Visual Programming, Programming by Example, and Program Visualization: A Taxonomy.

Brad A. Myers
Dynamic Graphics Project
Computer Systems Research Institute
University of Toronto
Toronto, Ontario, M5S 1A4
Canada

ABSTRACT

There has been a great interest recently in systems that use graphics to aid in the programming, debugging, and understanding of computer programs. The terms "Visual Programming" and "Program Visualization" have been applied to these systems. Also, there has been a renewed interest in using examples to help alleviate the complexity of programming. This technique is called "Programming by Example." This paper attempts to provide more meaning to these terms by giving precise definitions, and then uses these definitions to classify existing systems into a taxonomy. A number of common unsolved problems with most of these systems are also listed.

CR Categories and Subject Descriptors: D.1.2 [Software Engineering]: Automatic Programming; D.2.2 [Software Engineering]: Tools and Techniques-Flowcharts; D.2.5 [Software Engineering]: Testing and Debugging-Debugging Aids; D.3.2 [Programming Languages]: Language Classifications; I.2.2 [Artificial Intelligence]: Automatic Programming-Program Synthesis. 13.6 [Computer Graphics]: Methodologies and Techniques-Languages.

Additional Key Words and Phrases: Visual Programming, Program Visualization, Programming by Example, Inferencing.

General Terms: Documentation, Languages.

1. Introduction

As the distribution of personal computers and the more powerful personal workstations grows, the majority of computer users now do not know how to program. They buy computers with packaged software and are not able to modify the software even to make small changes. In order to allow the end user to reconfigure and modify the system, the software may provide various options, but those often make the system more complex and still may not address the users' problems. "Easy-to-use" software, such as the "Direct Manipulation" systems [Shneiderman 83] actually make the user-programmer gap worse since more people will be able to use the software (since it is easy to use), but the internal program code is now much more complicated (due to the extra code to handle the user interface). Therefore, systems are moving in the direction of providing end user programming. It is well-known that conventional programming languages are difficult to learn and use [Gould 84], requiring skills that many people do not have. In an attempt to make the programming task easier, recent research has been directed towards using graphics. This has been called "Visual Programming" or "Graphical Programming". Some Visual Programming systems have successfully demonstrated that non-programmers can create fairly complex programs with little training [Halbert 84].

Another motivation for using graphics is that it tends to be a higher-level description of the desired actions (often de-emphasizing issues of syntax and providing a higher level of abstraction) and may therefore make the programming task easier even for professional programmers. This may be especially true during debugging, where graphics can be used to present much more information about the program state (such as current variables and data structures) than is possible with purely textual displays. This is one of the goals of Program Visualization. Other Program Visualization systems use graphics to help teach computer programming.

Programming-by-Example is another technology that has been investigated to make programming easier, especially for non-programmers. It involves presenting to the computer examples of the data that the program is supposed to process and using these examples during the development of the program. Many, although not all, Programming-by-Example systems have also used Visual Programming, so these two technologies are often linked.

Recently, there has been a large number of articles about systems that incorporate some or all of these features [Grafton 85][Raeder 85]. Unfortunately, the
terma have been used imprecisely, and there has not been a comprehensive taxonomy that classifies these systems. This paper attempts to fill this gap in the literature. First, the important terms are defined in a precise manner, and then these definitions are used to differentiate the various systems. Finally, a number of common unsolved problems with these systems are delineated.

There are many systems that could be included in this paper in the various categories, but no attempt has been made to be comprehensive. It is hoped that the selection of systems listed will help the reader understand the intent of the classification system.

2. Definitions.

Programming. What is meant by computer "programming" is probably well understood, but it is important to have a definition that can be used to eliminate some limited systems. In this paper, "program" is defined as "a set of statements that can be submitted as a unit to a computer and used to direct the behavior of that system" [Oxford 83]. While the ability to compute "everything" is not required, the system must include the ability to handle conditionals and iteration, at least implicitly.

Interactive vs. Batch. Any programming language system may either be "interactive" or "batch." A batch system has a large processing delay before statements can be run while they are compiled, whereas an interactive system allows statements to be executed when they are entered. This characterization is actually more of a continuum than a dichotomy since even interactive languages like LISP typically require groups of statements (such as an entire procedure) to be specified before they are executed.

Visual Programming. "Visual Programming" (VP) refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Conventional textual languages are not considered two dimensional since the compiler or interpreter processes it as a long, one-dimensional stream. Visual Programming includes conventional flow charts and graphical programming languages. It does not include systems that use conventional (linear) programming languages to define pictures. This eliminates most graphics editors, like Sketchpad [Sutherland 63].

Program Visualization. "Program Visualization" (PV) is an entirely different concept from Visual Programming. In Visual Programming, the graphics is the program itself, but in Program Visualization, the program is specified in the conventional, textual manner, and the graphics is used to illustrate some aspect of the program or its run-time execution. Unfortunately, in the past, many Program Visualization systems have been incorrectly labeled "Visual Programming" (as in [Grafton 85]). Program Visualization systems can be divided along two axes: whether they illustrate the code or the data of the program, and whether they are dynamic or static. "Dynamic" refers to systems that can show an animation of the program running, whereas "static" systems are limited to snapshots of the program at certain points. If a program created using Visual Programming is to be displayed or debugged, clearly this should be done in a graphical manner, but this would not be considered Program Visualization. Although these two terms are similar and confusing, they have been widely used in the literature, so it was felt appropriate to continue to use the common terms.

Programming by Example. The term "Programming by Example" (PBE) has been used to describe a large variety of systems. Some early systems attempted to create an entire program from a set of input-output pairs. Other systems require the user to "work through" an algorithm on a number of examples and then the system tries to infer the general program structure. This is often called "automatic programming" and has generally been an area of Artificial Intelligence research.

Recently, there have been a number of systems that require the user to specify everything about the program (there is no inference involved), but the user can work out the program on a specific example. The system executes the user's commands normally, but remembers them for later re-use. Bill Buxton coined the phrase "Programming with Examples" to more accurately describe these systems. Halbert [84] characterizes Programming with Examples as "Do What I Did" whereas inferential Programming by Example might be "Do What I Mean". The term "Programming by Example" will be used to include both inferencing systems and Programming With Example systems.

Of course, whenever code is executed in any system, test data must be entered to run it on. The distinction between normal testing and "Programming with Examples" is that in the latter the system requires or encourages the specification of the examples before programming begins, and then applies the program as it develops to the examples. This essentially requires all Programming-with-Example systems (but not Programming-by-Example systems with inferencing) to be interactive.


Visual Programming, Program Visualization, and Programming by Example are very appealing ideas for a number of reasons. The human visual system and human visual information processing is clearly optimized for multi-dimensional data. Computer programs, however, are presented in a one-dimensional textual form, not utilizing the full power of the brain. Two-dimensional displays for programs, such as flowcharts and even the indenting of block structured programs, have long been known as helpful aids in program understanding [Smith 77]. Recently, a number of Program Visualization systems [Myers 80][Baecker 81][Brown 84] have demonstrated that 2-D pictorial displays for data structures, such as those drawn by hand on blackboard, are very helpful. It seems clear that a more visual style of programming could be easier to understand and generate for humans. Smith [77] discusses at length these and other psychological motivations for using more visual displays for programs and data.

It is also well known that people are much better at dealing with specific examples than with abstract ideas. A large amount of teaching is achieved by presenting important examples and having the students do specific problems. This helps them understand the general principles. Programming by Example attempts to extend these ideas to programming. In its most ideal case, the programmer acts like the teacher and just gives examples to the computer and the computer, like an intelligent pupil, intuits the abstraction that covers all the examples.
Programming-with-example systems require programmers to specify the abstraction, but allow them to work out the program on examples as an aid to getting the program correct. This is motivated by the observation that people make fewer errors when working out a problem on an example or when directly manipulating data as when editing text or moving icons on the Macintosh [Williams 84] as compared to performing the same operation in the abstract, as in conventional programming. The programmer does not need to try to keep in mind the large and complex state of the system at each point of the computation if it is displayed for him on the screen. This has been called "programming in debugging mode" [Smith 77]. In addition, these PBE systems may allow the user to specify a program using the actual user interface of the system, which is presumably familiar [Attardi 82].


This paper presents two taxonomies. This section discusses one for systems that support programming. Section 5 discusses a one for systems that use graphics after the programming process is finished (Program Visualization systems). A meaningful taxonomy can be created by classifying programming systems into eight categories using the orthogonal criteria of

- Visual Programming or not,
- Programming by Example or not, and
- Interactive or batch.

This taxonomy is original with this paper. Of course, a single system may have features that fit into various categories and some systems may be hard to classify, so this paper attempts to characterize the systems by their most prominent features. Figure 1 shows the division with some sample systems which are discussed in the following sections.

4.1. Not VP, Not PBE, Batch and Interactive

These are the conventional textual, linear programming languages that are familiar to all programmers, such as Pascal, Fortran, and Ada for batch and LISP and API for interactive.

4.2. VP, Not PBE, Batch

One of the earliest "visual" representations for programs was the flowchart. Grail [Ellis 69] could compile programs directly from computerized flowcharts, but the contents of boxes were ordinary machine language statements. GAL (see Figure 2) is similar except that it uses Nassi-Shneiderman flowcharts [Nassi 73] and is compiled into Pascal [Albizuri-Romero 84]. Another early effort was the AMBIT/G [Christensen 68] and AMBIT/L [Christensen 71] graphical languages. They supported symbolic manipulation programming using pictures. Both the programs and data were represented diagrammatically as directed graphs, and the programming operated by pattern matching. Fairly complicated algorithms, such as garbage collection, could be described graphically as local transformations on graphs.

3It is interesting to note that AMBIT/G, even though it was developed in 1969, used many of the "modern" user interface techniques, including iconic representations, gesture recognition, dynamic menus on the screen if selection from menus, selection of icons by pointing, modal and mode-free styles of interaction, etc. [Roventer 69].

4.3. Not Programming by Example

Not VP

<table>
<thead>
<tr>
<th>Batch</th>
<th>Interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Conventional Languages: Pascal, Fortran, etc.</td>
<td>LISP, API, etc.</td>
</tr>
<tr>
<td>Grail</td>
<td>Graphical Program Editor</td>
</tr>
<tr>
<td>[Ellis 69]</td>
<td>[Sutherland 66]</td>
</tr>
<tr>
<td>AMBIT/G/L</td>
<td>PIGS</td>
</tr>
<tr>
<td>[Christensen 69,71]</td>
<td>[Poug 83]</td>
</tr>
<tr>
<td>Query by Example</td>
<td>Flat</td>
</tr>
<tr>
<td>[Zloof 77,81]</td>
<td>[Ginzert 84]</td>
</tr>
<tr>
<td>FORMAL</td>
<td>PROGRAPH</td>
</tr>
<tr>
<td>[Shu 85]</td>
<td>[Pietrzykowski 83,84]</td>
</tr>
<tr>
<td>GAL</td>
<td>State Transition UIMS</td>
</tr>
<tr>
<td>[Albizuri-Romero 84]</td>
<td>[Jacob 85]</td>
</tr>
</tbody>
</table>

Programming by Example

<table>
<thead>
<tr>
<th>Batch</th>
<th>Interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O pairs*</td>
<td>Tinkor</td>
</tr>
<tr>
<td>[Shaw 75]</td>
<td>[Lieberman 82]</td>
</tr>
</tbody>
</table>

4.6. AutoProgrammer* |

- [Bauern 70b]  |
- Pygmalion |
- [Smith 77]  |
- Graphical Thinglab  |
- [Borning 86] |
- SmallStar |
- [Halbert 81,84]  |
- Rehearsal World |
- [Gould 84] |

Classification of programming systems by whether they are visual or not, whether they have Programming by Example or not, and whether they are interactive or batch. The small numbers refer to the section in which the group is discussed. Starred systems * have inferencing, and non-starred PBE systems use Programming With Example.

Figure 1.

A Nassi-Shneiderman flowchart program from GAL [Albizuri-Romero 84].

You might think that a system called "Query by Example" would be a "Programming by Example" system, but in fact, according to this classification, it is not. Query by Example (QBE) [Zloof 77] allows users to specify queries on a relational database using two-dimensional tables (or forms), so it is classified as a Visual Programming system. The "examples" in QBE are what Zloof called variables. They are called "examples" because the
user is supposed to give them names that refer to what the system might fill into that field, but they have no more meaning than variable names in most conventional languages. The ideas in QBE have been extended to mail and other non-database areas of office automation in Office by Example (OBE) [Zloof 81]. A related forms-based database language is FORMAL [Shu 85] which explicitly represents hierarchical structures.

4.3. VP, Not PBE, Interactive

Probably the first Visual Programming system was William Sutherland's [66] which represented programs somewhat like hardware logic diagrams. Some systems for programming with flowcharts have been interactive. PIGS [Pong 83] uses Nassi-Shneiderman flowcharts, and Pict [Glinert 84] uses conventional flowcharts. Pict is differentiated by its use of color pictures (icons) rather than text inside of the flowchart boxes (see Figure 3).

PROGRAPH [Pietrzykowski 83] is another interactive VP system without PBE, but it is distinguished by supporting a functional data flow language. PROGRAPH attempts to overcome some of the problems of this type of language by using a graphical representation that is structured, as shown in Figure 4. Pietrzykowski [84] claims that this alleviates the problem of functional languages where "the conventional representation in the form of a linear script makes it almost unreadable". PROGRAPH is one of the very few truly concurrent Visual Programming systems.

![Figure 3.](image1)

Three frames from Pict [Glinert 84] showing an implementation of the factorial procedure. The original pictures were in color.

![Figure 4.](image2)

Two procedures from PROGRAPH [Pietrzykowski 84].

A number of systems for automatically generating user interfaces for programs (User Interface Management Systems) allow the designer to specify the user interface in a graphical manner. An example of this is the state transition diagram editor by Jacob [85] (see Figure 5). Most other UIMSs require that designers specify the programs using some textual representation, so they do not qualify as Visual Programming.

![Figure 5.](image3)

State diagram description of a simple desk calculator [Jacob 85].

4.4. Not VP, PDE, Batch

Some systems have attempted to infer the entire program from one or more examples of what output is produced for a particular input. One program [Shaw 75] can infer simple recursive LISP programs from a single I/O pair, such as \((A B C D) \Rightarrow (D D C C B B A A)\). This system is limited to simple list-processing programs, and it is clear that systems such as this one cannot generate all programs, or are even likely to generate the correct program [Biermann 76a].

4.5. Not VP, PBE, Interactive

Tinker [Lieberman 82] is a "pictorial" system that is not VP. The user chooses a concrete example, and the system executes LISP statements on this example as the code is typed in. Although Tinker uses windows, menus, and other graphics in its user interface, it is not a VP system since the user presents all of the code to the system in the conventional linear textual manner.

4.6. VP, PBE, Batch

Inferencing systems that attempt to cover a wider class of programs than those that can be generated from I/O pairs have required the user to choose data structures and algorithms and then run through the computation on a number of examples. The systems attempt to infer where loops and conditionals should go to produce the shortest and most general programs that will work for all of the examples. One such system is by Bauer [78], which also decides which values in the program should be constants and which should be variables. It is visual since the user can specify the program execution using graphical traces.

4.7. VP, PBE, Interactive

Some of the most interesting systems fall into this final category. Except for AutoProgrammer [Biermann 76b], which is similar to Bauer's system (section 4.6), few attempt to do inferencing.

Pygmalion [Smith 77] was one of the seminal VP and PDE systems. It provides an iconic and "analogical" method for programming: concrete display images for data
and programs, called icons, are manipulated to create programs. The emphasis is on "doing" pictorially, rather than "telling". Thinglab [Borning 79 and 81] was designed to allow the user to describe and run complex simulations easily. A VP interface to Thinglab is described in [Borning 86]. Here the user can define new constraints among objects by specifying them graphically (see Figure 6). Also, if a class of objects can be created by combining already existing objects, then it can be programmed by example visually in Thinglab.

SmallStar [Halbert 81 and 84] uses PBE to allow the end user to program a prototype version of the Star [Smith 82] office workstation. When programming, the user simply goes into program mode, performs the operations that are to be remembered, and then leaves program mode. The operations are executed in the actual user interface of the system, which the user already knows. Since the system does not use inferencing, the user must differentiate constants from variables and explicitly add control structures (loops and conditionals). Halbert reports that Star users were able to create procedures for performing their office tasks with his system.

The goal of Rehearsal World is to allow teachers who do not know how to program to create computerized lessons easily [Gould 84]. Interactive graphics are heavily used to provide a "collaborative, evolutionary and exploratory" environment where programming is "quick, easy and fun." The metaphor presented to the user is a theater, where the screen is the stage and there are predefined performers that the user can direct to create a play (see Figure 7). The teacher developing the program sees at every point exactly what the student-user of the play will see. In addition, the teacher can have additional performers in the wings (so the student will not see them) that provide auxiliary functions such as flow control. Everything is made visible to the teachers, however, which allows their thinking to be concrete, rather than abstract as in conventional programming environments. When a new performer is needed, often its code can be created by example, but when this is not possible, some Smalltalk code must be written. The static representation for all performers is Smalltalk code, which can be edited by those who know how.

5. Taxonomy of Program Visualization Systems.

The systems discussed in this section are not programming systems since code is created in the conventional manner. Graphics in these are used to illustrate some aspect of the program after it is written. Figure 8 shows some Program Visualization systems classified by whether they attempt to illustrate the code or the data of a program (some provide both), and whether the displays are static or dynamic.

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flowcharts</td>
<td>BALSA</td>
</tr>
<tr>
<td>[Haibt 59]</td>
<td>[Brown 84]</td>
</tr>
<tr>
<td>[Baecker 86]</td>
<td>PV Prototype</td>
</tr>
<tr>
<td>PegaSys</td>
<td>[Brown 85]</td>
</tr>
<tr>
<td>[Moriconi 85]</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>5.2</td>
</tr>
<tr>
<td>Code</td>
<td>Two Systems</td>
</tr>
<tr>
<td>[Baecker 75]</td>
<td>[Brown 84]</td>
</tr>
<tr>
<td>Sorting Out Sorting</td>
<td>[Baecker 81]</td>
</tr>
<tr>
<td>[Brown 85]</td>
<td>BALSA</td>
</tr>
<tr>
<td>[Baecker 68]</td>
<td>Animation Kit</td>
</tr>
<tr>
<td>Incense</td>
<td>(London 85)</td>
</tr>
<tr>
<td>[Myers 90,93]</td>
<td>PV Prototype</td>
</tr>
<tr>
<td>[Brown 85]</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>5.4</td>
</tr>
</tbody>
</table>

Figure 8. Classification of Program Visualization Systems by whether they illustrate code or data, and whether they are dynamic or static. The small numbers refer to the section in which the group is discussed.

5.1. Static code visualization

The earliest example ofVisualization is undoubtedly the flowchart. As early as 1959, there were programs that automatically created graphical flowcharts from Fortran or assembly language programs [Haibt 59]. A modern static system [Baecker 86] has attempted to add multiple fonts, nice formatting, and other graphics to make code easier to read (see Figure 9).

Figure 7. A screen from Rehearsal World [Gould 84] showing the basic menu (on the left) and the standard set of "performers".
5.2. Dynamic code visualization

Most systems in this class do not animate the code itself, but rather dynamically show what parts of the code are being executed as the program is run using some sort of highlighting. Examples are [Brown 84] and [Brown 85], which are discussed in section 5.4.

5.3. Static data visualization

A very early system for the TX-2 computer could produce static pictures of the display file to aid in debugging [Baecker 68]. Incense [Myers 80] and 82 automatically generates static pictorial displays for data structures. The pictures include curved lines with arrowheads for pointers and stacked boxes for arrays and records, as well as user-defined displays (see Figure 10). The goal was to make debugging easier by presenting data structures to programmers in a way that they would draw them by hand on paper.

Figure 10.
A display produced automatically by Incense of 3 records containing pointers [Myers 80].

5.4. Dynamic data visualization

The first few systems in this class actually fall between dynamic and static. They were computer systems designed to create movies of data structures and algorithms (e.g. sorting) for teaching computer science [Baecker 75][Baecker 81]. The systems did not produce the animations in real time, however, so the movies were made frame by frame. The Balsa system from Brown University [Brown 84] was also designed to teach students about programming, but it produces the illustrations in real time on a personal workstation.

The "PV Prototype" [Brown 85] was designed to aid in debugging and program understanding, and it supports dynamic displays of data and easier construction of user-defined displays. Another system with similar goals, written in Smalltalk, features smooth transitions from one state to another [London 85].

6. Areas for Future Research.

Although these systems are attractive for a number of reasons, and some have been successfully used, they share a number of unsolved problems which are fruitful areas for future research.

6.1. Visual Programming

- Difficulty with large programs or large data: Almost all visual representations are physically larger than the text they replace, so there is often a problem that too little will fit on the screen. This problem is alleviated to some extent by scrolling and abstraction.
- Lack of functionality: Many VP systems work only in a limited domain.
- Inefficiency: Most VP systems run programs very slowly.
- Unstructured programs: Many VP systems promote unstructured programming practices (like GOTO). Many do not provide abstraction mechanisms (procedures, local variables, etc.) which are necessary for programs of a reasonable size.
- Static representations of programs that are hard to understand: For flowcharts, AMBIT and similar systems, the program begins to look like a maze of wires. For Behavioral World and similar systems, the static representation is simply normal linear code.
- No place for comments: An interesting point is that virtually no VP system provides a place for comments.

6.2. Inferential Programming by Example.

The major problem with these systems is that the user provides no guidance about the structure of the program so each new example can radically change the program. The programmer often knows, for example, which values are variables and which are constants or where conditions should go, but there is no way to directly convey this information to these systems. Choosing the correct examples requires great skill, and it is often difficult in these systems to modify programs once they exist.

The generated procedures are often "convoluted and unstructured" [Bauer 78, p. 131] and the user is never sure if the generated procedure is correct unless he reads the code and checks it explicitly. If this is required, however, most of the advantage of PBE is lost since the user must then know how to program in order to check it. In fact,
the central idea of this "inductive generalization" program-
ning is directly opposed to the modern software-
engineering idea that testing with a few examples can
never guarantee that a program is correct. Clearly, gen-
erating a program from a few examples has the same prob-
lem.

6.3. Programming with Example.

Programming with Example systems that do not
attempt to do inferencing have been more successful. Most
of these are VP systems, so they share the problems listed
in section 6.1. Some additional problems with these sys-
tems (from [Halbert 84]) include:

- Lack of static representation: These systems often have
  no user-understandable static representation for pro-
grams.
- Problem with editing programs: The lack of a static
  representation makes editing difficult. One alternative
  is to run a program from the beginning, but this may
take a long time. Specifying a change for the middle of
  a program by example may not be possible without run-
  ning it from the beginning since the state of the world
  may not be set up correctly to allow the user to specify
  the change. Saving periodic snapshots of the system
  state may alleviate this problem, but there may be a
  great deal of information to save. In addition, a change
  may invalidate steps of the program that come after-
wards.
- Problem with data description: It is often difficult to
  specify what the procedures should operate on: con-
  stants, user-specified data, or data found somewhere in
  the system qualified by its type, location, name, etc.
  Unless there is some explicit mechanism for the user to
tell it, the system does not know why the user chose
  some particular data. Also, if the user specifies the
  same data item in two different places, is this a coin-
  cidence, or should the identical item be used in both
  places?
- Problem with control structure: When specifying a con-
  ditional by example, only one branch can be traveled.
  To go back and travel the other branch, a different
  example must be given, and the system must be
  returned to the correct state for the "IF" statement to
  be re-evaluated. An additional problem is how to
  specify where in the program the conditionals and loops
  should be placed.
- Lack of functionality: Many systems only provide Pro-
  gramming with Example for a few data types and a
  small number of operations. As a patch, some provide
  escapes to conventional programming languages when
  PBE is insufficient.
- Avoiding the destruction of real data or other undesir-
  able consequences: In an environment such as the
  office, where actions in the system may have external
  consequences, it may be undesirable for the system to
  actually perform certain actions while the program is
  being written.

6.4. Program Visualization.

Data Visualization systems have the following problems:

- It is difficult to pick the appropriate picture for a data
  abstraction.
- After the picture is chosen, it usually requires a great
  deal of programming to get the system to produce that
  picture.
- The amount of data is usually large, and it is difficult
to fit enough on the screen.
- Related to the above is the layout problem: deciding
  where to place many differently shaped two-
dimensional pictures, which may have arrows and lines
  connecting them.
- For dynamic data visualization, it is difficult to specify
  when the displays should be updated. Issues of aesthetic-
  ics in timing are very important to produce useful ani-
mations.

For code, there is a separate set of problems:

- There has not been much work on interesting displays
  or ways to show progress.
- Like all the other Visual systems, there is the problem
  of the size of the pictures. Ways must be found to
decide what code to display and how to compress pro-
cedures to fit on the screen.
- When code and data are animated together, it is
difficult for the user to tell what data is being manipu-
lated by what parts of the code, so some way must be
found to show the relationships of variables to the
displayed data.

7. Conclusion.

Visual Programming, Programming by Example and
Program Visualization are all exciting areas of active com-
puter science research, and they promise to improve the
user interface to programming environments. A number of
interesting systems have been created in each area, and
there are some that cross the boundaries. This paper has
attempted to classify some of these systems and present
the general problems with them in hopes that this will
clarify the use of the terms and provide a context for
future research.

ACKNOWLEDGEMENTS

For help and support of this article, I would like to thank Bill
Buxton, Ron Ba'echer, Bernta Myers, and many others at the
University of Toronto. The research described in this paper was
partially funded by the National Science and Engineering
Research Council (NSERC) of Canada.

REFERENCES

Graphical Syntax-Directed Editor for Structured Program-
[Attardi 82] Giuseppe Attardi and Maria Simi. "Extending the
Power of Programming by Example," SIGOA Conference on
Office Information Systems, Philadelphia, PA, Jun. 21-23,
Debugging: The Interrogation of Complex Data Structures,
"(Summary only) First Hawaii International Conference on
[Ba'echer 75] R.M.Ba'echer. "Two Systems which Produce
Animated Animated Representations of the Execution of
158-167.
[Ba'echer 81] Ron Ba'echer. Sorting out Sorting. 16mm color,
sound film, 25 minutes. Dynamics Graphics Project, Com-
puter Systems Research Institute, University ofToronto,
Toronto, Ontario, Canada. 1981. Presented at ACM SIG-