Frameworks for Assessing Visual Languages

Tim Menzies *

December 22, 1995

Abstract

We present a theoretical framework for comparing visual languages. This framework was developed in order to teach an introductory visual programming subject. The subject aims at teaching students general principles of visual programming rather than just the details of a particular visual programming systems. To support these aims, we also developed an evaluation framework for visual programming concepts. Such frameworks would be useful for other educators as well as novices to visual programming who want a quick introduction to the numerous publications in this field.

1 Introduction

A summary of the visual programming (VP) field is a report of an emerging field which is still developing its theoretical foundations (though see [5] for an interesting approach to a precise theory of VP). An introduction to VP must therefore be an *exploration* of an emerging field, rather than a *report and summary* of an established field.

This paper presents two overview frameworks developed for SFT5030, a one semester postgraduate VP subject run by the Department of Software Development of Monash University. This subject adopts the exploratory approach. In SFT5030, students are encouraged to actively critique the claims on any particular paper. An evaluation framework was provided allowing students to assess VP systems (see Section 3). Lectures presented different classic VP systems (usually from [10, 11]), using the theoretical framework. Students had to select their own VP system and present it as a seminar using the same theoretical framework. In tutorials, students assessed commercially available VP systems (iTHINK, VisualAge, and the QBE editor of Microsoft Access). A theoretical framework was also provided that allowed students to compare different VP systems (see Section 2). This theoretical framework is based on (i) short overviews of the VP field [5, 18]; (ii) evaluation criteria for different VP systems [26, 36]; (iii) collections that "snapshot" the state-of-the-art in VP at various times [10, 11, 17, 19, 32]; (iv) the anecdotal notes (and an excellent bibliography) found in the `comp.visual.languages` frequently asked questions (FAQ) list; and (v) discussions with SFT5030 students.

The rest of this paper presents the theoretical and evaluation frameworks. An appendix shows the evaluation of a commercial VP system (the composition editor of VisualAge) using the theoretical framework.

2 A Theoretical Framework for VP

As a rough rule-of-thumb, a visual programming system is a computer system whose execution can be specified *without scripting* except for entering unstructured strings such as `''Monash University Banking Society''` or simple expressions such as `a > 7`. Visual representations have been used for many years (e.g. Venn diagrams) and even centuries (e.g. maps). Executable visual representations, however, have only arisen with the advent of the computer. With falling hardware costs, it has become feasible to build and interactively manipulate intricate visual expressions on the screen.

More precisely, a non-visual language is a one-dimensional stream of characters while a VP system uses at least two dimensions to represent its constructs [5]. We distinguish between a *pure VP* system and a *visually supported* system:

*Dept. of Software Development, Monash University, Caulfield East, Melbourne, VIC, Australia, 3145; +61-3-9903-1033; +61-3-9903-1077(fax);
Email: timm@insect.sd.monash.edu.au;
URL: http://www.sd.monash.edu.au/~timm

- A *pure VP system* must satisfy two criteria. *Rule #1*: the system must execute. That is, it is more than just a drawing tool for software or screen designs. *Rule #2*: the specification of the program must be modifiable within the system's visual environment. In order to satisfy this second criteria, the specification of the executing program must be configurable. This modification must be more than just (e.g.) merely setting numeric threshold parameters.

- There exists a class of VP systems that are not pure, but are *visually supported* systems. Most commercial VP systems are not pure VP systems, such as VISUAL BASIC, DELPHI, and VisualAge. For more details on visually supported systems, see Section 2.3.

A very useful feature of a VP system is *direct manipulation* [35]. A direct manipulation interface:

- Makes directly visible the object of interest;

- Supports rapid, reversible, and incremental actions;

- Replaces complex command structures by direct manipulation of the object of interest;

- Supports a spiral model of learning. Minimal initial knowledge is required to start with the system. Complex operations can be learnt subsequently, as required.

Note that direct manipulation is not a sufficient condition for calling a programming system "visual". Rather, it is a pragmatically useful feature of a productive interface.

In the theoretical framework, two basic tools are offered to students for analysing a visual system. Figure 1 shows a sample "Shu Triangle" [36]. Shu triangles are discussed in Section 2.1 and Section 2.6.

Figure 2 shows three dimensions along which we can characterise VP systems: their expressions, their purpose, and their design. These dimensions are discussed in Sections 2.2, 2.3, and 2.4.

2.1 The Shu Triangle

Shu [36] defines three comparative criteria for assessing VP systems: visual extent, language level, and scope (see Figure 1.i). While other VP classification schemes may be more up-to-date or more specific (e.g. [17]), we have found Shu's criteria is insightful and simple for novices to the VP field.

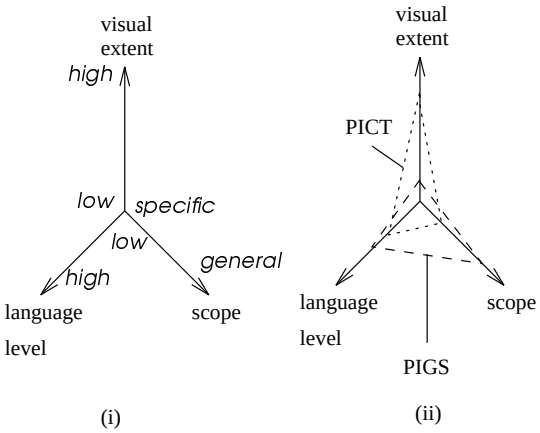

Figure 1: The Shu triangle

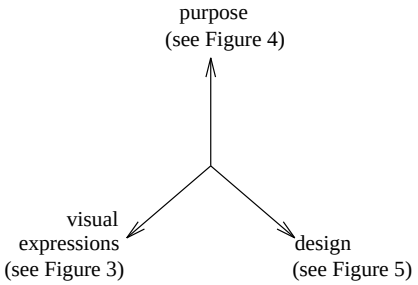

Figure 2: Dimensions of a VP system

We explain the Shu triangle to students as follows:

- *Visual extent* refers to the intricacy of the visual entities used by the system. For example, a system with only text has a very low (? zero) visual extent while a virtual reality system scores off the dial on visual extent.

- *Language level* is measured as the inverse of the effort required to perform a particular task. For example, compare the effort involved coding a database query in raw C with the same query in SQL. The programming would have to type more in C. Therefore, for this task, C has a lower language level than SQL.

- *Scope* measure the generality of the VP system. In Shu's terminology, scope moves from a low value of *specific* to a high value of *general*. For example, there exist a class of VP systems which can only be customised by coding in a traditional text-based programming language (e.g. Xerox Star [29], and VisualAge- see appendix). These systems are less general than VP systems where

new sub-routines can be specified visually (e.g. THINGLAB [3]).

Roughly speaking, the language level is a measure of what can be done with the VP system in a limited amount of time (say, an hour) while the scope is a measure of what can be done with the VP system in an unlimited time.

The Shu triangle is a relative criteria. It can only be meaningfully used when two or more criteria are simultaneously displayed. For example, Figure 1.ii compares a graphical structure-chart editor/interpreter (PICT [12]) to a Nassi-Shneiderman diagram editor/interpreter (PIGS [28]). PICT has a very specific scope due to the limitations of the language constructs which it can handle. The graphics used for the Nassi-Shneiderman graphics in PIGS are fixed while the PICT visual editor is a general point-and-click icon editor. Therefore PICT has a higher visual extent than PIGS. Both have similar language level, but Shu believes that PIGS can support a wider variety of language constructs. Hence, Shu argues that PIGS has a higher language level.

Due to the comparative nature of the Shu triangle, students must be given some initial system to "calibrate" the triangle. In the first tutorial of SFT5030, students spend an hour building database queries with Microsoft Access's visual query editor. Once this system was familiar to students, they could use it as the baseline for future comparisons.

Despite the merits of the Shu triangle for teaching purposes, we found that we had to further refine its criteria. These refinements are presented in Section 2.6, after an expansion of our theoretical framework.

2.2 Expressions in a VP System

Visual expressions are of at least six types in increasing order of visual extent: text, simple forms, tables, icons, diagrams, and "other" (see Figure 3).

Purely text-based systems have the lowest-level of visual expressiveness. Simple form-based systems are slightly more expressive. These systems let the user "fill-in-the-blanks" of some prototype specification to generate a more specific specification. Simple form-based systems may require very limited graphical support and can be developed on character-based screens. Such character-based screens limit the representation of (e.g.) two-dimensional graphs. Hence, simple form-based systems are low-end VP systems.

We take care to distinguish between "simple

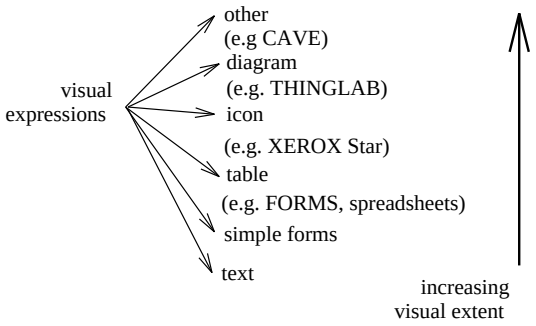

Figure 3: Visual expression types

form-based" systems and more general form-based systems such as Ambler's FORMS system [1]. Ambler argues that forms are a really an extension of the tabular/spreadsheet expression. FORMS uses a sophisticated interface in which users can specify the properties of adjacent cells by a single mouse drag across multiple cells.

Tabular expressions make extensive use of the position of their cells. For example, a spreadsheet cell can be the sum of the cell above and the cell immediately to the left. Having mentioned spreadsheets, we stress that the current generation of commercially available spreadsheet packages are weak examples of VP systems. All non-trivial spreadsheet applications require the use of intricate syntax to define formulae or macros. A better example of the use of tabular expressions is the original QBE system [41]. QBE allows a user to "draw" a database query on a character-based screens. The drawing is a little table that reflects the relational structure of the database being queried. Projects and selects can be specified by filling in the cells of the drawn table with restrictions or matches for its values. Joins can be specified by drawing the joined tables, then using the same variable names in the different tables.

In icon-based languages (e.g. Xerox Star [29]), the position of the icon usually does not effect the services offered by that icon. Typically, users can click on the icon to access a menu of services. However, moving the icon around the screen can represent the transfer of data or the application of some function to some data (e.g. moving a file between a directory).

Diagrammatic systems utilise a wide variety of pictures in their interface. Diagrammatic systems are characterised by "plug-and-play"; i.e. the user creates an diagram by linking up visual components offered from a palette. Often, users can create the visual analogue to a sub-routine by batching up a commonly used diagram into a single icon.

For example, a new visual part representing the constraint that a point is (i) on a line and (ii) midway between the two end points can be created in THINGLAB [3] by placing these two existing visual constraints into the same "construction view" area. Once there, the net constraints are the union of the individual constraints. This new constraint can be used subsequently in the same manner as the constraints supplied with the start-up system. The new icon for this construct can then be added to a palette thus extending the system's functionality. Alternatively, the icon stays on the screen and only expands out into its full detail if the user clicks on it.

Above diagrammatic systems, there exist "other", more intricate visual expressions such as virtual reality (VR) systems; i.e. systems...

> ...which provides real-time viewer-centered head-tracking perspective with a large angle of view, interactive control, and binocular display [6].

In the CAVE and COSMIC WORM VR systems, users watch interactive displays of heavy fluids lying on top of a lighter fluid (the Rayleigh-Taylor Instability) and gravitational wave components predicted by Einstein's Theory of General Relativity [6, 33]. The projection is in stereo, which means that users wear high-tech 3D glasses producing 3D objects that "virtually beg to be touched" [6]. Such exotic visual expressions require non-trivial resources to produce. The CAVE is powered by five Silicon Graphics high-end work stations.

In practice, a VP system uses a combination of many of the above visual expressions. For example, MICROSOFT's ACCESS database product implements a QBE variant in which users can specify joins across tables by drawing lines between icons representing the different fields. Projects, selects are specified via a tabular interface. The data dictionary is controlled by a form-based interface while screen designs are specified by an iconic interface.

2.3 Purpose of a VP System

Figure 4 shows a rough characterisation of the *purpose* of a VP system: i.e. *specifiers* or *visualisers*. Specifiers are tools that let the user record their requirements visually. For example, an interactive E-R diagramming tool could automate the generation of database tables directly from the entered diagrams (e.g. SUPER [7]).

Specifiers may or may not be able to execute their specification within their own visual environment. Two interesting sub-categories of non-executing specifiers are *screen painters* and *design-only tools*:

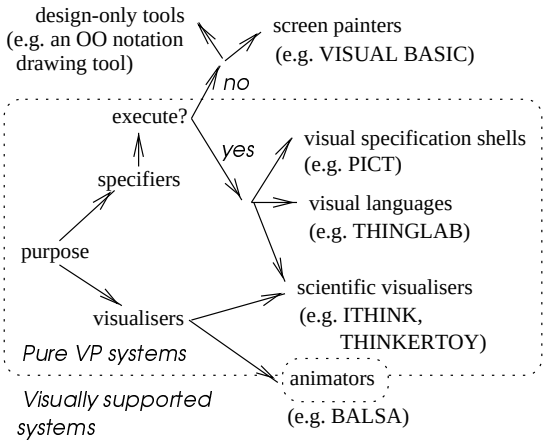

Figure 4: Purpose of a VP system

- *Screen painters* (e.g. MICROSOFT's VISUAL BASIC) give a developer an interactive point and click environment for placing window widgets (e.g. buttons, list boxes, text fields) on a screen. Typically, these systems can then automatically generate the layout code for these widgets. The developer must then use a textual language to code the semantics and interactions of the widgets
- *Design-only tools* (e.g. an OO notation drawing tool) give a designer an interactive point and click environment for placing design notation icons on a screen. Depending on the tool, relationships between the icons can be specified via specialised edge types. These tools may or may not support automatic code generation. These tools are design-only in the sense that the developer cannot watch the design execute within the environment of the design tool. Note that SUPER is *not* a design-only tool since developers can watch database queries executing within the SUPER environment.

Screen painters and design-only tools do not satisfy *RULE #1* of a pure VP systems since they do not execute. However they are widely used in industry and so, pragmatically speaking, they represent a important category of VP systems. Hence, we call non-executing specifiers visually supported systems.

Executing specifiers can be divided into *visual specification shells* and *visual languages*. A visual shell hides a conventional syntactic language beneath an visual specification environment. The

shell executes by translating its diagrams down into the underlying language (for example, PICT converts its diagrams into a subset of Pascal [12]). A visual language allows the user to specify new language primitives. For example, recall the new constraint added in the above THINGLAB example. Specifiers may include a visual trace facility. For example, whenever control moves to a part of a program, its associated icon may highlight.

Visualisers can be divided into *scientific visualisers* and *animators*. Animators try to represent in a comprehensible way the inner-workings of a program. In the BALSA animator system [4], students can (e.g.) contrast the various sorting algorithms by watching them in action. Note that animation is more than just tracing the execution of a program. Animators aim to *explain* the inner workings of a program. Extra explanatory constructs may be needed on top of the programming primitives of that system. For example, when BALSA animates different sorting routines, special visualisations are offered for arrays of numbers and the relative sizes of adjacent entries.

Animators may or may not be pure VP systems. BALSA does not allow the user to modify the specification of the animation. To do so requires extensive textual authoring by the developer. BALSA therefore does not satisfy *Rule #2* of pure VP system a visually supported system. However, there is no theoretical reason why future animation system could not permit visual specification of the animations. Hence we draw animators in Figure 4 on the border of pure VP and visually supported systems.

Scientific visualisers (e.g. THINKERTOY [15] and rTHINK [16]) are tools for support simulations. Arbitrary networks of computational devices can be drawn and executed. Tools are provided for reporting visually the output of the executions. A particular focus of current scientific visualisers is the display of changes to continuous variables over time. Scientific visualisation requires the presentation of large amounts of data. THINKERTOY provides the user with numerous visual tools that support (e.g.) the graphical displays of time-varying values; crystal growth across some 3-D terrain; and the manipulation of data values by user-specifiable filters. Since the user can modify the specification of the simulation within the visualiser's environment, scientific visualisers are also be executable specifiers.

2.4 Designing a VP System

When building a VP system, designers have to specify a *semantic base*, a *syntactic base* and a set of *basic constructs* which can be used in the start up system. For a description of the *syntactic base*, see the above discussion on visual expressions (Section 2.2). Many systems permit the extension of the basic constructs (e.g.) in the manner described above for THINGLAB (see Section 2.2). Note that if a base construct does not include some sort of conditional branching, then the VP system can only ever piece together building blocks defined outside the VP system.

Semantics bases include *control-flow*, *functional*, *data-flow*, *constraint-based*, *logic-based* and *procedural-based* (see Figure 5). Procedural-based systems convert their diagrams into some underlying language (e.g. recall that PICT is converted into PASCAL). While this has some advantages (e.g. simple execution), it implies that the idiosyncrasies of the language have to be handled at the visual level. The other semantic bases listed above strive for a simple uniform view of the program structures. Such uniformity simplifies the interface construction, decreases the amount a user has to learn, and promotes a uniform mental model for the user of the VP system.

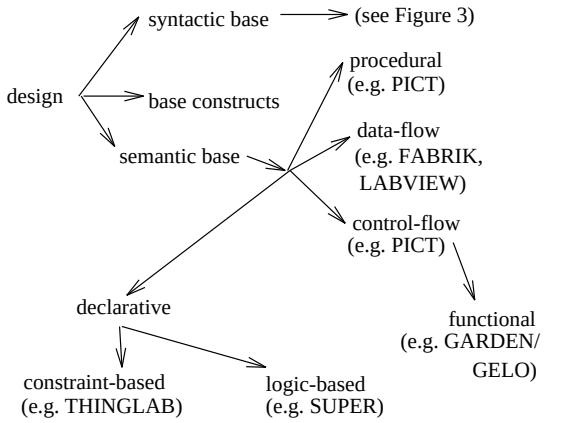

Figure 5: Design choices within a VP system

In a control-flow VP system, users manipulate iterator expressions (e.g. `while-do`, `repeat-until`), conditional expressions (e.g. `if-then`, `case`) and sequence expressions (e.g. `do this, then this, then that`) to explicitly specify the control of the system. Control-flow systems model traditional flow chart systems. Interestingly, PICT is both a control-flow system and a procedural system.

An interesting variant of control-flow sys-

tems are functional VP systems (e.g. GARDEN [30]/ GELO [31]). In these systems, users manipulate expressions that can recursively contain expressions. All the expressions respond to the same high-level protocol. For example, sending the message `execute` to an `if` expression will result in the conditional part of that `if` being sent the message `execute`. If this returns true, then the `if` expression will send the message `execute` to its action part, which will contain some sequence expressions. These will all be `execute`d in turn. A simple trace facility for such functional systems can be implemented as follows. Whenever `execute` is sent to an expression, its visual representation on screen highlights. As this highlight moves over the screen, users can watch the control flow. While simple to implement, this approach has certain intrinsic limitations. Only in a purely functional system can the semantics of an expression be contained in itself and its contained expressions. Certain global processing (e.g. joins across two relational tables) cannot be easily visualised by such a local propagation trace algorithm.

In a data-flow VP system (e.g. LABVIEW [39], FABRIK [24], and the systems reviewed in [17]), control is implicit. Each expression manipulated by the user describes:

- A set of data sources;
- Possibly, a set of conditionals;
- And action(s) to perform when:
    1. All the data sources are available and
    2. The conditions (if any) that use data from those sources are satisfied.

If the actions are performed then the expression is said to have *fired*. After firing, an expression may become an available data source for some downstream expression. At runtime, the pattern of firings ripples out across a network of connected expressions. A simple example of a data-flow system is a *Petri net*. In a basic Petri net comprising directed *arcs* and *places* (i.e. edges and vertices respectively), a set of *tokens* move out over the net. A place is fired if out all of its incoming places and none of its outgoing places have tokens. On firing, tokens are removed from each incoming place and one token is deposited in each outgoing place.

Hils argues [17] that a data-flow system is a good design choice for the purpose of scientific visualisation. Given a library of filters that can modify data, it is a simple and intuitive process for users to add filters to data-flow edges in order to transform data. Monitors for data values can be added in the same simple manner.

We view the data-flow model as a generalisation of the event-driven programming model. Each event handler is like a data-flow node that waits for certain data (events) to arrive. We note that production systems can also be viewed as data-flow systems. Production rules conditions act as demons that await the arrival of certain data elements before executing their conclusion. The connection between data-flow/event-flow and production rule systems should be stressed to students. Once this is clear in their minds, then it is a simple matter to introduce the visual programming cognitive psychology research that is based on the production rule metaphor (see the Larkin, Simon, and Goel work discussed in Section 3.1) [1].

Note that a data-flow system could be used to partially emulate a control-flow system. Consider a hypothetical `if-then-else` expression visual element in a data-flow system. This expression could have two outputs: one for the `then` part and one for the `else` part. The conditionals of this expression could model the `if` part. If the `if` is satisfied, then the `then` output could be enabled causing the actions associated with the `then` to be fired. Otherwise, the `else` output could be enabled. We call this a partial implementation of control-flow diagrams for two reasons:

1. Using a data-flow system to emulate a control-flow system may have significant computational overheads. Data-flow systems may include a complicated controlling algorithm that searches for expressions to fire. Control-flow systems do not need such complicated control: the required control is built into the network of control expressions supplied by the user.
2. Recursion is a common construct in control-flow systems. Implementing recursion is more that just visually attaching some output arc back into an input arc. On each recursive entry to some function, new copies of that function's variables have to be loaded into a separate name space lest the processing at recursive level N overwrites variables set at some recursive level less than N. If such renaming is not supported in the data-flow system, then recursion cannot be implemented.

In a constraint-based VP system (e.g. THINGLAB), the user visually speci-

[1] For an introductory tutorial on the production rule metaphor, see [23]

fies the invariants for each expression. At runtime, a constraint-solver permits manipulations that do not violate the invariants. Declarative constraints can be used to test user-proposed actions or to propose valid-actions. Any user-proposed action that violates the invariants is blocked. Given the current state of the system, a constraint-based system can generate menus of valid actions by generating all variable bindings that would not violate the invariants, given the current state.

Constraint-based systems are a variant on logic-based systems (e.g. SUPER). Such logic-based systems represent their expressions in a uniform recursive manner. Expressions can contain logical variables which can be bound at runtime and only unbound after backtracking on failure. At runtime, a general theorem prover is used to seek a set of bindings that are consistent with the theory. Visual logic-based systems can be traced by updating the display of expressions whenever a variable is bound/unbound. Unlike tracing for functional systems, this logic-based tracing can visualised global variables. For example, all the rows in a database table are global. As the theorem prover searches over the table, the attributes that satisfy the expressions are fetched and displayed.

2.5 Using this Framework

One important exercise performed by the students was the application of the above framework to commercial VP systems. The case study was IBM's Smalltalk VisualAge system and is described in the appendix.

2.6 A Second Look at the Shu Triangle

Using the theoretical framework, we can clarify some aspects of the Shu triangle.

- *Visual extent*: Clearly, the hierarchy of visual expressions offered in Figure 3 mark different levels in Shu's visual extent axis.
- *Scope*: Shu argues that the scope of QBE is more general than PICT but comments that QBE is limited to simple flat tables. It was hard to defend this claim to the SFT5030 students. In the end, it came down to my own contentious belief that declarative programming is ultimately more powerful than procedural programming. A similar comparisons was also problematic. In class, we tried to compare a screen-painter system (VisualAge) with a scientific visualiser (iTHINK). The attempt failed when we realised that the goals of the systems were entirely different. In no sense can you do screen design in iTHINK. Nor can you watch quantitative models execute in VisualAge. The general lesson from this problematic scope comparisons is that it is unwise to compare the scope of systems with a different semantic base.

Also, after discussions with SFT5300 students, we propose another clarification of the Shu criteria. When measuring the language level, training effects should be ignored. That is, language level should be measured using subjects who have already been trained in the system. Otherwise, the "language level" criteria would confuse ease of use with ease of learning.

3 An Evaluation Framework for VP

In the previous section we described a framework of current directions in VP. In this section, we examine the systems built in that framework and assess their utility.

Visual programming (VP) is an seen by many as an exciting alternative to traditional text-based computing. For example:

> When we use visual expressions as a means of communication, there is no need to learn computer-specific concepts beforehand, resulting in a friendly computing environment which enables immediate access to computers even for computer non-specialists who pursue application domains of their own. [18]

Green *et. al.* [14] and Moher *et. al.* [25] summarise claims such this as the *superlativist* position; i.e. graphical representations are inherently superior to textual representations. Both the Green and Moher groups argue that this claim is not supported by the available experimental evidence. Further, they argue against claims that visual expressions offer a higher *information accessibility*; for example:

> Pictures are superior to texts in a sense that they are abstract, instantly comprehensible, and universal. [18]

We will return to the analysis of the Green and Moher groups below. Section 3.1 argues for the

utility of VP. Section 3.2 reviews the available experimental evidence to argue that the utility of VP over other representations has not been proved.

Section 3.2 will be somewhat negative about the VP paradigm. However, recall that VP is a report of an emerging field exploring a new direction. It is to be expected that evaluation criteria are only just being evolved. The VP approach should be actively explored. When we change modalities to a 2-d screen, we find our technologies and terminology challenged and pushed to their limits. For example, the core problem of constraint-based VP systems is the incremental evaluation of constraints [8]. Also, when we created a controller for our data-flow system, we have to address some basic issues about distributed control. We should study VP because it extends and tests our concept of a computer. However, Section 3.2 does suggest that we should be more precise about our claims for VP.

3.1 Evidence For VP

Our own experience with students using visual systems is that the visual environment is very motivating to students. Others have had the same experience:

> The authors report on the first in a series of experiments designed to test the effectiveness of visual programming for instruction in subject-matter concepts. Their general approach is to have the students construct models using icons and then execute these models. In this case, they used a series of visual labs for computer architecture. The test subjects were undergraduate computer science majors. The experimental group performed the visual labs; the control group did not. The experimental group showed a positive increase in attitude toward instructional labs and a positive correlation between attitude towards labs and test performance [40].

For another example of first year students being motivated by a VP language, see [12, p18-19]. However, merely motivating the students is only half the task of an educator. Apart from motivating the students, educators also need to train students in the general concepts that can be applied in different circumstances. The crucial case for evaluating VP systems is that VP systems improve or simplify the task of comprehending some conceptual aspect of a program. If we extend the concept of VP systems to diagrammatic reasoning in general, then we can make a case that VP has some such benefits. Larkin & Simon [23] distinguish between:

- *Sentential representations* whose contents are stored in a fixed sequence; e.g. propositions in a text.
- *Diagrammatic representations* whose contents are indexed by their position on a 2-D plane.

While these two representations may contain the same information, their computational efficiency may be different. Larkin & Simon present a range of problems modeled in a diagrammatic and sentential representation using production rules. Several effects were noted:

- *Perceptual ease*: Certain features are more easily extracted from diagrams than from sentential representations. For example, adjacent triangles are easy to find visually, but require a potentially elaborate search through a sentential representation.
- *Locality aids search*: Diagrams can group together related concepts. Diagrammatic inference can use the information in the near area of the current focus to solve current problems. Sentential representations may store related items in separate areas, thus requiring extensive search to link concepts.

In a similar study, Koedinger [22] argued that diagrams optimise reasoning since they can model model whole-part relations. Both the Larkin & Simon and the Koedinger study argue for the computational superiority of diagrams for representing problems that have a two-dimensional component.

Other authors such as Goel and Kindfield argue that diagrams are useful for more than just two-dimensional reasoning. Goel [13] studies the use of *ill-structured* diagrams at various phases of the process of design. In a *well-structured* diagram (e.g. a picture of a chess board), each visual element clearly denotes one thing of one class only. In a ill-structured diagram (e.g. an impressionistic charcoal sketch), the denotation and type of each visual element is ambiguous. In the Goel study, subjects explored (i) preliminary design, (ii) design refinement, and (iii) design detailing using a well-structured diagramming tool (MacDraw) and a ill-structured diagramming tool (free-hand sketches using pencil and paper). Free-hand sketches would generate many variants. However, the well-structured tool seemed to inhibit new

ideas rather than help organise them. Once something was recorded in MacDraw, that was the end of the evolution of that idea.

> One gets the feeling that all the work is being done internally and recorded after the fact, presumably because the external symbol system (MacDraw) cannot support such operations [13].

Goel found that ill-structured tools generated more design variants (i.e. more drawings, more ideas, more use of old ideas) than well-structured tools. We make two conclusions from Goel's work. Firstly, at least for the preliminary design, ill-structured tools are better. Secondly, after the brain-storming process is over, well-structured tools can be used to finalise the design.

Kindfield [21] studied how diagram used changes with expertise level. Expert geneticists, experienced genetics students, and introductory genetics students explored genetics problems using diagrams of chromosones. Kindfield found that novices used literal diagrams (i.e. exact replicas of what could viewed down a microscopy) while experts used diagrams that showed only the subset of the features in the literal diagrams. Interestingly, the subset of the features found in the expert's diagrams changed according to the task at hand. Kindfield argues that:

> Diagrams... serve as an external storage device that frees working memory, allowing for the performance of additional cognitive tasks during the pause when the problem solver is looking or touching the diagram [21].

That is, according to Kindfield, diagrams are like a temporary swap space which we can use to store concepts that (i) don't fit into our head right now and (ii) can be swapped in rapidly; i.e. with a single glance.

3.2 Evidence Against VP

The previous section discussed theoretical issues or small experiments that argue for the utility of visual/diagrammatic representations over textual/sentential representations. This section reviews larger-scale studies which suggest that, despite the arguments made in the last section, the potential benefits of VP has yet to be conclusively demonstrated.

It is not clear that any of the advantages of diagrammatic reasoning offered by Larkin & Simon, Goel and Kindfield apply to general software engineering. Many software engineering problems are not naturally two-dimensional. For example, while we write down an E-R diagram on the plane of a piece of paper, the inferences we can draw from that diagram are not dependent on the physical position of (e.g.) an entity.

In terms of the ill-structured/well-structured division, the VP tools we have seen are a well-structured tool. That is, they are less suited to brain-storming than producing the final product.

Nor can we view most available VP systems as Kindfield-style temporary swap spaces for formative ideas. Kindfield's results suggest that such swap spaces need to be filterable. We should be able to hide away the details that are not relevant to some current problem. A VP environment that supported such filtering would need to (i) model the current task, (ii) store a library of easily-applied filters, and (perhaps) have (iii) some knowledge of the user's level of expertise. None of the systems we have reviewed here meet this criteria.

Jarvenpaa & Dickson (hereafter, JD) report an interesting pattern in the VP literature [20]. In their literature review on the use of graphics for supporting decision making, they find that most of the proponents of graphics have never tested their claims. Further, when those tests are performed, the results are contradictory and inconclusive. For example:

- JD cite 11 publications arguing for the superiority of graphics over tables for the purposes of elementary data operations (e.g. showing deviations, summarising data). None of these publications tested their claims. Such tests were performed by 13 other publications which concluded that graphics were better than tables (37.5%), the same as tables (25%), or worse than tables (37.5%)
- JD cite 11 publications arguing for the superiority of graphics over tables for the purposes of decision making (e.g. forecasting, planning, problem finding). None of these publications tested their claims. Such tests were performed by 14 other papers which concluded that graphs were better than tables (27%), the same as tables (46%), or worse than tables (27%).

Similar contradictory results can be found in the study of control-flow and data-flow systems.

- The utility of flowcharts for improving program comprehension, debugging, and extensi-

bility was studied by Shneiderman [35]. Shneiderman found no difference in the performance of the subjects using/not using control-flow diagrams.

- On the other hand, recent results have been more positive [34].
- Studies have reported that Petri nets are comparatively worse as specification languages when compared to pseudo-code [2] or E-R diagrams [37].
- On the other hand, another study suggests that Petri nets are better than E-R diagrams for the maintenance of large expert systems [38].

Given these conflicting results, all we can conclude at this time is that the utility of control-flow or data-flow visual expressions are an open issue.

Perhaps rather than seeking a the "best" representation, we should acknowledge that any particular representation is useful only for certain tasks. Recall our conclusion from Goel's work: different diagramming techniques are useful for different stages of the design process. Other researchers explore the different circumstances under which certain types of visual expressions are useful: Gerstendorfer & Rohr [9] (hereafter, GR), Moher *et. al.* [25] (hereafter, the Moher group) and Green *et. al.* [14] (hereafter, the Green group). The GR work distinguishes between *visual*, *verbal*, and *formal* tasks:

- A GR visual task is inherently structural; e.g. laying out a table so that related utensils are near each other. GR visual tasks are best specified and executed using diagrams.
- A GR verbal task is inherently sequential; e.g. cooking a meal where the steps have a time dependency on each other. GR verbal tasks are best specified and executed using written text.
- A GR formal task requires classification and abstraction; e.g. to decide the cost of an order, the order items have to be grouped into abstract categories since each group has its own price. GR formal tasks are best specified and executed using tables.

In the GR study, subjects were trained and then had to solve a layout, cooking, or costing problem. For different students and for each problem, a visual/verbal/ or tabular presentation was used for the training and execution of that problem. For both training and execution, using written text proved to the slowest option. Visual presentations lead to better training and execution times for GR visual and GR verbal problems. However, tabular presentations were best for training and executing GR formal tasks.

The Green group explored two issues: superlativism and information accessibility (defined above at the start of Section 3). Subjects attempted some comprehension task using both visual expressions and textual expressions of a language. The Green group rejected the superlativism hypothesis when they found that tasks took longer using the graphical expressions than the textual expressions. The Green group also rejected the information accessibility hypothesis when they found that novices had more trouble reading the information in their visual expressions than experts. That is, the information in a diagram not " instantly comprehensible and universal". Rather, such information can only be accessed after a training process.

The Moher group performed a similar study to the Green group. In part, the Moher study used the same stimulus programs and question text as the Green group. Whereas the Green group used the LABVIEW data-flow system, the Moher group used Petri nets. The results of the Moher group echoed the results of the Green group. Subjects were shown three variants on a basic Petri net formalism. In no instance did these graphical languages outperform their textual counterparts.

The Moher group caution against making an alternative superlativism claim for text; i.e. text is better than graphics. Both the Moher and Green groups distinguished between *sequential* programming expressions such as a decision true and *circumstantial* programming expressions such as a backward-chaining production rule. Both sequential and circumstantial programs can be expressed textual and graphically. The Moher group comments that:

> Not only is no single representation best for all kinds of programs, no single representation is ... best for all tasks involving the *same* program [25].

Sequential programs are useful for reasoning forwards to perform tasks such as prediction. Circumstantial programs are output-indexed; i.e. the thing you want to achieve is accessible separately to the method of achieving it. Hence, they are best used for hypothesis-driven tasks such as debugging.

3.3 Discussion

Theoretical studies and small scale experimental studies suggest an inherent utility in visual expressions. However, when we explore the available experimental evidence, we find numerous contradictory results. For example, note that the Green group's conclusion that text expressions were always faster than visual expressions contradicts the results of the GR study. We cannot resolve this discrepancy here except to speculate that perhaps the crucial factor determining the value of a representation is not the surface features of a representation (e.g. its appearance). Rather, its relevance to the task at hand seems more important.

However, one issue that is clear from this section is that evaluation is an open-issue in VP research. When we build VP systems, we should include in them logging software that supports the evaluation of these systems.

4 Discussion

We have described the theoretical framework and evaluation frameworks for SFT5030: a one semester post-graduate introductory subject on VP. Students find the subject describe here stimulating, but demanding. If we had space in our curriculum, we would divide SFT5030 into two:

- The VP-1 subject would teach graphical user interfaces and visually supported systems using screen painters such Visual Basic. The theoretical component of this subject would be cognitive issues of interface design (perhaps based on [27]) and some introductory cognitive psychology. In tutorials, students could explore the programming details of their screen painter. For assignment work, students could use the graphical widgets in their screen painter to build a front-end to some application. The front-end must have some decision-support functionality; i.e. it would have to encourage the detection of some business problem.

- The VP-2 subject would teach pure VP systems via a review of the state-of-the-art in the VP literature. VP-2 would present the theoretical and evaluation framework described above. In their tutorials, students could practice evaluating real VP systems using commonly available systems such as iTHINK and LABVIEW. For assignment work, students could design some experiment to test the utility of the program they wrote for VP-1 system. Also, students could present seminar papers assessing VP systems from the literature using our theoretical framework.

5 Acknowledgments

Don Watson did all the VisualAge coding and assisted in the VisualAge evaluation. Heinz Schmidt offered helpful comments on an earlier draft of this paper. The students of SFT5030 cheerfully read different versions of this paper and corrected many aspects. The comments of two students, Glenn Roslin and Tom Leden, were particularly insightful.

6 Appendix: Evaluating a Commercial VP System

This section contains a sample evaluation of a commercial VP system.

6.1 General Notes

IBM's Smalltalk VisualAge is a two-layered system with numerous support tools:

- On top is the *composition editor* (hereafter, $\mathcal{CE}$). This editor allows the visual specification of forms-style screens. In this editor, a developer can draw lines between screen components representing calls from (e.g.) a clicked button to adding an item to a list.

- Underneath $\mathcal{CE}$, is Smalltalk, an industrial-strength object-oriented programming language supporting an incremental compiler, a well-developed class library, and automatic garbage collection.

- VisualAge also includes numerous tools for database connection, interfacing to C or COBOL programs, and networking.

$\mathcal{CE}$ is a system for specifying event-based interfaces. The interfaces specified by $\mathcal{CE}$ are primarily form-based, but may include tables. Visual components can be specified, then partially-encapsulated inside a `ViewWrapper`. In this way, complicated specifications can be expressed in parts. A single icon on a screen can expand, if requested, into some intricate network of visual elements. This net can then be contracted again to a single icon to improve the readability of this screen.

One advantage of $\mathcal{CE}$ over raw Smalltalk code is that the relative positions of the screen items can be specified visually. Such relative positions are intricate to specify in raw Smalltalk. Impressive forms-based screens containing (e.g.) drop-down multiple-choice lists, buttons, text and graphics panes can be created. However, the range of visual items is limited. For example, it is not possible to create a screen containing the $\mathcal{CE}$ interface using $\mathcal{CE}$. Such an editor would require the ability to build a runtime screen where the user can drop icons, then connect them with rubber-banding lines. It is not possible to build such a screen in $\mathcal{CE}$.

6.2 Specific Notes

In terms of our theoretical framework, $\mathcal{CE}$ is a:

- *Semi-direct manipulation* system: $\mathcal{CE}$ permits some direct manipulation experimentation but the undo system is somewhat clumsy which discourages experiments. Further, we found that it is not a smooth transition from initial experience to more expert use.

- *Icon-based diagramming tool.* Developers pick icons from a palette and add them to an arbitrary network of connections.

- *Data-flow procedural* system. $\mathcal{CE}$ is an event-driven system. We discussed above the close connection between event driven-systems and data-driven systems. $\mathcal{CE}$ is a procedural system since its visual representations are executed via the generation of Smalltalk code.

- *Visually supported* system. The $\mathcal{CE}$ documentation is extensive but in all that documentation, there are only half a dozen purely visual examples (i.e. use no scripting). We know several professional VisualAge developers who all agree that to use VisualAge, developers must generate scripts. This scripting may be non-trivial since it requires an understanding of the underlying Smalltalk language. Since the customisation of $\mathcal{CE}$ requires non-visual coding, $\mathcal{CE}$ is not a pure VP system.

Figure 6 uses a Shu triangle to compare $\mathcal{CE}$ with LABVIEW and PICT. We scored LABVIEW higher on all axes than PICT since LABVIEW has been under active development for longer.

As to the comparison between $\mathcal{CE}$ and these other two systems:

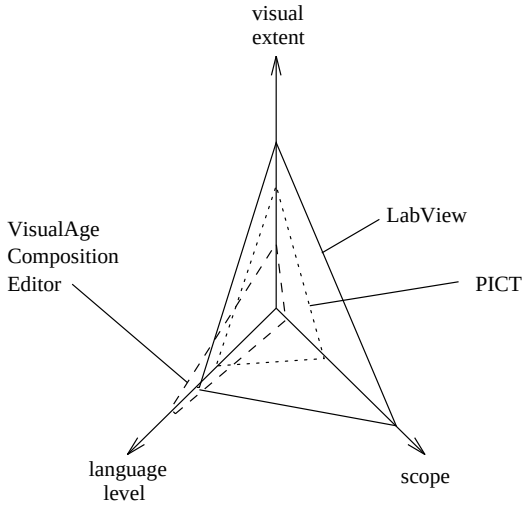

Figure 6: Shu triangle for $\mathcal{CE}$

- *Visual extent*: $\mathcal{CE}$, LABVIEW and PICT are all icon/diagrammatic systems. $\mathcal{CE}$ lacks a visual concept of execution trace. Therefore we score it lowest on the visual extent axis.

- *Language level*: LABVIEW and PICT have simpler, more uniform interfaces than $\mathcal{CE}$. It could be argued that this uniformity simplifies application development and that, hence, $\mathcal{CE}$ has the lowest language level. However, after training, it is possible to built more intricate applications in an hour in $\mathcal{CE}$ than LABVIEW and PICT. For example, $\mathcal{CE}$ comes with (e.g.) mainframe database hooks so it is easy to build a client-server application. Hence, we say that $\mathcal{CE}$ has an arguably higher language level than the other systems but acknowledge that this rating is debatable.

- *Scope*: Shu rated PICT very low on the scope axis and we rate $\mathcal{CE}$ even lower. $\mathcal{CE}$ cannot be customised without leaving its visual environment. Hence, as a VP, we argue that $\mathcal{CE}$ has a low scope (Shu gave XEROX STAR a low scope rating for the same reason).

When working with $\mathcal{CE}$, we often ran up against the *control-flow/ data-flow* distinction. Control-flow diagrams are a natural modeling tool for procedural code. However, it is difficult to use VisualAge for general control-flow programming. We considered extending $\mathcal{CE}$ with extra control icons for `if-then`, `while`, etc. However, we rejected this approach after considering the expansion factor, and the recursion problem:

The expansion factor: Visual programming is a failure if the visual expression is more complex

than some alternative representation. With two drag-and-drops, and a few additional mouse clicks, $\mathcal{CE}$ lets a programmer specify in seconds the position of a button on a screen and the action that happens when it is pressed. The same process in Smalltalk would take dozens of lines of typing to specify. In this example, $\mathcal{CE}$ is a success since it simplifies rather than complicates the specification process. However $\mathcal{CE}$ is not always so successful. We tried re-implementing some old first-year assignments in VisualAge. One assignment asked students to display the available spare seats in an airplane as a tool for supporting airline reservations. We considered visually specifying an iterator that ran over the seats looking for an empty one. However, the equivalent textual code was so brief, that we could not justify the effort. $\mathcal{CE}$ was a failure in this case since the textual representation was far less verbose. More generally, we argue that for tasks that actually involve detailed algorithmic control, a data-flow environment like $\mathcal{CE}$ is inappropriate.

The recursion problem: We have argued above that implementing recursion in a data-flow environment that does not support recursion is tricky (see Section 2.4). $\mathcal{CE}$ supports an `ObjectFactory` icon which can generate many instances of a class. We could model a recursive function as a class that can generate multiple instances of itself: one for each recursive call. The generation process could be organised by the `ObjectFactory`. Encapsulated instance variables would separate the name space of variables used at different recursive levels. However, this seems a rather arcane use of objects. Also it is not clear how to modify the $\mathcal{CE}$ such that the visual representation of recursion is intuitively displayed. Further, the VisualAge experts we spoke to about this proposal all felt that a non-trivial degree of scripting was required to implement this approach. Some were skeptical that it was even a practical suggestion.

References

[1] A.L. Ambler. Forms: Expanding the Visualness of Sheet Languages. In *Proceedings Workshop on Visual Languages, Tryck-Center, Linkoping, Sweden*, pages 105–117, 1987.

[2] D.A. Boehm-Davis and A.M. Fregly. Documentation of Concurrent Programs. *Human Factors*, 27:423–432, 1985.

[3] A.H. Borning. Graphically Defining New Building Blocks in ThingLab. *Human-Computer Interaction*, 2(4):269–295, 1986.

[4] M.B. Brown and R. Sedgewick. Techniques for Algorithm Animation. *IEEE Software*, pages 28–39, January 1985.

[5] T.B. Brown and T.D. Kimura. Completeness of a Visual Computation Model. *Software- Concepts and Tools*, pages 34–48, 1994.

[6] C. Cruz-Neira, D. Sandin, and T. DeFanti. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. In *Proceedings of SIGGRAPH '93, ACM SIGGRAPH*, pages 135–142, August 1993.

[7] Y. Dennebouy, M. Andersson, A. Auddino, Y. Dupont, E. Fontana, M. Gentile, and S. Spaccapietra. Super: Visual Interfaces for Object and Relationship Data Models. *Journal of Visual Languages and Computing*, pages 73–99, 1995.

[8] B.N. Freeman-Benson, J. Maloney, and A. Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33:54–63, 1 1990.

[9] M. Gerstendorfer and G. Rohr. Which Task in Which Representation on What Kind of Interface. In *Human-Computer Interaction - INTERACT '87*, page 6 pages, 1987.

[10] E.P. Glinert, editor. *Visual Programming Environments: Applications and Issues*. IEEE Computer Society Press, 1990.

[11] E.P. Glinert, editor. *Visual Programming Environments: Paradigms and Systems*. IEEE Computer Society Press, 1990.

[12] E.P. Glinert and S.T. Tanimoto. Pict: An Interactive Graphical Programming Environment. *IEEE Computer*, pages 7–25, November 1984.

[13] V. Goel. "Ill-Structured Diagrams" for Ill-Structured Problems. In *Proceedings of the AAAI Symposium on Diagrammatic Reasoning Stanford University, March 25-27*, pages 66–71, 1992.

[14] T.R.G. Green, M. Petre, and R.K.E. Bellamy. Comprehensibility of Visual and Textual Programs: The Test of Superlativism Against the "Match-Mismatch" Conjecture. In *Empirical Studies of Programmers: Fourth Workshop*, pages 121–146, 1991.

[15] S.H. Gutfreund. ManiplIcons in ThinkerToy. In E.P. Glinert, editor, *Visual Programming Environments: Applications and Issues*, pages 25–45. IEEE Computer Society Press Tutorial, 1990.

[16] Inc. High Performance Software. iThink 3.0.5, 1994.

[17] D.D. Hils. Visual Languages and Computing Survey. *Journal of Visual Languages and Computing*, 3(1):69–101, 1992.

[18] M. Hirakawa and T. Ichikawa. Visual Language Studies - A Perspective. *Software- Concepts and Tools*, pages 61–67, 1994.

[19] T. Ichikawa and S.K. Chang. Special Issue on Visual Programming. *IEEE Transactions on Software Engineering*, 16(10):1105–1197, 1990.

[20] S.L. Jarvenpaa and G.W. Dickson. Graphics and Managerial Decision Making: Research Based Guidelines. *Communications of the ACM*, 31(6):764–774, June 1988.

[21] A.C.H. Kindfield. Expert Diagrammatic Reasoning in Biology. In *Proceedings of the AAAI Symposium on Diagrammatic Reasoning Stanford University, March 25-27*, pages 41–46, 1992.

[22] K.R. Koedinger. Emergent Properties and Structural Constraints: Advantages of Diagrammatic Representations for Reasoning and Learning. In *Proceedings of the AAAI Symposium on Diagrammatic Reasoning Stanford University, March 25-27*, pages 154–159, 1992.

[23] J.H. Larkin and H.A. Simon. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science*, pages 65–99, 1987.

[24] F. Ludolph, Y. Chow, D. Ingalls, S. Wallace, and K. Doyle. The Fabrik Programming Environment. In *IEEE Proceedings Workshop on Visual Languages*, pages 222–230, 1988.

[25] T.G. Moher, D.C. Mak, B. Blumenthal, and L.M. Leventhal. Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets. In *Empirical Studies of Programmers: Fifth Workshop*, pages 137–161, 1993.

[26] B.A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.

[27] D.A. Norman. *The Design of Everyday Things*. DoubleDay Currency, 1989.

[28] M.C. Pong and N. Ng. PIGS - A System for Programming with Interactive Graphical Support. *Software Practice Experience*, 13(9):847–855, 1983.

[29] R. Purvey, J. Farrell, and P. Klose. The Design of Star's Records Processing: Data processing for the noncomputer professional. *ACM Tans Office Inf. Syst.*, 1(1):3–34, 1983.

[30] S. P. Reiss. Working in the Garden Environment for Conceptual Programming. *IEEE Software*, pages 16–27, November 1987.

[31] S.P. Reiss, S. Meyers, and C. Duby. Using GELO to Visualize Software Systems. In *Proceedings of the Second Annual Symposium on User Interface Software and Technology*, pages 149–157, November 1989.

[32] L.J. Rosenblum and B.E. Brown. Special Issue on Visualization. *IEEE Computer Graphics and Applications*, 12(4):18–71, July 1992.

[33] M. Roy, C. Cruz-Neira, and T. De Fanti. *Networks and Virtual Environments of Presence Teleoperators and Virtual Envionrment*, chapter Cosmic Worm in the Cave: Steering a High Performance Computing Application From a Virtual Environment. MIT Press. To appear.

[34] D.A. Scanlan. Structured Flowcharts Outperform Psuedocode: an Experimental Comparison. *IEEE Computer*, 6(5):28–36, 1989.

[35] B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, pages 57–69, August 1983.

[36] N.C. Shu. Visual Programming Languages: A Perspective and a Dimensional Analysis. In S.K. Chang and P.A. Ligomenides, editors, *Visual Languages*, pages 11–34, 1986.

[37] K.M. Swigger and R.P. Brazile. Experimental Comparisons of Design/Documentation Formats for Expert Systems. *International Journal of Man-Machine Studies*, 31:47–60, 1989.

[38] K.M. Swigger and R.P. Brazile. An Empirical Study of the Effects of Design/Documentation Formats on Expert System Modifiability. In J. Koenemann-Belliveau, T. Moher, and S. Robertson, editors, *Empirical Studies of Programmers: Fourth Workshop*. Ablex Publishing Corp, 1991.

[39] G.M. Vose and G. Williams. LabVIEW: Laboratory Virtual Instrument Engineering Workbench. *Byte*, 11:84–92, 1986.

[40] M.G. Williams, W.A. Ledder, J.N. Buehler, and J.T. Canning. An Empirical Study of Visual Labs. In *Proceedings 1993 IEEE Symposium on Visual Languages*, pages 371–373. IEEE Comput. Soc. Press., 1993.

[41] M.M. Zloof. QBE/OBE: A Language for Office and Business Automation. *Computer*, pages 13–22, May 1981.