhaps one specific AC *specification* would be inadequate to handle all of the subtle variations of all the languages possible that could be generated from one AC *specification*. However, there are categories of languages that share similar properties. Therefore, having categories such as OO, non-OO, declarative, etc. would be sufficient. The downside is that by being specific to a category, more complexity entered the AC *specification*. By possibly trying to fit every possible language into an OO format might be best because there are atomic programming units that could be grouped into an object regardless of language. This would create a simple extensible language that should allow for any language to be validly mapped to the AC. Deciding on the specific details of creating this XML-based language could best be accomplished by studying a book on programming language design such as one by Pratt and Zelkowitz (1996).

The next layer below the AC is the *Abstract Code To Target Language* or ACTTL. ACTTL is merely a *code generator* that takes the code represented in the AC *specification*, shared by all ACs generated by all CPTACs, and changes it into the specified *Target Language*, or TL, which would be the final level of the coding framework. There is a one-to-one mapping between the ACTTL and the TL in the same manner that there is one CPTAC for each CP. With this analogy, the CP is a visual language as where the TL is more likely textual. The only difference between CPTAC and ACTTL is that the CPTAC has a *translator* that “interprets” the

state of the CP to some intermediate form that would then be used by its *code generator* to create the AC relevant to a CP.

Though the coding framework is generally complete, it does not easily allow for providing abstractions below the AC. As these abstractions are created, how can code be generated that allows the written program to remain language-independent and platform-independent while allowing for changes in the code at the algorithmic or data structure level? For instance, a programmer may need to change the algorithm for sorting a certain type of object or translate some graphics functions to take advantage of a new API that is conceptually similar to the old API. This is where a cascading ACTTL would be most useful.

To better describe the cascading ACTTL, it would be best to think it terms of what makes an application, in the same way that NetBeans provides a framework for creating a generic desktop application. As with a CP, the purpose of the AC is to map some aspect of domain of knowledge to a language, and the AC is used to ensure platform-independence as well as language-independence (note that it is not necessary to provide for a cascading CPTAC because an abstraction mechanism should be provided for that permits the creation of higher-level CPs by using lower-level CPs). Again as with CPs, a set of ACs constitutes a complete domain of knowledge. For example, let a set of ACs constitute an application of an IDE for P-T-V development. The highest-level ACs are analogous to a scripting language for the application. Lower-level ACs consist of the domains of knowledge used to create the application such as I/O APIs, GUI APIs, various data structure APIs, and so forth. On an even lower level, there are ACs which consist of the language (such as C++, C#, Java, etc.) used to implement the APIs. On

the lowest level, and taking the place of compilers, is the ACs that maps the language to the assembly language level. Therefore, if it seemed more advantageous to use C# instead of C++, it should just be necessary to tweak an ACTTL-AC pair within a cascading ACTTL. Thus, the higher-level abstractions (higher-level APIs, for example) remain intact, but one of the AC levels changed. The same thing could be done for porting to a different assembly language or porting to a different operating system-specific set of APIs.

Even with the cascading ACTTL, it is important to remember that there is one and only one AC *specification* in which all other ACs are encoded. The AC serves as input to the highest-level ACTTL. Then, the ACTTL uses its *code generator* to take the AC (or in the case of an entire program, a PAC) and translate it into a conceptually lower-level AC. In order for the cascading to continue, it is important that the output from this first step is in an AC specification format; otherwise, the output of a TL in the cascading ACTTL makes the process language-specific and stops the ability to abstract upon an abstraction. This process of feeding a lower-level AC into the ACTTL continues until a TL is produced. In this case, a TL is a language format that does not conform to the AC specification.

To summarize, there is one CP that maps to one CPTAC, and one CPTAC that maps to many ACs since a CP usually provides many different functions. The AC conforms to a specification, probably in an XML format, as do all other ACs. The ACs from the other panels used by the programmer will be assembled into a sequence consisting of the PAC. This single AC can be rendered using the ACTTL that will eventually create the TL. One AC can map to many TLs. However, there is a one-to-one correspondence between the ACTTL and a TL. The

ACTTL may also have cascading layers that will permit varying levels of coding abstraction where each one of the cascading layers has an ACTTL and non-PAC. The cascading continues until the final ACTTL in the cascading layers produces the TL.

6.1.2 **Debugging-Runtime Framework**

One of the problems that was encountered with the PTV program was that a separate program performed the runtime and debugging activities. In order to have a complete development framework within P-T-V, it is crucial to include these activities. However, this begins to leave the intended scope of this project since it only sought to demonstrate that P-T-V could be an effective framework for writing programs. This aspect falls into the area of software visualization, or SV. SV intends to use metaphors, although the traditional highlighting of a line of code in a text editor still would qualify to some extent, that tend to utilize the time dimension for illustrating changes in a program. However, in a complete development environment that would allow the creation of panels, tools, and views within the P-T-V environment, it is important to have a debugging-runtime framework to complete the development framework.

In this case, by creating a *runtime panel*, or RP, that is different from the CP, animating the actions related to the metaphor for this particular domain of knowledge would be more helpful than using the *coding panel* for the same purpose. Additionally, the repeated structure that could be found in the CP with buttons such as “Apply”, “Clear”, and “Help”, in the example of the PTV program, would be much less relevant. By seeing the animation of the data structures, it is hoped that the algorithms involved in a program would be much more understandable to the programmer. The debugging-runtime framework would have a similar

layered structure as the coding framework except for the primary difference in what is being modeled, i.e. code versus a runtime environment.

The topmost layer is the *Runtime Environment*, or RE. Information contained in the RE would consist of a call stack, a list of data objects currently in scope, etc. RE is the specific runtime environment provided using the TL provided in the coding framework. In essence, the RE provides the runtime state of the PAC from the coding framework. The next layer down is the *Runtime Environment To Abstract Runtime Environment*, or RETARE. This layer translates the RE to an ARE, or *Abstract Runtime Environment*. Cascading layers of RETAREs are permitted. The state of the RE propagates to the RETARE, and, within the RETARE, an ARE is generated.

As with the AC, there are many AREs, but they must all adhere to the same specification. Unlike the multiple ACs that are sequenced together to create a PAC, the ARE would be arranged more in a hierarchical fashion as is typically found in most debugging-runtime environments today. Therefore, because of the ability to cascade RETAREs, embedded definitions can exist in an ARE where some runtime object has a lower-level runtime state and so forth until its lowest runtime level is reached. Yet at each level, the runtime data conforms to the ARE specification. The ARE *specification* would probably exist as an XML document that is constantly changing. Although seemingly more complex than creating a single specification for an AC, it might be possible to provide for a single specification for an ARE. For information about creating the specification for an ARE, the author advises consulting a book on language design such as Pratt and Zelkowitz (1996). As a result, there is a one-to-one correspondence

among the RE, RETARE, and ARE. There is only one ARE active at one time unless there is a facility to allow for multiple AREs to run concurrently. This is analogous to running more than one debugger at one time in possibly another language environment. It is possible but dependent upon a set of circumstances, as in the case of running concurrent programs written in different languages.

The remaining bottom layers consist of the ARETRP, or *Abstract Runtime Environment To Runtime Panel*, and the RP, or *Runtime Panel*. The ARETRP interprets the ARE(s) to generate the correct rendering of the RP. Some of the responsibilities of the ARETP include generating the animations relevant to the RP and updating the RP to illustrate the current state of the ARE(s). The RP is simply the rendering of the domain of knowledge during the running of the program. Again, it may not be wise to have a CP identical to an RP. Also, it is important to have an RP with the appropriate *panels, tools, and views* that facilitate the task of debugging and understanding the underlying algorithms related to the domain of knowledge represented by the RP. There is a one-to-one correspondence between RPs and ARETRPs, and there are many pairs of RPs and ARETRPs for the single ARE *specification* from which would allow for the creation of many different AREs, as with the ACs.

The framework for debugging-runtime alone will not be sufficient for providing adequate debugging-runtime functionality. The PAC should always be visible in order to provide an orientation for what part of the program is executing in relation to the remainder of the program. Also, the framework should provide additional panels that allow for setting watches of objects and for setting breakpoints in the PAC that would be implicitly encapsulated within the RE. To

correctly illustrate the interactions between the different levels of CPs used for the PAC, it is necessary during step-by-step debugging to allow lower-level RPs to be skipped over or further analyzed depending on the programmer's preference. This is very similar to stepping through the code line-by-line and either choosing to look at the steps occurring in a function call or simply skipping over the intermediate steps of a function to go to the next line of code.

Because of the animation capability, it would also be nice to have a recording system that would record what the debugging process is doing so that if something unexpected happens, the programmer can pause the execution of the program and play back the steps leading to the unexpected behavior. If the state and actions of the ARE are fully recorded during the debugging process, it may be possible to "rewind" to a previous state, set different values, and continue recording the debugging process from the new point. Likewise, "rewinding" to a previous state can allow instant replay of events to better understand the reason for an unexpected event. Watches should not be recorded since they represent a selected view of the ARE. Only the actual ARE should be recorded, but the amount of data recorded may be too large, and this entire process may be impractical unless only a small window of time of the debugging process was recorded.

To summarize the debugging-runtime framework, there typically will only be one RE active at one time unless it is necessary to concurrently debug multiple programs or multiple instances of a program. If multiple REs are running, each RE will typically be bound to a different TL. An RE serves as input to the RETARE in order to create the ARE. The RETARE may utilize cascading layers where each layer consists of a RE-RETARE pair. At some point,

each of the higher-level REs generated from the cascading layers will be converted into one cascaded ARE. The data represented in the ARE serves as input to the ARETRP. The output from ARETRP is the proper configuration and possibly animation of components within the RP.

6.1.3 **Possible Contributions**

The first possible contribution is a mix of the NetBeans platform and the idea alluded to in an article by Duntemann (1998). Duntemann predicts how software development will settle into programmers choosing which of the following layers or categories to program for: operating systems, operating systems drivers, development tools, software components, applications, databases, and structures of meaning. The term “structures of meaning” can best be described by Dunteman (1998) in the following way: “Many of these will be what you could call ‘active documents’...In this category falls much of what we now call multimedia authoring, scripting, HTML, XML, SGML...It’s not quite programming in the traditional sense, but it’s not just writing or drawing pictures either.” Both the NetBeans architecture and this article inspired the coding framework. From the Duntemann (1998) viewpoint, the task of programming becomes more focused on thinking about the knowledge domain instead of about the programming task. For example, I can have an AC, similar to the core that is present in the NetBeans platform, that could specify the configuration of a hypothetical P-T-V IDE, while the code used to compose this APIs (maybe scripting language would be more appropriate at this high level) first begins with APIs for I/O, GUIs, data structures, etc. The next lower level would consist of high-language implementation of these APIs in C#, C++, Java, etc. At the lowest level, is the assembly language level that implements the high-level language to the chip level— that is, what compilers do. If P-T-V provides an abstraction for the assembly level, one could write a

compiler using P-T-V, and the creation of higher-level panels from the assembly level would be analogous to the syntax of a high-level language. Then one could abstract the abstraction and so forth. However, the argument against this is that the AC specification becomes the language and there is really nothing new other than what C++ and Java do for algorithm portability across many OS platforms.

The second possible contribution is with the debugging-runtime framework. Being able to provide an animation of the runtime environment might be useful in assisting the programmer to better understand an algorithm. These ideas of software visualization are collected into a book by Stako et al. (1998). If implemented properly, this would be especially useful in programs that rely on concurrent execution. Being able to implement an RP onto an ARE would allow different aspects of a library of functions or an object depending on what the programmer was trying to understand. Additionally, one could use multiple RPs that map to the same portion of the ARE in order to have alternate views of the same library of functions or object during execution. On a simpler level, having multiple layers of ARE as a result of the layered AC may further assist a programmer in understanding the processes occurring in the code. However, one could also argue of the existence of this idea because debugger-runtime environments now can show the various data represented in objects from a more abstract level of an object to the primitive level of the actual bit pattern stored for the data or the value of the primitive data types in human-readable form.

7. CONCLUSION

An attempt was made to try to describe a visual programming framework that is based on two of the cornerstones in user interfaces today— direct manipulation and the model-view-controller design pattern. This framework, named panel-tool-view, provides the means to improve the tasks related to programming by simplifying the process of specifying the program through providing a more intuitive interface that, in the author’s opinion, surpasses the text editor. A panel, with its related tools, views, and perhaps other panels, represents an aspect of a domain of knowledge, while a set of panels represents a complete domain of knowledge. By freeing the programmer from the syntax, non-programmers could begin to program because the metaphors presented in the panel-tool-view framework would focus on the relations between the visual metaphors for the functionality provided in the panels. The framework does not become overwhelming because of the manner in which related knowledge within a domain is grouped together into the panels and will not allow the programmer to be completely overwhelmed by trying to learn a new visual language.

From the panel-tool-view framework, a program called “Panels, Tools, Views” was written to illustrate the concepts of the panel-tool-view. The author believes that because the target language was assembly language-based, this was one of the reasons for its poor evaluation. The visual translation of the assembly language was too literal and greatly offset the possible gains of using a visual environment to write assembly language code. Additional reasons for the failure of the program included menu organization that made working in the environment more cumbersome, a lack of keyboard shortcuts, and the inability to organize the panels to be present

on the screen at all times. Finally, the possible gains achieved in being able to specify a program without knowing the syntax were offset by a visual presentation of the written program that did not seem to have an advantage over textual code. By applying improvements described in a previous chapter and providing a means for the programmer to create higher-level abstractions, perhaps a future incarnation of “Panels, Tools, Views” will be more purposeful and user-friendly.

Finally, a glimpse into the future of the panel-tool-view framework was provided through the description of a development environment for the framework. This development framework consists of two parts— a coding framework and a debugging-runtime framework. The coding framework tries to provide the ability to introduce a core representation for any programming language now or in the future (as long it becomes mapped into the programming framework). From this abstract code core, panels will generate code to this abstract code specification. Additionally, the abstract code encapsulates the concepts of the algorithm allowing multiple panels that have different appearances to generate the same code. Furthermore, through the use of cascading levels of abstract code and the translation of abstract code to a target language, this process allows algorithms to be defined to any level of abstraction and implemented in any existing or future textual language. Ideally, algorithms will now have a platform-independence and language-independence that was not known before.

The debugging-runtime framework allows for a layered runtime environment in the same way that there is a layered runtime environment in debuggers today starting at the lowest level of bit patterns and memory locations for all the data of an object all the way up to the highest level

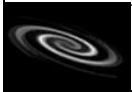
abstraction where high-level object consist of the data members of this object. Special runtime panels are introduced that allow for the actions of the program to be animated with the help of the abstract runtime environment. By viewing actions that are occurring in the code, as one would view the workings of a physical machine, the algorithm becomes more understandable and easier to debug if there are problems. Also, this animation of data changing would hopefully facilitate a more pleasant experience when debugging concurrent programs.

There is still a lot of work to be done in the area of visual programming. Progress has been made in opening up the field of programming to individuals who have not chosen programming as their careers. However, additional work is necessary in the area of visual programming to not only further provide the tools that simplify the task of programming but to provide tools which also enable the writing of programs with fewer errors and faster running times. The author believes that the ideas described in the panel-tool-view framework and in its future work of a complete development environment built around this concept provide hope for the future because the ideas are built upon a strong foundation of the concept of direct manipulation and the design pattern of model-view-controller which is prevalent in the user interfaces of software today. Eventually, the ideal of programming may be reached which Ota (1999) describes where “[t]he gap between the information environment and everyday environment disappears, and they willfully merge to become one.”

APPENDIX

Panels, Tools, and Views: User Guide

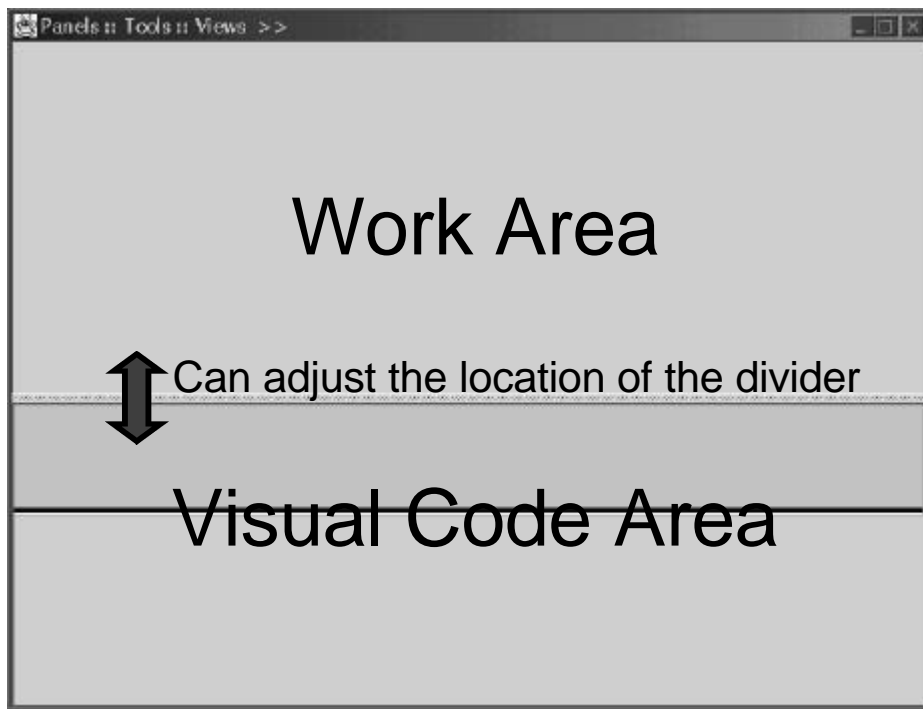
Allan Spale



Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

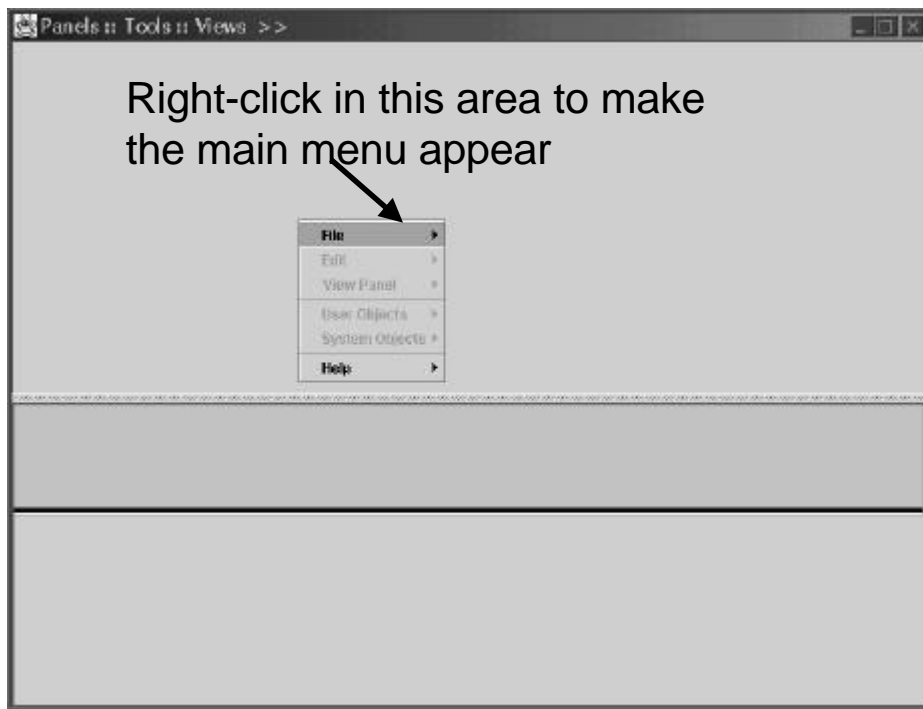
Environment



Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

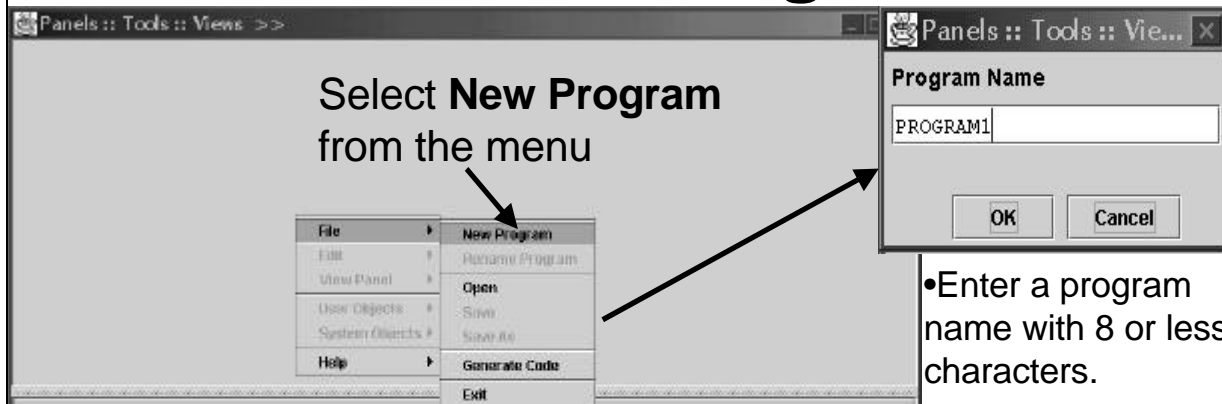
Main Menu



Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

New Program / Rename Program



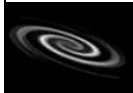
Later, if you want to rename the program, select **Rename Program** from the menu, and follow the same naming process.

- Enter a program name with 8 or less characters.

- Start with a letter.

- Can use letters or numbers.

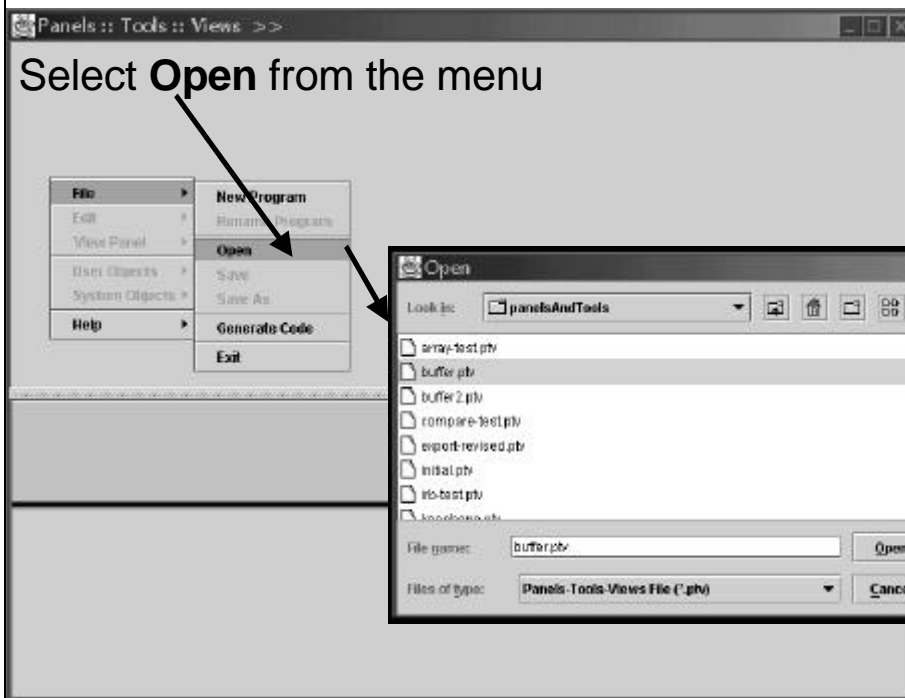
- Program name will appear in the window title.



Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

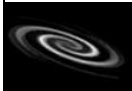
Open Program



Select **Open** from the menu

- Select a file from the list of available Panel-Tools-Views files (.ptv)

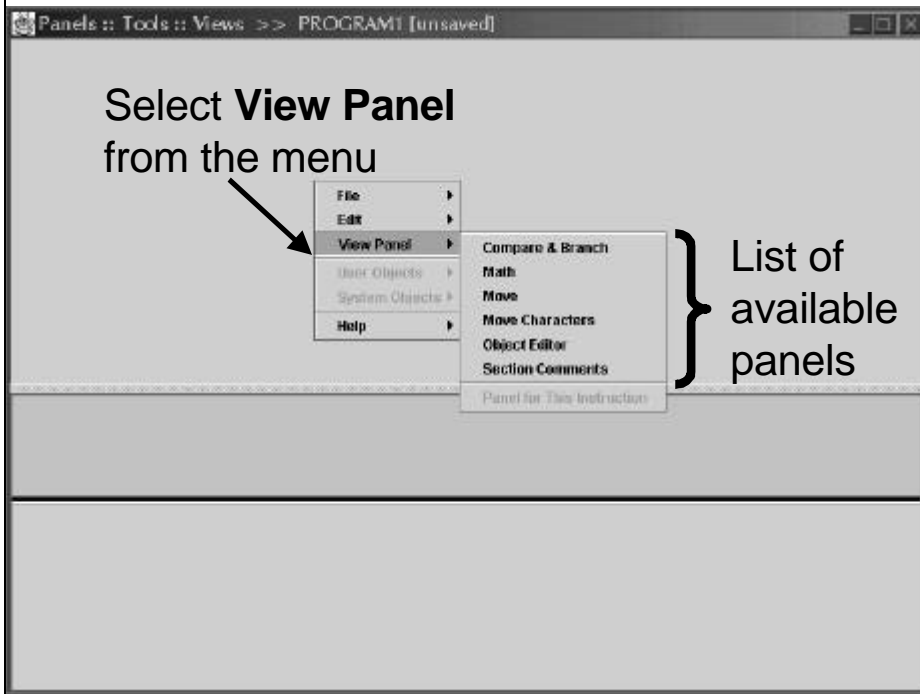
- Or, type the name of the file in the **File Name** field (no extension is required).



Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

Panels



- A panel is an organization for groups of similar functions

- User and system defined objects are used as arguments in the functions of a panel

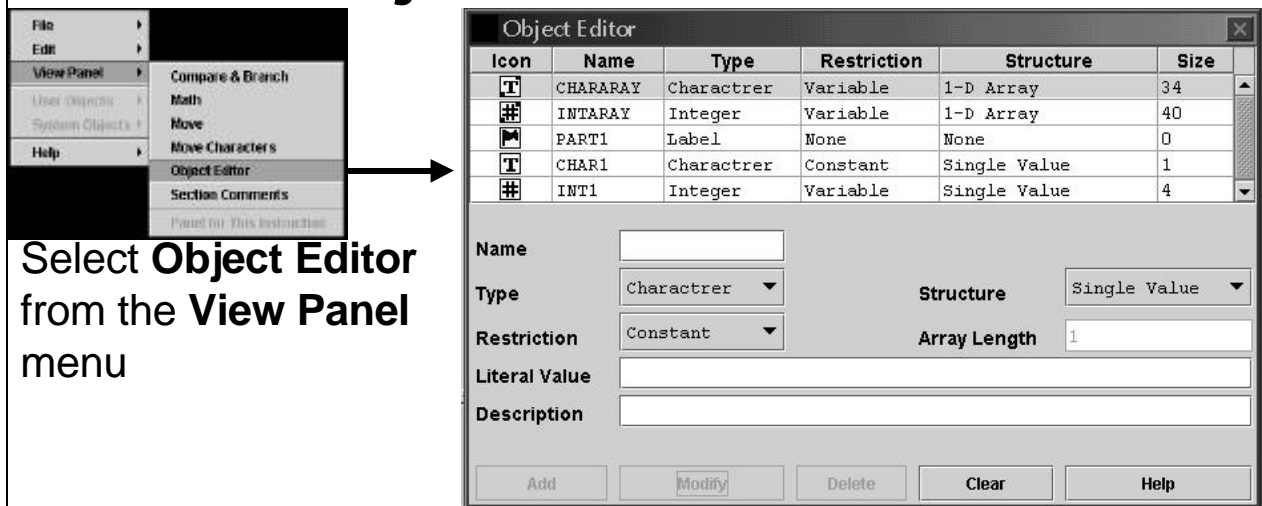
- Panels have a visual metaphor with respect to the functionality provided



APPENDIX

Defining Data Objects: Object Editor Panel

Select Object Editor from the View Panel menu



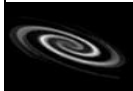
The Object Editor panel consists of a table and a form. The table lists data objects with their icons, names, types, restrictions, structures, and sizes. The form allows for editing the selected object's properties.

Icon	Name	Type	Restriction	Structure	Size
T	CHARARRAY	Characterer	Variable	1-D Array	34
#	INTARRAY	Integer	Variable	1-D Array	40
M	PART1	Label	None	None	0
T	CHAR1	Characterer	Constant	Single Value	1
#	INT1	Integer	Variable	Single Value	4

The form includes the following fields and controls:

- Name:
- Type: (dropdown)
- Restriction: (dropdown)
- Structure: (dropdown)
- Array Length:
- Literal Value:
- Description:
- Buttons: Add, Modify, Delete, Clear, Help

Define all data objects using this table and form



Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

Defining Data Objects: Name

The screenshot shows the 'Object Editor' window. At the top is a table with columns: Icon, Name, Type, Restriction, Structure, and Size. Below the table is a configuration panel with fields for Name, Type, Restriction, Structure, Array Length, Literal Value, and Description. At the bottom are buttons for Add, Modify, Delete, Clear, and Help.

Icon	Name	Type	Restriction	Structure	Size
T	CHARARRAY	Character	Variable	1-D Array	34
#	INTARRAY	Integer	Variable	1-D Array	40
M	PART1	Label	None	None	0
T	CHAR1	Character	Constant	Single Value	1
#	INT1	Integer	Variable	Single Value	4

Name:

Type: Structure:

Restriction: Array Length:

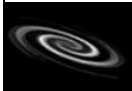
Literal Value:

Description:

Add Modify Delete Clear Help

- Must start with a letter
- Can contain characters and numbers but no special symbols (?, !, _, etc.)
- All letters must be uppercase
- Cannot use the program name or a system-defined data object name

Name: Unique identifier used to reference data objects in a program.



Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

Defining Data Objects: Type

The screenshot shows the 'Object Editor' window with a table of data objects. The table has columns for Icon, Name, Type, Restriction, Structure, and Size. The 'Type' column is highlighted with a red box. Below the table, the 'Type' dropdown menu is also highlighted with a red box, showing 'Characterer' selected. Other fields like 'Restriction' (Constant), 'Structure' (Single Value), and 'Array Length' (1) are visible.

Icon	Name	Type	Restriction	Structure	Size
T	CHARARRAY	Characterer	Variable	1-D Array	34
#	INTARRAY	Integer	Variable	1-D Array	40
M	PART1	Label	None	None	0
T	CHAR1	Characterer	Constant	Single Value	1
#	INT1	Integer	Variable	Single Value	4

Name:

Type: **Characterer** (dropdown)

Restriction: **Constant** (dropdown)

Structure: **Single Value** (dropdown)

Array Length:

Literal Value:

Description:

Buttons: Add, Modify, Delete, Clear, Help

•Integer

- Any number between -2 billion to +2 billion approx.

•Character

- Any valid character typed on the keyboard (for this application)

•Label

- Used to *uniquely* mark a program line
- Used to jump to a program line during execution

Type: A category for a fixed set of values such as integer and character. A label is also included though it does not have a value.



Electronic Visualization Laboratory University of Illinois at Chicago

Defining Data Objects: Structure

Icon	Name	Type	Restriction	Structure	Size
T	CHARARRAY	Character	Variable	1-D Array	34
#	INTARRAY	Integer	Variable	1-D Array	40
M	PART1	Label	None	None	0
T	CHAR1	Character	Constant	Single Value	1
#	INT1	Integer	Variable	Single Value	4

Object Editor

Name:

Type: Character

Restriction: Constant

Structure: Single Value

Array Length: 1

Literal Value:

Description:

Buttons: Add, Modify, Delete, Clear, Help

•None

- Has no structure
- Only occurs for Label type

•Single Value

- Only one storage area allocated for a type

•1-D Array

- Multiple storage areas allocated contiguously for the same type
- If used, an **Array Length** must be specified

Structure: The number of storage areas allocated for a type arranged contiguously.



APPENDIX

Defining Data Objects: Restriction

The screenshot shows the 'Object Editor' window with a table of data objects. The 'Restriction' column is highlighted with a red box. Below the table, the 'Restriction' dropdown menu is also highlighted with a red box, showing 'Constant' selected.

Icon	Name	Type	Restriction	Structure	Size
T	CHARARRAY	Character	Variable	1-D Array	34
#	INTARRAY	Integer	Variable	1-D Array	40
M	PART1	Label	None	None	0
T	CHAR1	Character	Constant	Single Value	1
#	INT1	Integer	Variable	Single Value	4

Name:

Type:

Structure:

Restriction:

Array Length:

Literal Value:

Description:

Buttons: Add, Modify, Delete, Clear, Help

•Constant

- Literal Value** required
- Value cannot be changed

•Variable

- No **Literal Value** required
- Value can be changed

•None

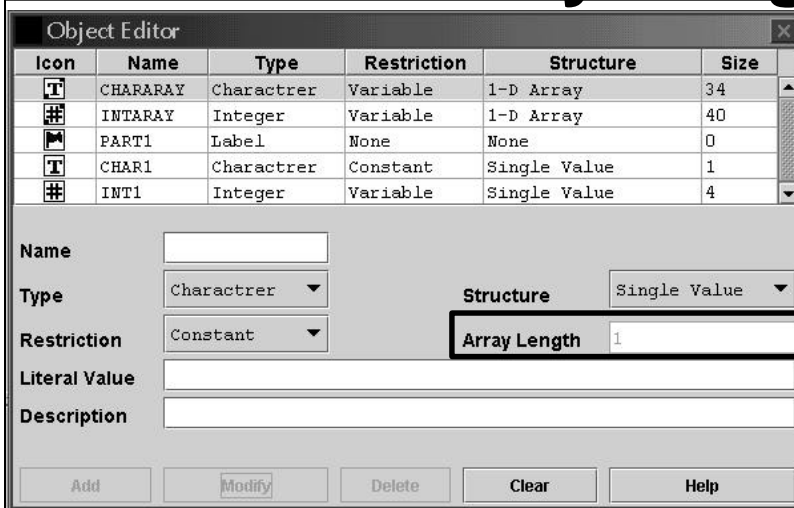
- A default value when the **Label** type is used

Restriction: Permission that determines whether or not a data object may be modified.



Electronic Visualization Laboratory University of Illinois at Chicago

Defining Data Objects: Array Length

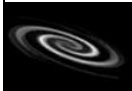


- Available only when **Structure** is 1-D Array

- The size of each array element is dependent on the memory needed to hold the values of a **Type**.

- The total size of the data object is the **Array Length** multiplied by the type size

Array Length: The number of contiguous allocated memory spaces to allow a data object to hold multiple values.



APPENDIX

Defining Data Objects: Literal Value

Icon	Name	Type	Restriction	Structure	Size
T	CHARARRAY	Characterer	Variable	1-D Array	34
#	INTARRAY	Integer	Variable	1-D Array	40
M	PART1	Label	None	None	0
T	CHAR1	Characterer	Constant	Single Value	1
#	INT1	Integer	Variable	Single Value	4

Name:
 Type:
 Restriction:
 Structure:
 Array Length:
 Literal Value:
 Description:
 Add Modify Delete Clear Help

•Values assigned are dependent on **Type** and **Structure**

•Single Value Character: One character surrounded by ". Escape character is /. (ex: 'a')

•Character Array: Characters surrounded by "". Maximum of 35 characters including escape character. (ex: "My Program").

•Single Value Integer: Any value between approx. -2 billion and +2 billion.

•Integer Array: Series of integers each separated by a comma (,) and surrounded by {}. (ex: {1, 2, 3, -4})

Literal Value: The value permanently assigned to a data object with the Restriction of constant.



Electronic Visualization Laboratory

University of Illinois at Chicago

APPENDIX

Defining Data Objects: Description

The screenshot shows the 'Object Editor' dialog box. It contains a table with the following data:

Icon	Name	Type	Restriction	Structure	Size
T	CHARARRAY	Characterer	Variable	1-D Array	34
#	INTARRAY	Integer	Variable	1-D Array	40
M	PART1	Label	None	None	0
T	CHAR1	Characterer	Constant	Single Value	1
#	INT1	Integer	Variable	Single Value	4

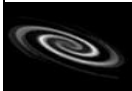
Below the table, there are several configuration options:

- Name:
- Type:
- Restriction:
- Structure:
- Array Length:
- Literal Value:
- Description:

At the bottom, there are buttons for 'Add', 'Modify', 'Delete', 'Clear', and 'Help'.

- Can contain any printable text
- Can have any length
- Can be used for any data object

Description: A text description that describes the purpose of the data object.



Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

Defining Data Objects: Icon

Object Editor

Icon	Name	Type	Restriction	Structure	Size
	CHARARRAY	Character	Variable	1-D Array	34
	INTARRAY	Integer	Variable	1-D Array	40
	PART1	Label	None	None	0
	CHAR1	Character	Constant	Single Value	1
	INT1	Integer	Variable	Single Value	4

Name:

Type:

Restriction:

Structure:

Array Length:

Literal Value:

Description:

Add Modify Delete Clear Help

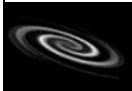
User-defined

- Integer Array
- Single Value Integer
- Character Array
- Single Value Character
- Label

System-defined

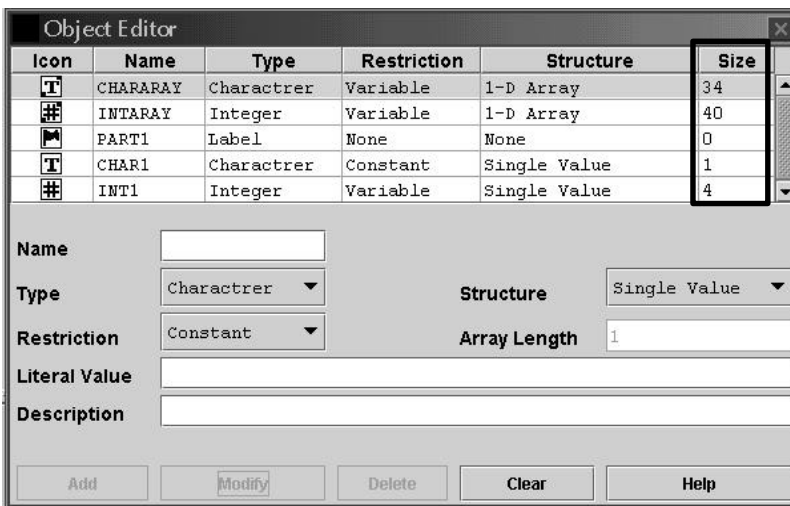
- Single Register
- Keyboard (input)
- Display (output)

Icon: Pictorial representation of the **Type** and **Structure** of the data object.



APPENDIX

Defining Data Objects: Size



The screenshot shows the 'Object Editor' window with a table of data object types. The 'Size' column is highlighted with a red box. Below the table are fields for Name, Type, Restriction, Structure, Array Length, Literal Value, and Description, along with buttons for Add, Modify, Delete, Clear, and Help.

Icon	Name	Type	Restriction	Structure	Size
T	CHARARRAY	Character	Variable	1-D Array	34
#	INTARRAY	Integer	Variable	1-D Array	40
M	PART1	Label	None	None	0
T	CHAR1	Character	Constant	Single Value	1
#	INT1	Integer	Variable	Single Value	4

- Each type has a certain size

 - Integer = 4 bytes

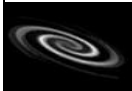
 - Character = 1 byte

- A Single Value data object will have the **Size** value of its type size

- A 1-D Array data object will use this formula:

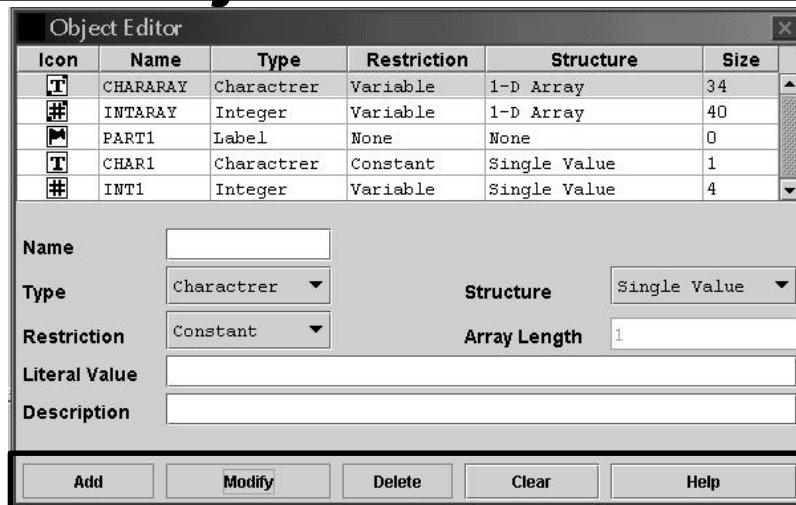
Array Length x Type Size

Size: *The amount of memory in bytes allocated to the data object. This value is not modified by you directly.*



Electronic Visualization Laboratory University of Illinois at Chicago

Defining Data Objects: Object Editor Action Buttons



•Add

- Create a new data object with a name not used before

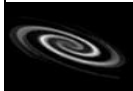
•Modify

- Change some or all properties of an **existing** data object

•Delete

- Remove an **existing** data object from the list of data objects

Function buttons: Submits the data in the fields for validation and entry into the list of data objects.



APPENDIX

Defining Data Objects: Object Editor Action Buttons

The screenshot shows the 'Object Editor' window. At the top is a table with columns: Icon, Name, Type, Restriction, Structure, and Size. Below the table is a form with fields for Name, Type, Restriction, Structure, Array Length, Literal Value, and Description. At the bottom are five buttons: Add, Modify, Delete, Clear, and Help.

Icon	Name	Type	Restriction	Structure	Size
T	CHARARRAY	Characterer	Variable	1-D Array	34
#	INTARRAY	Integer	Variable	1-D Array	40
P	PART1	Label	None	None	0
T	CHAR1	Characterer	Constant	Single Value	1
#	INT1	Integer	Variable	Single Value	4

Name:

Type: Structure:

Restriction: Array Length:

Literal Value:

Description:

Add Modify Delete Clear Help

•Clear

- Remove data from all of the fields and set drop-down boxes to default values

•Help

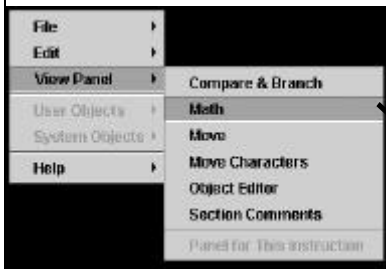
- Display help that explains how to use this panel

Function buttons: Submits the data in the fields for validation and entry into the list of data objects.

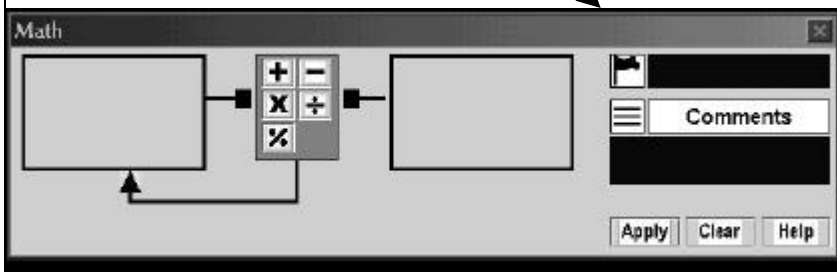


Electronic Visualization Laboratory University of Illinois at Chicago

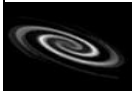
Using Panels: View Panel Menu



Select **View Panel** from the menu and choose a panel.

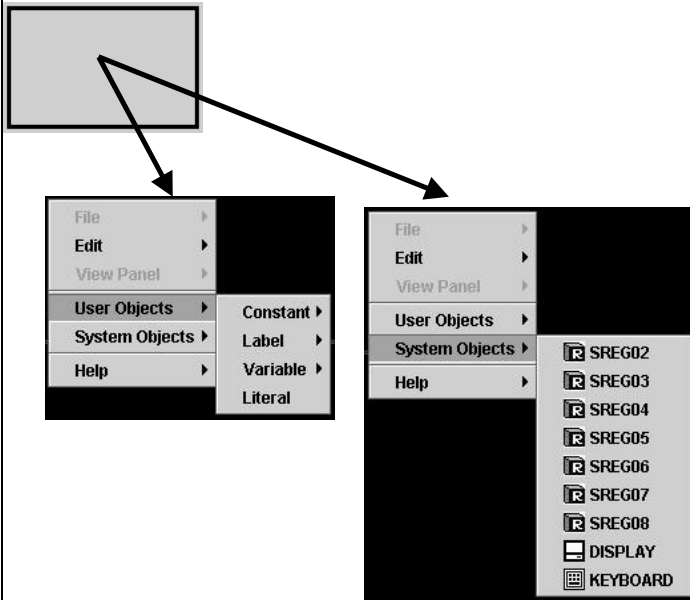


A panel will
panels have a
similar
functionality.



APPENDIX

Using Panels:



A menu will appear where **User Objects** or **System Objects** may

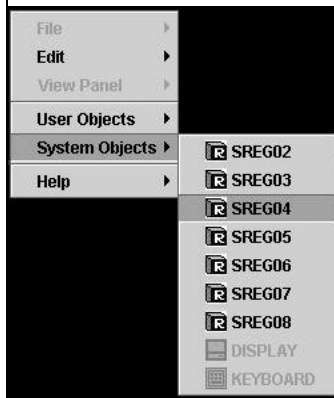
objects will appear in the appropriate

objects in **Objects** remains fixed.



APPENDIX

Using Panels:

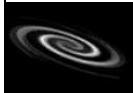


Select the desired object from the

appear in the rectangular area.



None/
Constant



Electronic Visualization Laboratory University of Illinois at Chicago

Using Panels: Clearing Values in Data Objects

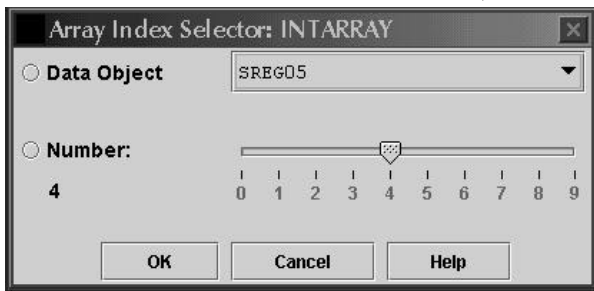
- Because data objects are not initialized at the start of the program with default values, ***it is the responsibility of the programmer to make sure the data objects have default values if necessary.***
- This means the following:
 - Assigning 0 to integer type and register data objects
 - Assigning an array of spaces to a character array
 - Assigning a space to a single character
- If these precautions are not taken, unexpected behavior may result in your program such as in the printing of unexpected characters or incorrect results from math calculations.



Using Panels: Selecting an Array Element



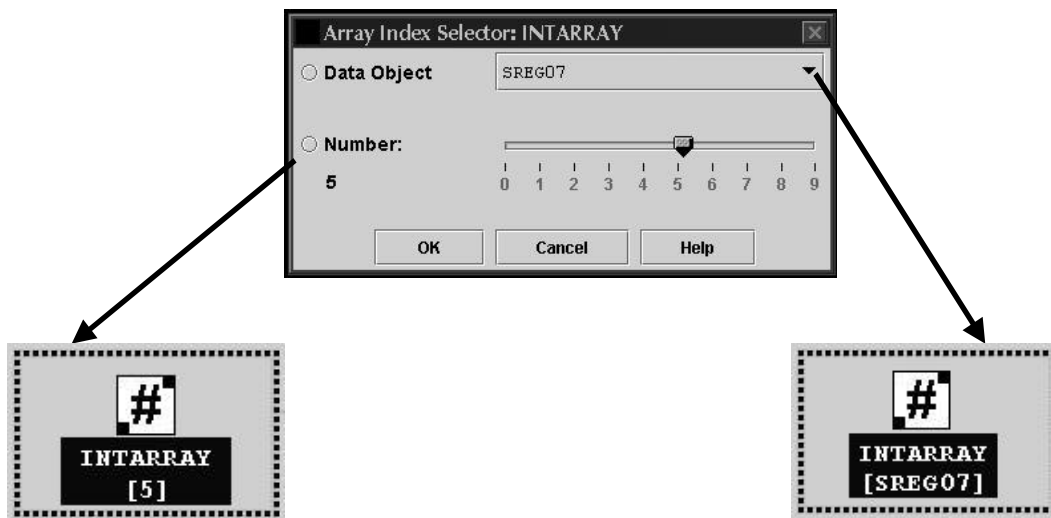
If you select an array structured data object, you may be prompted as to whether or not you want to select an array element.



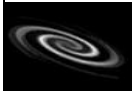
Select either a data object whose value in that instruction will represent the data object index, or select the index directly.



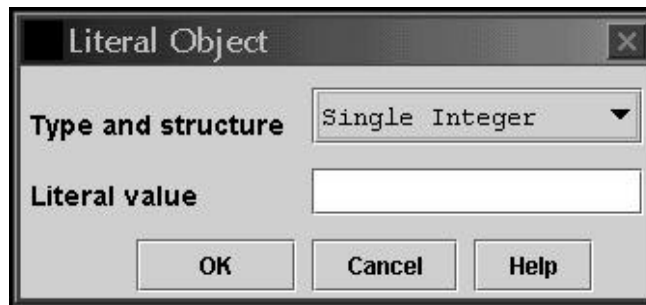
Using Panels: Selecting an Array Element



The result will appear in the rectangular area.



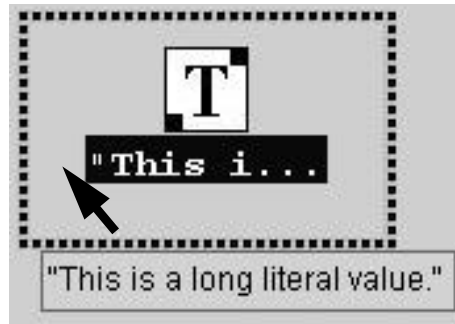
Using Panels: Entering a Literal Value



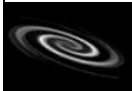
Based on the **Type and Structure** value selected from the drop-down list, enter a **Literal Value** that corresponds a legal value. For information on legal values, please view the slide named “**Object Editor: Data Object Literal Value**”



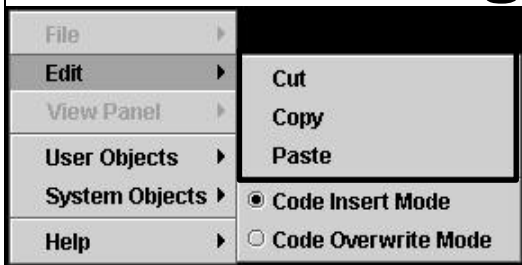
Using Panels: Entering a Literal Value



The literal value will appear in the rectangular area as it was entered. If the value is too long to be displayed in the limited space provided, move the cursor over this data object, pause, and text will appear that will display the complete literal value.



Using Panels: Editing Data Objects



It may be necessary to perform an **Edit** menu function on the data object. The **Edit** menu function uses two different buffers— one for data objects and another for instructions.

•Cut

- Remove the data object in the rectangular area and place it into the data object buffer.

•Copy

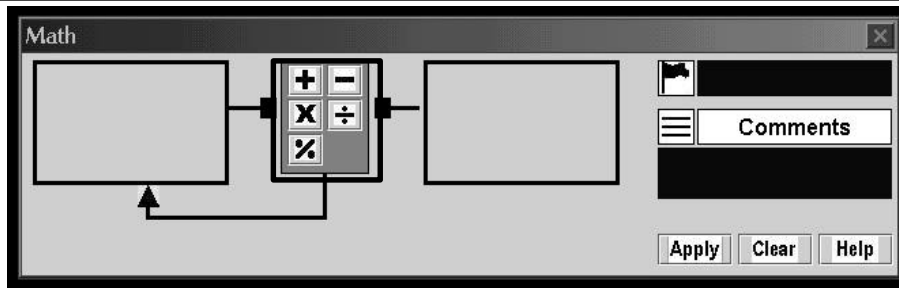
- Keep the data object in the rectangular area and place a duplicate of it into the data object buffer.

•Paste

- Place a duplicate of the data object in the data object buffer into the rectangular area.



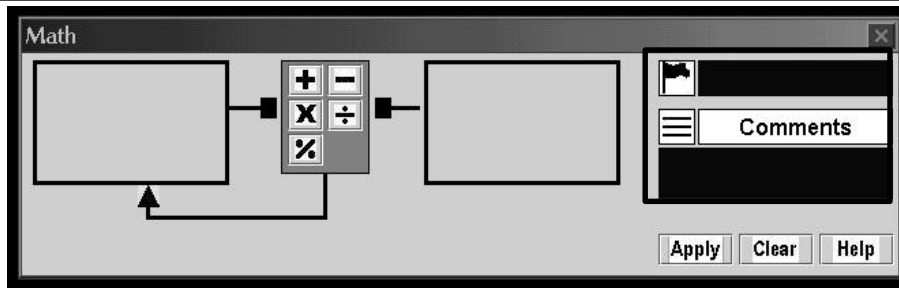
Using Panels: Function Button Panel



- Because a panel contains similar functions, it may be necessary to specify which function you would want to use. Simply press one of the toggle buttons in a button panel to select the function to perform on the operands.
- If you are uncertain about the meaning of a function icon, move your mouse over the button, pause, and text will appear describing the purpose of the function.



Using Panels: Label and Comments



•Label:

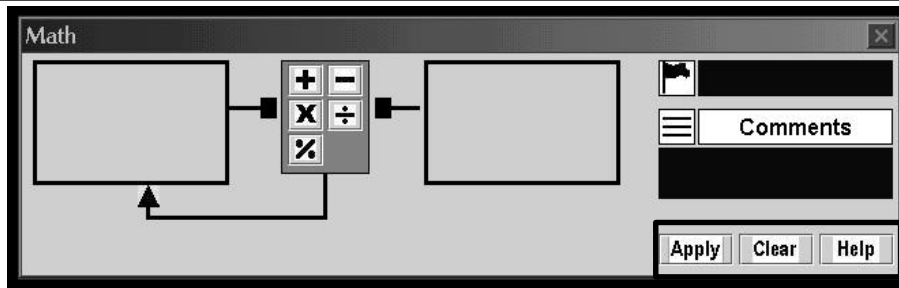
- This is the area for marking the instruction-to-be in the program. Using the **Compare and Branch** panel, you can jump to a specific area in a program marked by a label. A label may only be used **once** to identify a line of program code.

•Comments:

- Write a description about the instruction-to-be here. Space is limited to about two short lines of text. For more extensive comments, use the **Section Comments** panel.



Using Panels: Action Buttons



•Apply:

- Commit the instruction into the program at the location marked by the code cursor.

•Clear:

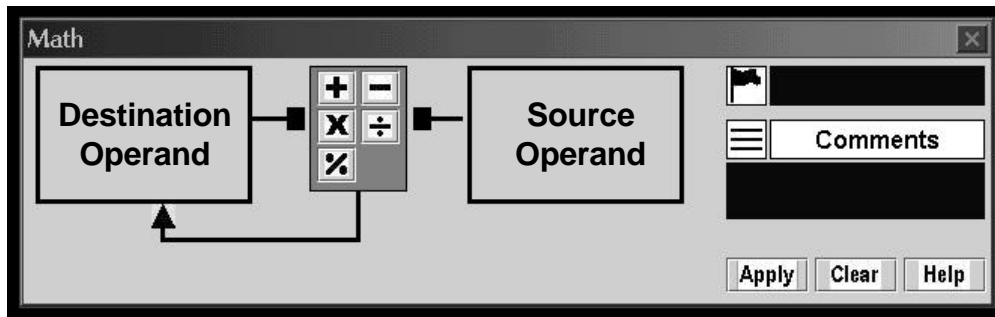
- Remove all data in all areas and, if necessary, any function button pressed will become unpressed.

•Help:

- Display information about how to use this panel.



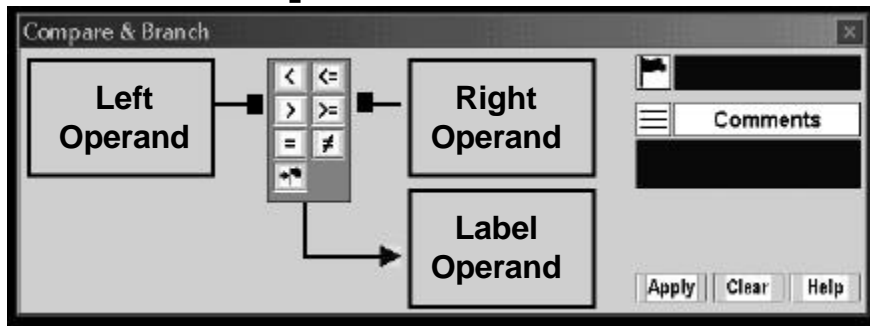
Instruction Panel: Math



- Perform addition, subtraction, multiplication, division, and modulo operations on operands.
- The destination operand receives the result of the operation (ex: $\text{LEFTINT} = \text{LEFTINT} \times \text{RIGHTINT}$).



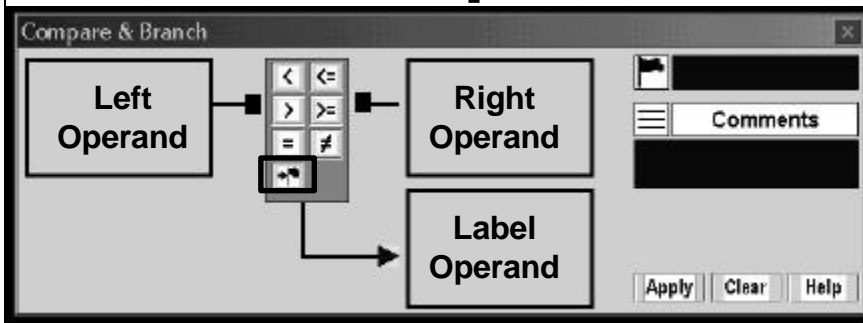
Instruction Panel: Compare & Branch



- Compare the values of two left and right operands of ***the same type and structure*** using the comparisons less than, less than or equal to, greater than, greater than or equal to, equal to, and not equal to.
- If the comparison is true, have program execution jump to the part of the program marked by the label given in this instruction; otherwise, continue execution with the next line of the program.

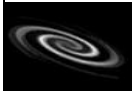


Instruction Panel: Compare & Branch

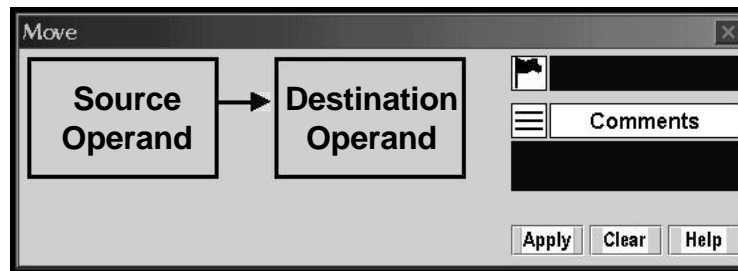


Unconditional
Branch Icon

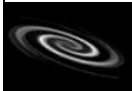
- If the function button with the arrow pointing to the flag is chosen, this indicates a unconditional branch. The program will begin execution at the part of the program marked with the label specified in this panel.
- No comparison operands are used for this function.



Instruction Panel: Move

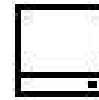


- Move the contents of the source operand to the contents of the destination operand. There are only a certain number of combinations of types and structures permitted.
- For operations involving converting a number to its text representation and vice versa, ***it is your responsibility to ensure that valid text or a valid number is used for the conversion.***

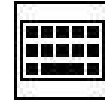


APPENDIX

Instruction Panel: Move



Display



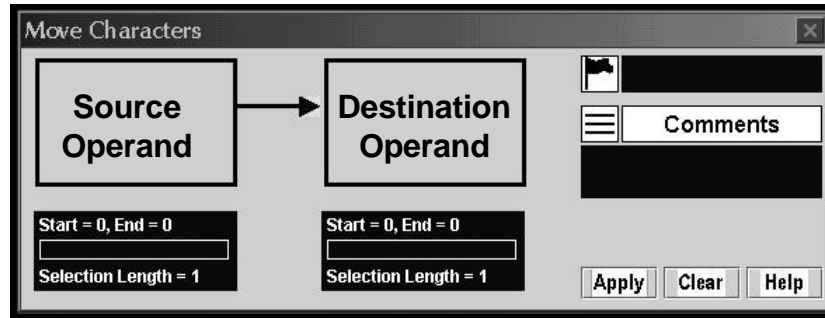
Keyboard

- If the source operand is the **Keyboard**, user input from the keyboard is stored in the destination operand.
- If the destination operand is the **Display**, data from the source operand is converted to its text equivalent representation and displayed on the screen.



Electronic Visualization Laboratory University of Illinois at Chicago

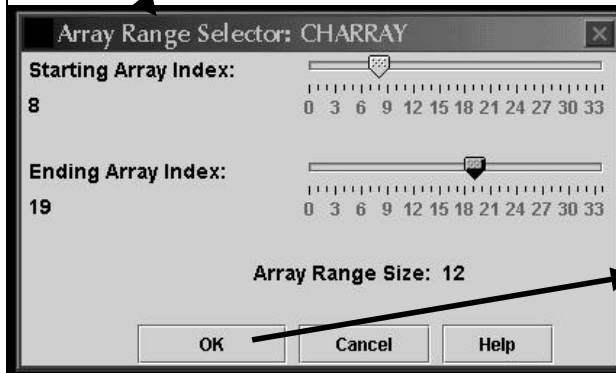
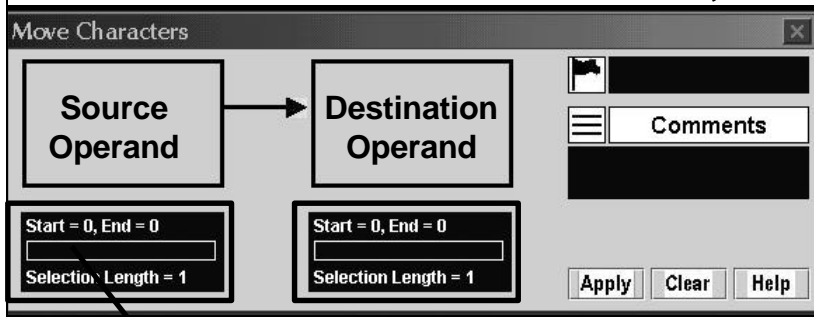
Instruction Panel: Move Characters



- Copy a range of characters from the source operand to replace a range of characters in the destination operand.
- The operands must be character arrays.
- The range of characters used in the source operand and destination operand must be the **same length**.



Instruction Panel: Move Characters, Array Range



Start = 8, End = 19
Selection Length = 12

- To select a range of characters, click one of the array range boxes with the **left mouse button**.

- Move the sliders to obtain the desired array range, and press **OK**.

- The result will appear in the array range box.



APPENDIX

Instruction Panel: Section Comments

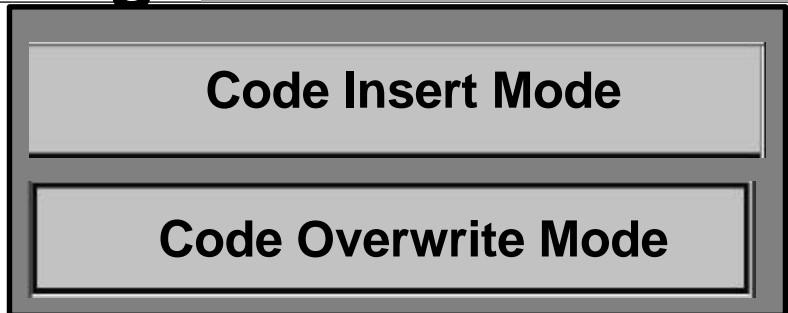
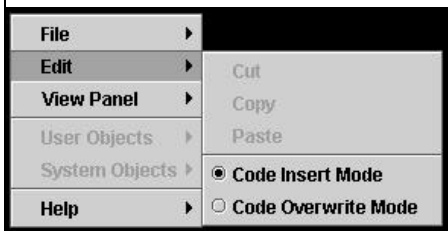


- Enter any text information about a part of the program in the text area.
- Can be of any length. Scrollbars will appear on the right side of the text box to accommodate viewing of all text typed.
- Will “serve” as an instruction in the code but ***has no effect on the execution of the program.***



Electronic Visualization Laboratory University of Illinois at Chicago

Writing a Program: Editing Modes

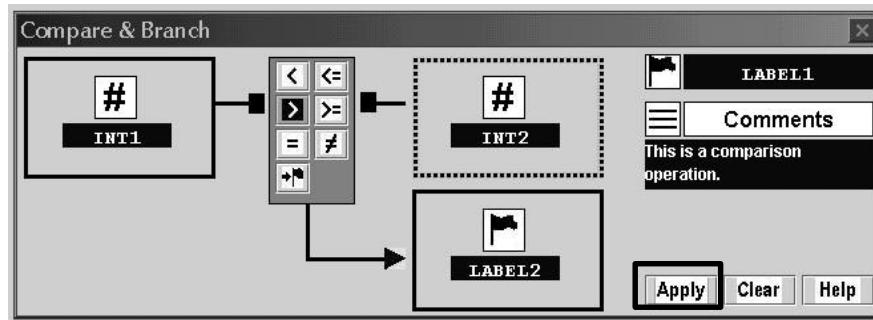


- Select an editing mode from the **Edit** menu.
 - Code Insert Mode** will place the newest instruction in the line with the cursor and shift all other instructions down.
 - Code Overwrite Mode** will replace the old instruction in the line with the cursor with the newest instruction. Good for editing a single line of the program. Cannot overwrite the last empty line of code.



APPENDIX

Writing a Program: Visual Code Layout

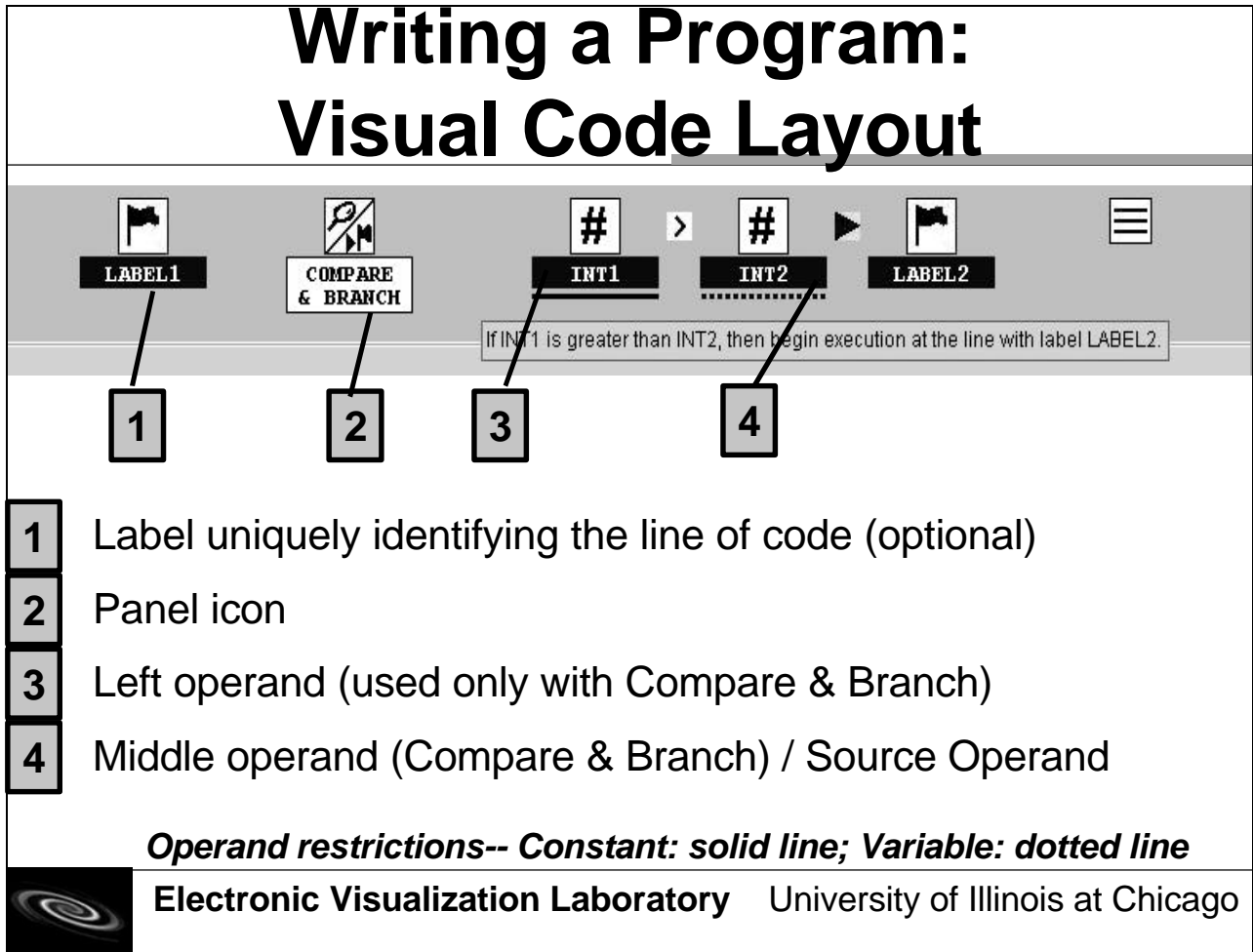


Press the **Apply** button in the instruction panel to add a line to the program.



Electronic Visualization Laboratory University of Illinois at Chicago

Writing a Program: Visual Code Layout



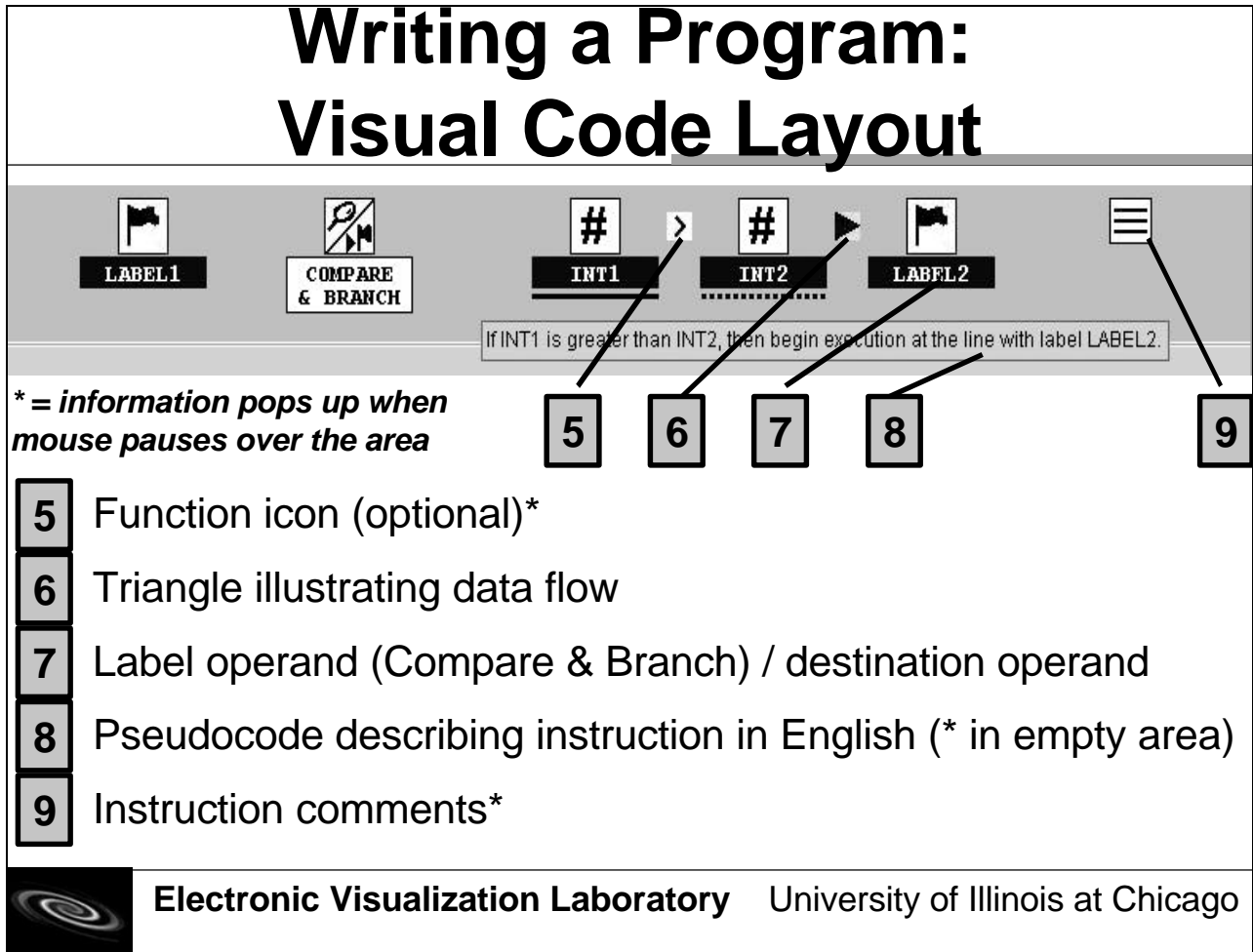
- 1 Label uniquely identifying the line of code (optional)
- 2 Panel icon
- 3 Left operand (used only with Compare & Branch)
- 4 Middle operand (Compare & Branch) / Source Operand

Operand restrictions-- Constant: solid line; Variable: dotted line

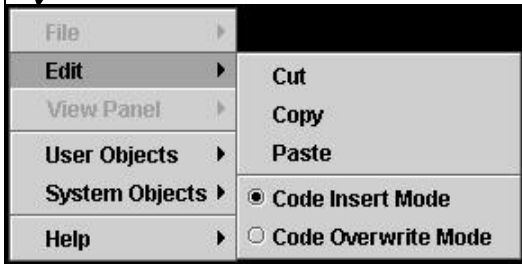
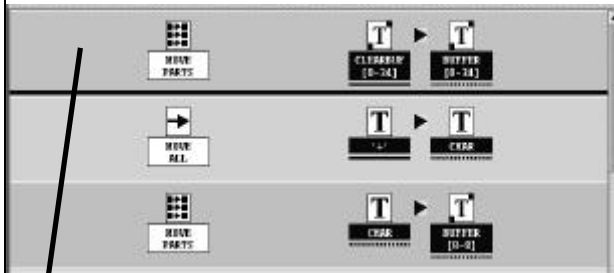


Electronic Visualization Laboratory University of Illinois at Chicago

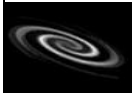
Writing a Program: Visual Code Layout



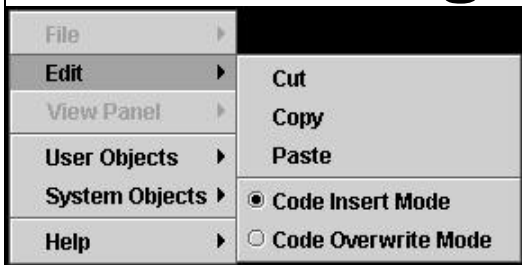
Editing a Program: General Procedure



- Highlight the rows of code to edit.
- Right-click on any part of the visual code area to bring up the menu seen here.
- Select the menu function of your choice.
- If performing a **Copy** or **Paste**, move the cursor to the desired location (this is important)
- Select **Copy** or **Paste** from the menu



Editing a Program: Using the Edit Menu



If more than one line of code is highlighted and the paste operation is performed, the copied lines of code will first overwrite the highlighted lines of code. Then, if there are still more lines of code, the remaining copied lines of code will be inserted into the program.

•Cut

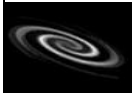
- Remove the line(s) of code from the program and place the line(s) into the code buffer. Remaining code shifts upward.

•Copy

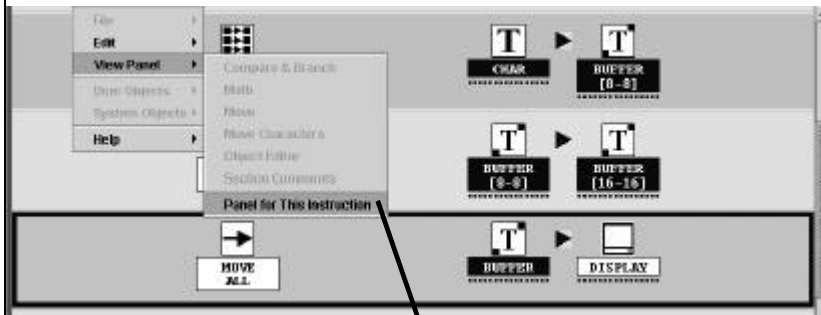
- Keep the line(s) of code in the program and place a duplicate of the line(s) of code into the code buffer.

•Paste

- Place a duplicate of the line(s) of code into the program at the cursor location. Remaining code shifts downward.



Editing a Program: Correcting a Line of Code

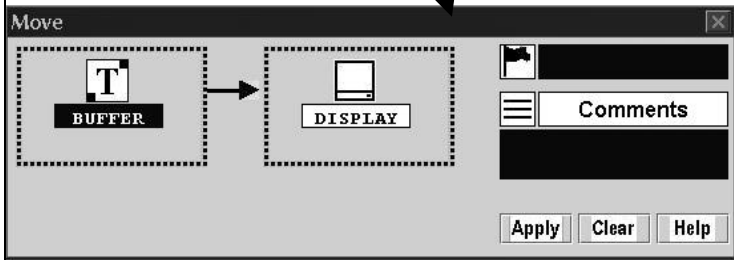


- Enter **Code Overwrite Mode**.

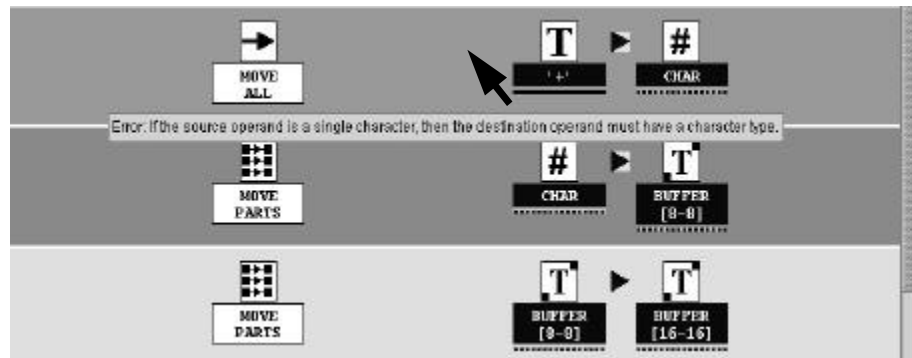
- Move the cursor to the program line that you want to edit.

- Select **Panel for This Instruction** from the **View Panel** menu.

- The appropriate panel will appear with all of the operands, function button selection, comments, and label.



Editing a Program: Finding an Error



- Move the mouse cursor over the erroneous line and pause.

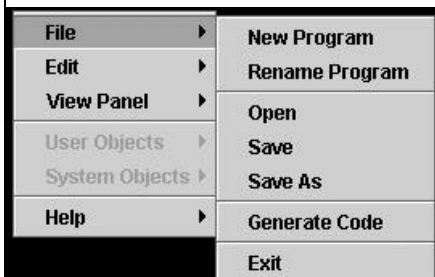
- Text will appear indicating the reason for the error.

- Correct the error by editing the line or by changing data object properties in the **Object Editor**.

Most errors are detected before you are able to add a line of code to a program. However, it is possible to receive errors later. For instance, if a Single Value Character named *CHAR* has its type changed to Integer, it is likely that any line of code using *CHAR* will have an error. If a line has an error, the background of the line will become a shade of red. When the error has been corrected through editing, it will no longer have a red color.



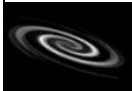
Save Program



- **Save As:** Select a file from the list of available Panel-Tools-Views files (.ptv), or type the name of the file in the **File Name** field (no extension is required). If the file name already exists, you will be asked if the new version should replace the old version.

- **Save:** The first time the program is saved, it will act as **Save As**. Otherwise, file will save without any other intervention.

- ***The name of the file does not have to resemble the program name but must adhere to the file naming conventions of the OS.***



APPENDIX

Save Program



Panels :: Tools :: Views >> ARRAY [unsaved]

If a file has not been saved, the word “unsaved” surrounded by [] will appear after the name of the program.



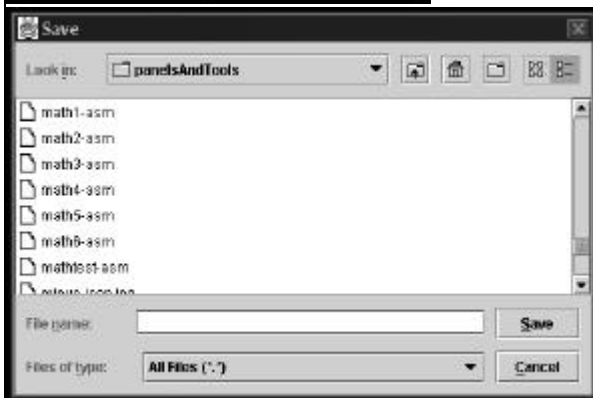
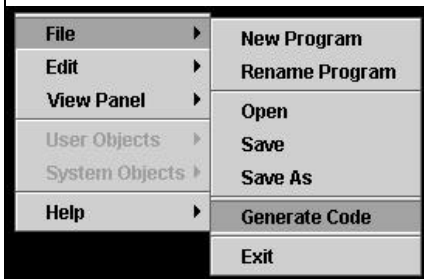
Panels :: Tools :: Views >> ARRAY

Once the file has been saved, the “unsaved” surrounded by [] will disappear.



Electronic Visualization Laboratory University of Illinois at Chicago

Generate Assembly Code



- This functions the same as saving the file except that the saved file has naming restrictions tied to MS-DOS file naming conventions. No filename extensions should be used.

- Assembly code will be written to the given file **only if** there are no errors in the code. A message will appear indicating if the file was created.

- To view the assembly language file, open the file in your favorite text editor.



Running the Assembly Language Program



- Open a program named **CAS.EXE**
- Select **Run a Program** from the menu by pressing the “R” key.
- Enter the name of the assembly file (use full pathname if necessary, or copy the file to the directory of **CAS.EXE**).



Running the Assembly Language Program

```

CAS
PSW AT BREAK FFC50000 0F000000
R0-7: F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4
R8-15: F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 00000020 00000068 00000000

000000 5810F010 582F0014 1A125020 0003501F *+0.....6...&+
000010 001807FE 00000004 00000006 F5F5F5F5 *+++++++5555+
000020 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
000030 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
000040 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
000050 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
000060 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
000070 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
000080 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
000090 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
0000A0 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
0000B0 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
0000C0 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
0000D0 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
0000E0 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+
0000F0 F5F5F5F5 F5F5F5F5 F5F5F5F5 F5F5F5F5 *5555555555555555+

====> B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(step), T(race)
    
```

•The important aspect is to interact with the program. So, if you see a screen similar to the one here, press the “G” key at the prompt to **Go** to continue execution of the program.

•Repeat this each time this screen appears.

•One execution is complete, you should return to the main menu screen.

Once execution is complete, you can view a “log file” of the execution in a text file (.txt) having the same name as the assembly language program.

APPENDIX

Finding Errors in the Assembly Language Program

- If there are errors in the program, execution may stop prematurely or an error message will appear. To fix this error, open the text file (ex. PROG1.TXT, where PROG is the name of the assembly language program), and search for the error.

```
ASSIST/I Version 2.03, Copyright 1984, BDM Software.

      LOC  OBJECT CODE      ADDR1 ADDR2  STMT   SOURCE STATEMENT
000000                                1 TESTERR  CSECT
000000                                2          USING TESTERR,SREG15
000000 1948                                3          CR   SREG04,SREG08  This line will ca
*
000002  F5F5 F5F5                                4          BC   B'0100',LABEL1  If less than, br
$
Error # 130, UNDEFINED SYMBOL
```

- The error is highlighted here. A \$ appears at the location of the error followed by the error message below.



Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

Finding Errors in the Assembly Language Program

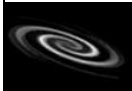
```
AS818T/I Version 1.03, Copyright 1984, EDS Software.

LOC  OBJECT CODE  ADDR1 ADDR2  STMT  SOURCE STATEMENT
000000                                1  TESTER:  C8C7
000000                                2  USING TESTER,REG15
000000 1940                                3  CR  SREG04,SREG08  This line will ca
000002 F5F5 F5F5                                4  BC  B'0100',LABEL1  If less than, br
Error V 130 UNDEFINED SYMBOL
```

After you have determined an error, edit the program in PTV to correct the error. Then repeat the process of generating the assembly code and running the assembly language program.

- The error is caused implicitly. Labels cannot be defined, so when a label is used in **Compare & Branch**, a line of code **must exist** that is identified by this label; otherwise, an error like this will occur.

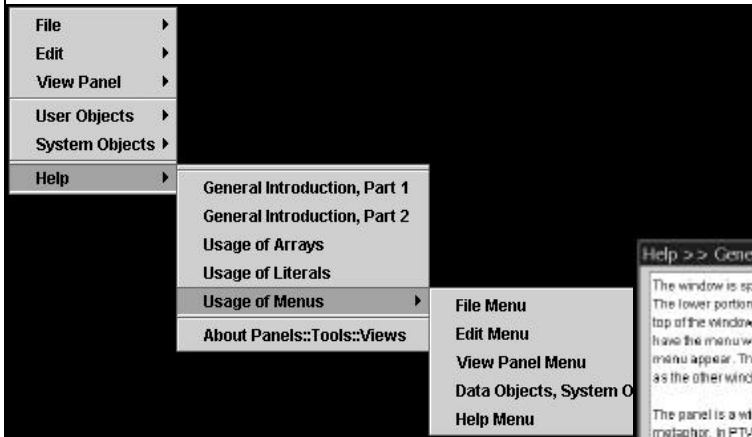
- From this, a lesson should be learned that not every error will be caught by PTV (although this one could have), but most will. The awareness of how the assembler works is important in debugging errors.



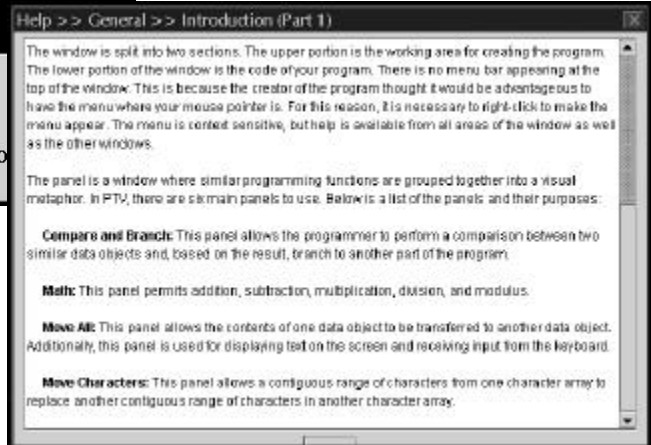
Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

Help is Available



•While using the program, simply select the **Help** menu to obtain help on a number of different topics discussed in this introduction.



•A window will appear containing the information you requested.



Electronic Visualization Laboratory University of Illinois at Chicago

APPENDIX

Enjoy coding.



Electronic Visualization Laboratory University of Illinois at Chicago

CITED LITERATURE

- Ambler, Allen L. and Burnett, Margaret M.: Influence of Visual Technology on the Evolution of Language Environments. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 19-32. Los Alamitos, CA, IEEE Computer Society Press. 1990.
- Beaumont, Mark and Jackson, David: Low Level Visual Programming. In: Proceedings of the 1997 IEEE International Symposium on Visual Languages, Isle of Capri, Italy.
- Borning, Alan: The Programming Language Aspects of ThingLab, a Constraint-Orientated Simulation Laboratory. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 416-449. Los Alamitos, CA, IEEE Computer Society Press. 1990a.
- Borning, Alan: Graphically Defining New Building Blocks in ThingLab. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 450-469. Los Alamitos, CA, IEEE Computer Society Press. 1990b.
- Chang, S. K., Burnett, Margaret M., Levialdi, Stefano, Marriott, Kim, Pfeiffer, and Joseph J., Tanimoto, Steven L.: The Future of Visual Languages. In: Proceedings of the 1999 IEEE International Symposium on Visual Languages, Tokyo, Japan.
- Cypher, Allen, Halbert, Daniel C., Kurlander, David, Lieberman, Henry, Maulsby, David, Myers, Brad A., and Turransky, Alan: Watch What I Do: Programming by Demonstration. MIT Press, 1993.
- Duntemann, Jeff: "The Future of Programming," *Visual Developer*, June-July-August 1998, 46-58.
- Finzer, William and Gould, Laura: Programming by Rehearsal. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 356-366. Los Alamitos, CA, IEEE Computer Society Press. 1990.
- Frei, H. P., Weller, D. L., Williams, R.: A Graphics-Based Programming-Support System. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 252-258. Los Alamitos, CA, IEEE Computer Society Press. 1990.
- Glinert, Ephraim P. and Tanimoto, Steven L.: Pict: An Interactive Graphical Programming Environment. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 265-283. Los Alamitos, CA, IEEE Computer Society Press. 1990.

CITED LITERATURE (continued)

- Halbert, Daniel C.: SmallStar: Programming by Demonstration in the Desktop Metaphor. In: Watch What I Do: Programming by Demonstration, ed. Allen Cypher, pp. 103-123. Cambridge, MA, MIT Press. 1993.
- Jackiw, R. Nicholas and Finzer, William F.: The Geometer's Sketchpad: Programming by Geometry. In: Watch What I Do: Programming by Demonstration, ed. Allen Cypher, pp. 293-307. Cambridge, MA, MIT Press. 1993.
- Jung, Matthias T., Kastens, Uwe, Schindler, Christian, Schmidt, Carsten: A Pattern-Based Generator for Implementation of Visual Languages. In: Proceedings of the 2000 IEEE International Symposium on Visual Languages, Seattle, Washington.
- Kurlander, David: Chimera: Example-Based Graphical Editing. In: Watch What I Do: Programming by Demonstration, ed. Allen Cypher, pp. 271-290. Cambridge, MA, MIT Press. 1993.
- Lieberman, Henry: Tinker: A Programming by Demonstration System for Beginning Programmers. In: Watch What I Do: Programming by Demonstration, ed. Allen Cypher, pp. 49-64. Cambridge, MA, MIT Press. 1993.
- Lieberman, Henry: Your Wish Is My Command: Programming by Example. Morgan-Kaufmann Publishers, 2001.
- Ludoph, Frank, Chow, Yu-Ying, Ingalls, Dan, Wallace, Scott, Doyle, Ken: The Fabrik Programming Environment. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 405-413. Los Alamitos, CA, IEEE Computer Society Press. 1990.
- Maulsby, David and Turransky, Alan: A Programming by Demonstration Chronology: 23 Years of Examples. In: Watch What I Do: Programming by Demonstration, ed. Allen Cypher, pp. 527- Cambridge, MA, MIT Press. 1993.
- McDaniel, Richard: Demonstrating the Hidden Features that Make an Application Work. In: Your Wish Is My Command: Programming by Example, ed. Henry Lieberman, pp. 163-174. San Francisco, CA, Morgan-Kaufmann Publishers. 2001.
- Myers, Brad A.: Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 33-40. Los Alamitos, CA, IEEE Computer Society Press. 1990.
- Myers, Brad A.: Demonstrational Interfaces: A Step Beyond Direct Manipulation. In: Watch What I Do: Programming by Demonstration, ed. Allen Cypher, pp. 485-512. Cambridge, MA, MIT Press. 1993.

CITED LITERATURE (continued)

- Ota, Yukio: Historical Role and Capability of Visual Languages. In: Proceedings of the 1999 IEEE International Symposium on Visual Languages, Tokyo, Japan.
- Pong, M. C. and Ng, N.: PIGS-- A System for Programming with Interactive Graphical Support. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 324-333. Los Alamitos, CA, IEEE Computer Society Press. 1990.
- Potter, Richard: Triggers: Guiding Automation with Pixels to Achieve Data Access. In: Watch What I Do: Programming by Demonstration, ed. Allen Cypher, pp. 361-380. Cambridge, MA, MIT Press. 1993.
- Reiss, Steven P.: PECAN: Program Development Systems that Support Multiple Views. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 259-264. Los Alamitos, CA, IEEE Computer Society Press. 1990.
- Repenning, Alexander and Perrone, Corrina: Programming by Analogous Examples. In: Your Wish Is My Command: Programming by Example, ed. Henry Lieberman, pp. 351-369. San Francisco, CA, Morgan-Kaufmann Publishers. 2001.
- Rubin, Robert V., Golin, Eric J., Reiss, Steven P.: ThinkPad: A Graphical System for Programming by Demonstration. IEEE Software 2:2:73-79. 1985.
- St. Amant, Robert, Lieberman, Henry, Potter, Richard, Zettlemoyer, Luke: Visual Generalization in Programming by Example. In: Your Wish Is My Command: Programming by Example, ed. Henry Lieberman, pp. 371-385. San Francisco, CA, Morgan-Kaufmann Publishers. 2001.
- Sheehan, Robert. Lower Floor, Lower Ceiling: Easily Programming Turtle-Graphics. In: IEEE Symposium on Visual Languages, Seattle, Washington, 2000.
- Shneiderman, Ben: Direct Manipulation: A Step Beyond Programming Languages. Computer 16:8:57-69. 1983.
- Shu, Nan. C.: Visual Programming Languages: A Perspective and a Dimensional Analysis. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 41-58. Los Alamitos, CA, IEEE Computer Society Press. 1990.
- Smith, D. C.: Principles of Iconic Programming. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 216-251. Los Alamitos, CA, IEEE Computer Society Press. 1990.

CITED LITERATURE (continued)

- Smith, David Canfield: Novice Programming Comes of Age. In: Your Wish Is My Command: Programming by Example, ed. Henry Lieberman, pp. 351-369. San Francisco, CA, Morgan-Kaufmann Publishers. 2001.
- Stako, John et al.: Software Visualization: Programming as a Multimedia Experience. MIT Press, 1998.
- Sun Microsystems: "The NetBeans Platform," *About the NetBeans Platform*, 29 July 2002a, (31 October 2002). <<http://www.netbeans.org/about/platform/index.html>>.
- Sun Microsystems: "An Introduction to the NetBeans IDE," *About the NetBeans IDE*, 3 March 2002b, (31 October 2002). <<http://www.netbeans.org/about/ide/index.html>>.
- Sutherland, Ivan E.: Sketchpad: A Man-Machine Graphical Communication System. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 198-215. Los Alamitos, CA, IEEE Computer Society Press. 1990.
- Swaine, Michael: "Getting Skinned," *Dr. Dobb's Journal*, April 2001, 160.
- Traynor, Carol and Williams, Marian G.: End Users and GIS: A Demonstration Is Worth a Thousand Words. In: Your Wish Is My Command: Programming by Example, ed. Henry Lieberman, pp. 115-133. San Francisco, CA, Morgan-Kaufmann Publishers. 2001.

BIBLIOGRAPHY

Burnett, Margaret M, Baker, Maria J., Carlson, Paul, Yang, Sherry, and van Zee, Pieter: Scaling Up Visual Programming Languages. Computer 28:3:45-51. 1995.

Glinert, Ephraim P.: Visual Programming Environments: Paradigms and Systems. IEEE Computer Society Press, 1990.

Horstmann, Cay S. and Cornell, Gary: Core Java 1.2, Volume I – Fundamentals. Sun Microsystems Press, 1999.

Proceedings of the 1999 IEEE International Symposium on Visual Languages, Tokyo, Japan.

Proceedings of the 2000 IEEE International Symposium on Visual Languages, Seattle, Washington.

Schach, Stephen R: Classical and Object-Orientated Software Engineering with UML and C++. WCB/McGraw-Hill, 1999.

Shneiderman, Ben: Designing the User Interface: Strategies for Effective Human-Computer Interaction. Addison-Wesley, 1998.

Tanimoto, Steven L. and Runyan, Marcia S.: PLAY: An Iconic Programming System for Children. In: Visual Programming Environments: Paradigms and Systems, ed. Ephraim P. Glinert, pp. 367-377. Los Alamitos, CA, IEEE Computer Society Press. 1990.