A Framework for Describing and Implementing Software Visualization Systems

John Domingue
Blaine A. Price[1]
Marc Eisenstadt
Human Cognition Research Laboratory
The Open University
Milton Keynes, UK, MK7 6AA
Phone: +44 908 65-3800 (Fax: -3169)
Internet e-mail: j.b.domingue@open.ac.uk

Abstract

In recent years many prototype systems have been developed for graphically visualizing program execution in an attempt to create a better interface between software engineers and their programs. Several classification-based taxonomies have been proposed to describe computer program visualization systems and general frameworks have been suggested for implementation. In this paper we provide a framework for both describing existing systems and implementing new ones. We demonstrate the utility of automatic program visualization by re-implementing a number of classic systems using this framework.

Résumé

Récemment on a développé beaucoup de systèmes prototype pour visualiser graphiquement l'exécution d'un programme afin de créer une meilleure interface entre les créateurs de logiciel et leurs programmes. Plusieurs taxonomies basées sur une classification ont été proposées pour décrire des systèmes de visualization de programmes et des cadres généraux ont été proposés pour leur réalisation. Dans ce papier nous fournissons un cadre dans lequel on peut décrire des sytèmes actuels et exécuter de nouveaux systèmes. On montre l'utilité de visualization automatique de programmes en ré-exécutant quelques anciens systèmes dans le cadre de ce prototype.

Keywords: Program Visualization, Algorithm Animation, CASE, Debugging Aids, Software Visualization, Software Engineering

Introduction: What is Software Visualization?

The tools traditionally used by software engineers to help monitor and analyse program execution have been plain ASCII-text based debugging environments which usually allow the user to trace the currently executing code, stop and start execution at arbitrary points, and examine the contents of data structures. Although they can be understood by experts, these tools have a limited pedagogic value and by the early 1980's the work of Baecker and Sherman (1981) and Brown (1988) showed how algorithms could be animated with cartoon-like displays that show a high level abstraction of a program's code and data (Brown referred to this as "algorithm animation").

A concurrent development in the mid-1980's was the appearance of systems which displayed graphical representations that were more tightly coupled with a program's code or data and showed more or less faithful representations of the code as it was executing. Although the displays were not as rich as the custom built algorithm animations, these systems were closer to the tools that software engineers might use. These "program animators" together with the algorithm animators became known as "program visualization" systems. We prefer the more descriptive term "software visualization" (Price, Small, & Baecker, 1992) which encompasses both algorithm and program visualization as well as the visualization of multi-program software systems. In this paper we will use the term software visualization (SV) to describe systems that use visual (and other) media to enhance one programmer's understanding of another's work (or his own).

Classifying SV Systems

One of the first taxonomies of SV was that of Myers (1986) (updated later as (Myers, 1990)), which served to differentiate SV, visual programming, and programming-by-example. In classifying SV systems, Myers used only two dimensions: static vs. dynamic and code vs. data. The first dimension is based on the style of implementation;

[1]Also affiliated with: The Dynamic Graphics Project, Computer Systems Research Institute, The University of Toronto, Toronto, Canada M5S 1A1

static displays show one or more motionless graphics representing the state of the program at a particular point in time while animated displays show an image which changes as the program executes. The second dimension describes the type of data being visualized, be it the program source code or its data structures.

The taxonomies that have been proposed since Myers have also used few dimensions, which seems to ignore that fact that there are many styles for implementation and interaction as well as different machine architectures and ways of utilizing them. Price, Small, and Baecker (1992) recently proposed a taxonomy which describes 6 broad categories for classifying SV systems: scope, content, form, method, interaction, and effectiveness. Each of these categories has between three and seven characteristics, for a total of thirty dimensions to describe each system. This pragmatic classification system provides a means for comparing the functionality and performance of a wide range of SV systems, but it does not provide a language or framework for implementing new systems. Eisenstadt et al. (1990) described nine qualitative dimensions of visual computing environments which can form the basis of a language for describing SV systems, but these serve only to describe the attributes of systems rather than drive their construction.

From Taxonomy to Framework and System: "what goes on?"

Taxonomies are useful, but we need more if we are to provide a firm basis upon which to describe SV systems in depth, let alone implement them. A *framework* for describing SV systems could provide extra leverage by being a little more prescriptive, i.e. making a commitment regarding how to approach the design and construction of SV systems. In fact, it is not a very big step from specifying such a framework to designing a system for *building* a SV system (SV system-building system). An important difference is that the former activity is merely a paper exercise, whereas the latter activity is intended to lead towards a working tool. Indeed, the latter activity serves as a useful forcing function: it encourages us to build re-usable libraries of software that we believe encapsulate important generalizations about SV system-building. The proof of the soundness of a design built in this way lies in the ability to use it both to reverse-engineer existing SV systems and construct new systems with ease.

Programming Language Visualization vs Algorithm Animation

Several of the noteworthy SV building systems and frameworks focus on algorithm animation, which means that the animations that they produce are custom designed and each new program requires manual annotation to animate it. Programs are animated in BALSA (Brown, 1988) by adding calls to the animation system at "interesting events" in the code. Systems implemented by Stasko (1990) and London and Duisberg (1985) provide facilities for smooth transitions in animations based on "interesting event" calls.

In our work, we have focussed on supporting the construction of systems which visualize the execution of programming languages. By visualizing a programming language interpreter (or compiler) one also, in some sense, automatically gets a visualization for any program written in that language (thus achieving the *automatic* goal suggested by Price, Small, & Baecker (1992)). Programming language visualization (PLV) and algorithm animation (AA) overlap, but there are differences in the approach. AA systems typically show a very high level picture of a program's execution and the images that it generates can be far removed from the data structures contained in the program. The animations cover a narrow set of programs (typically a single algorithm). PLVs on the other hand have to deal with any program which can be realised in the language. Thus PLV displays usually have much simpler images than AA displays since they must be highly generalized whereas AA displays can be custom tuned. The problem for an AA system is to show the characteristics (signature) of an algorithm as clearly as possible. The problem for a PLV is to allow arbitrarily large execution spaces to be examined in a comprehensible fashion.

Our approach is to concentrate primarily on PLV, but to provide generalizations which are applicable to AA as well. In the rest of this paper we describe the design of a SV system-building system (and framework) called Viz, which we have implemented as a prototype running in Common Lisp and CLOS on Sun workstations. Our Viz implementation has already been used to reconstruct three well known PLV systems: an OPS-5 visualization system (based on TRI (Domingue & Eisenstadt, 1989)), a Prolog visualization system (based on TPM (Eisenstadt & Brayshaw, 1988)), and a Lisp tracer (based on the Symbolics™ tracer). After describing the Viz architecture, we explain how one of these reconstructions was implemented. In order to explore the relationship between Viz's PLV-oriented approach and AA-oriented systems we have also used Viz to implement some of the animations from Brown's BALSA (sorting) and Stasko's TANGO (bin-packing). We conclude with a comparison of the terminology used in BALSA, TANGO, and Viz to describe abstractions and we highlight aspects of the Viz design.

Viz Architecture

In Viz, we consider program execution to be a series of *history events* happening to (or perpetrated by) *players*. To allow SV system builders considerable freedom, a player can be any part of a program, such as a function, a data structure, or a line of code. Each player has a name and is in some *state*, which may change when a history event occurs for that player. A player may also contain other players, enabling groups of players to be formed. History events are like Brown's "interesting events" in BALSA—each event corresponds to some code being executed in the

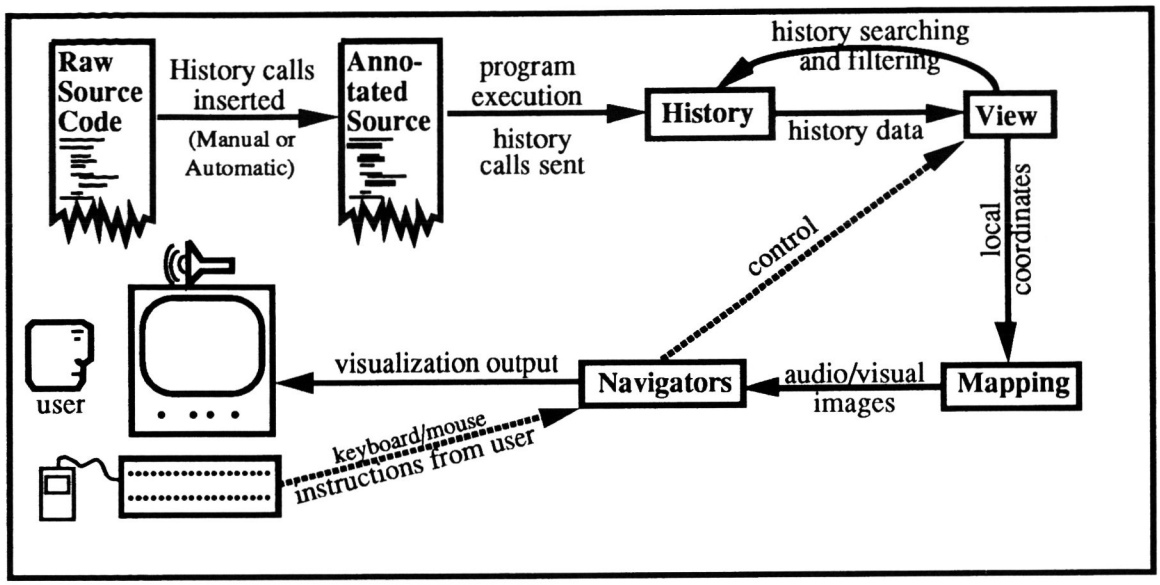


Figure 1. The Architecture of Viz

program or some data changing its value. These events are recorded in the history module, which allows them to be accessed by the user and "replayed." Events and states are *mapped* into a visual representation which is accessible to the end-user (the programmers who need to use the SV system, not the SV system builder). But the mapping is not just a question of storing pixel patterns to correspond to different events and states—we also need to specify different views, and ways of navigating around them. The main ingredients of Viz are:

- Histories: a record of key events that occur over time as the program runs, with each event belonging to a player; each event is linked to some part of the code and may cause a player to change its state (there is also some pre-history information available before the program begins running, such as the static program source code hierarchy and initial player states).
- Views: the style in which a particular set of players, states or events is presented, such as using text, a tree, or a plotted graph; each view uses its own style and emphasizes a particular dimension of the data that it is displaying.
- Mappings: the encodings used by a player to show its state changes in diagrammatic or textual form on a view using some kind of graphical language, typography, or sound; some of a player's mappings may be for the exclusive use of its navigators.
- Navigators: the tools or techniques making up the interface that allows the user to traverse a view, move between multiple views, change scale, compress or expand objects, and move forward or backward in time through the histories.

This framework is equally at home dealing with either program code or algorithms, since a player and its history events may represent anything from a low-level (program code) abstraction such as "invoke a function call" to a high level (algorithm) abstraction such as "insert a pointer into a hash table."

Figure 1 shows the general architecture of Viz. The target system source code is annotated to generate history calls. When the system being visualized is a programming language, hooks into the interpreter or compiler are used to generate history events. As the code executes, the inserted calls cause "interesting events" regarding players to be recorded in the history module.

When the user runs the visualization, the view module reads the history data at the request of the navigator. The view module sets the layout of the history events and sends local coordinates for each history datum through the mapping module, which draws a graphical or textual representation for each event. The screen images are then transformed and presented on the screen by the navigator. The user interacts with the visualization using the navigator, which sends control signals to the view module to cause all changes in the visualization, such as panning, zooming, local compression and expansion, and moving forward and backward in time through the program execution space.

Histories

The first task for the visualization programmer using Viz is to decide what types of events may occur during program execution, which elements in the program will be the players and how the players change state. After defining these, the programmer may insert *create new player* and

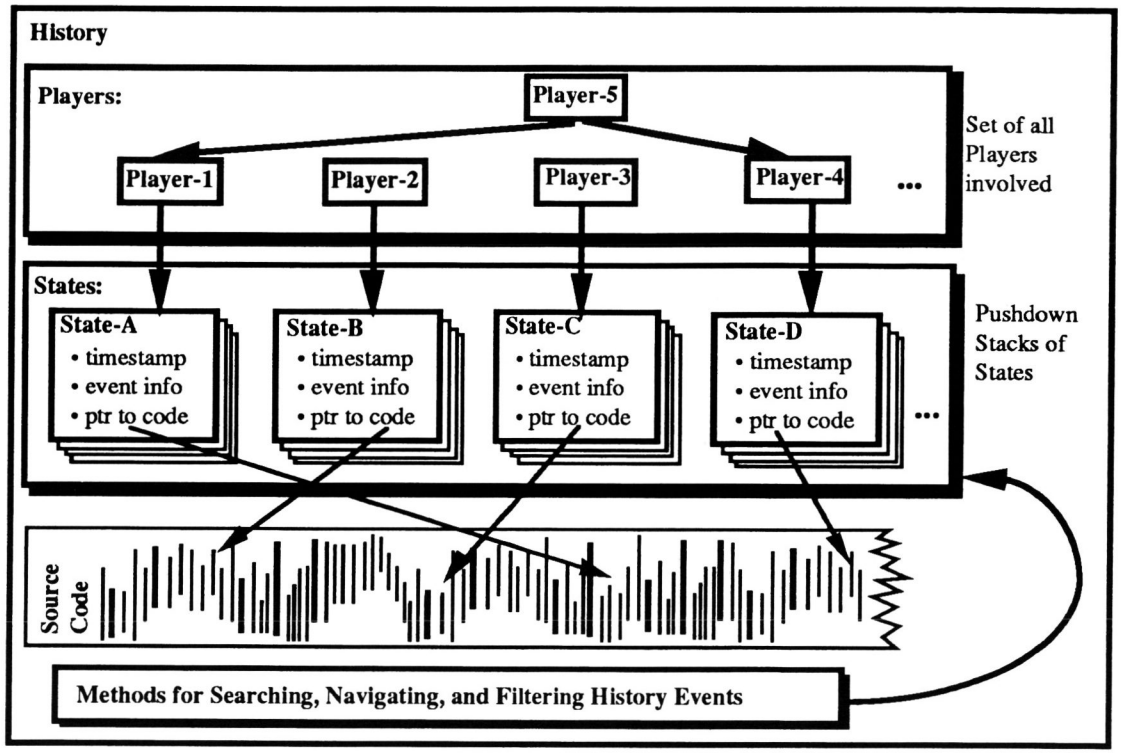


Figure 2. A Prototypical history structure.

note event calls in the code, which form the interface between a program and its visualization.

Figure 2 shows a prototypical history structure in the history module. This consists of a set of players and a sequence of history states. Each player has a name, a pointer to its current history state and a pushdown stack of previous states. A player may contain other players, as shown by player 5 in figure 2. This feature is useful in navigation. Each state has a timestamp, a pointer to the appropriate segment of source code and an event structure. As a program executes, new players and history states are created, and existing players are "moved" into new states, pushing previous states onto a stack. The various states of the players are caused by the different types of events.

The choice of players and event types together with the judicious placement of *note event* calls in code determine the execution model. Currently, we do not advocate any methodology for creating the execution model, except to point out that events are the "things that happen" in a program causing a player or players to move into a particular state.

Views

A view can be thought of as a perspective or window on some aspect of a program or algorithm, with (possibly) many views making up a visualization. There are some similarities between our views and the animation views, adapted from the Model-View-Controller paradigm, described by London and Duisberg (1985). The main difference is that within the animation views, the layout, handled by views in Viz, and appearance, handled by mappings in Viz, are handled together. Each view in Viz can be thought of as embodying a style of formatting collections of objects. Within Viz we have constructed a hierarchy of views, including text, graph, table and tree based views, each embodying a particular layout style, which can be used or specialized by the SV builder.

A view requests data from the history module and sends it to the mapping module, which decides the appearance. The view module then tells the mapping module where to display the mapped data. This means that the view module is concerned only with the position of the history data item, not its appearance.

The view module is also responsible for managing the compression (ellision) and expansion of elements in the display based on user commands from the navigator. If the user selects the compression of display elements, such as a subtree in a tree hierarchy, then the view module groups the players concerned into a new player which has its own mapping to represent the compressed players. When the navigator tells the view module that an element is to be expanded, the view disbands that new player and displays the individual mappings for the players.

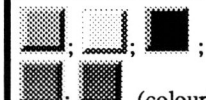
	Prolog	OPS5	Lisp	Sort	Bin-Packing
Player	predicate instantiation (goal)	rule	form	data item	data item bin
States	pending goal; succeeded; failed; failed on backtracking; redo-goal	failing to match working memory; firing	unevaluated; evaluated	location	attempting to fit; suceeded; new
Events	call; exit; fail-1st; fail-nth; redo	choose for firing	call; return	assignment of item to cell	attempt-fit succeed-fit new-item
Mappings	 (colour)	blank; 	-> *italic*; <- **bold**		
Views (in order of decreasing granularity)	tree: players, players current state; formatted text	table: players vs. cycles, player's state @ cycle formatted text	pretty printed code: player's current state	point plot: players, player's value & current state; formatted text	point plot using rectangles: bin-players and current state

Table 1: A Viz description of five example systems

Mappings

The goal of a mapping in Viz is to communicate the maximum amount of information about a player's state while imposing the least possible cognitive load on the user.

In conjecturing a theory of effectiveness of graphical languages, Mackinlay (1986) noted Cleveland and McGill's observation that people accomplish the perceptual tasks associated with the interpretation of graphical presentations with different degrees of accuracy. Using psychophysical results, Mackinlay extended Cleveland and McGill's work to show how different graphical techniques ranked in perceptual effectiveness for encoding quantitative, ordinal, and nominal data. He found that the position of the data item in the x-y plane is ranked first for all three types of data, which is why we separate the view layout from the mappings. The other techniques which may be varied to create an effective mapping are (in decreasing order of effectiveness): colour hue, texture, connection, containment, density (brightness), colour saturation, shape, length (size), angle or slope (orientation), and area (or volume).

A mapping in Viz is attached to a particular type of player, event or state, and view. Multi-method inheritance occurs over the class of entity and view, allowing a Viz user to formulate expressions such as "all entities in view-x are to be displayed as a filled triangle", "entity-y is always displayed as a white circle" and "entity-a is displayed as a circle in tree based views but as a square in all other views". Mappings can be inherited, forming an inheritance hierarchy in much the same fashion as views. Our future work in Viz will create a library of mappings.

Navigators

The Viz navigator module encapsulates the interface between the user and the visualization, although the methods for performing the navigation tasks are found in the view module, thus allowing custom navigation interfaces to be built independently of the task.

Our prototype provides a replay panel (see the screen snapshot in figure 3) for searching, which has buttons for moving to the beginning or end of the animation, single-stepping forward or backward, playing forward, fast-forwarding and stopping. Stepping in Viz involves notifying the history and view modules of the change of focus (the history module then selects the next appropriate event). Horizontal and vertical scroll bars are provided for panning while simple zoom in and zoom out buttons provide scaling. The user can select a fine grained view of a data element by clicking on it.

Examples Defined in Viz

The descriptions of the three systems that we have implemented using Viz are presented in table 1 along with the two examples from BALSA and TANGO animations. The table provides a summary of the players, states, events, mappings, and views used in each visualization. Each row represents a distinct Viz entity type and each column represents one of the visualizations. The player row lists the players which can take part in each example. The states

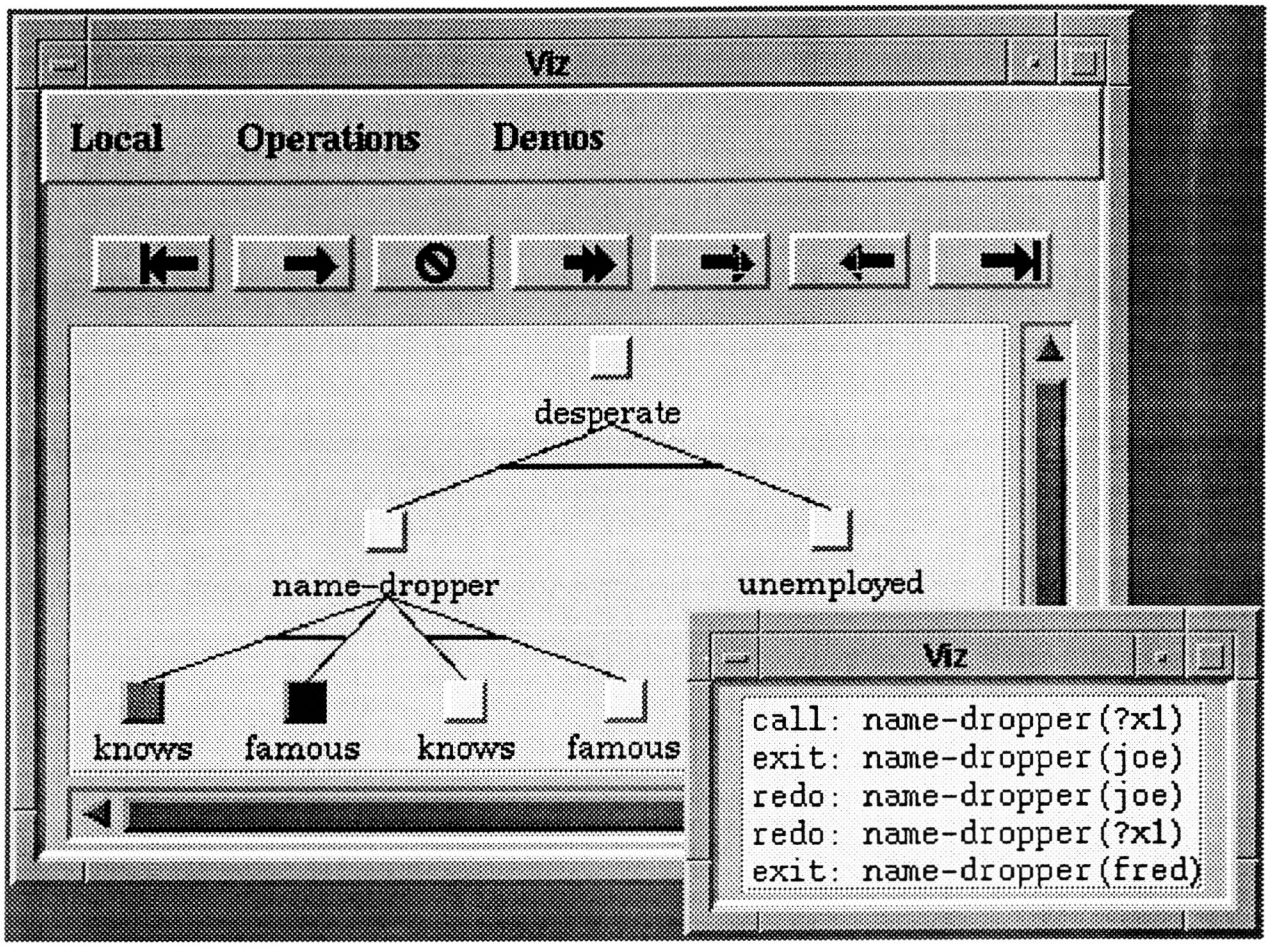


Figure 3: Screen Snapshot of Prolog Visualizer (greyscale from a colour screen)

row shows the possible states players can enter. The events row shows the events which cause state changes. The mappings row contains, in order, the icon mapping for each state. The views row lists the names of the possible views in decreasing order of granularity. The connection between a view and the history is also shown. We shall now explain the first column, the Prolog visualizer, in detail.

A Prolog Visualizer

The Prolog visualizer is based on the Transparent Prolog Machine (TPM) (Eisenstadt & Brayshaw, 1988). TPM uses an AND-OR tree representation where the nodes represent goals which are instantiated Prolog predicates, and the arcs represent conjunctions or disjunctions of subgoals.

The players in the visualization are the instantiated Prolog predicates or goals in the proof tree. The events, which are adapted from the Byrd Box Model (Byrd, 1980), are: call (trying to prove a goal), exit (a goal succeeding), fail-1st (a goal failing the first time attempted), fail-nth (a goal having succeeded earlier, later failing on backtracking), and redo (re-attempting to satisfy a goal). There is a corresponding state and mapping for each event type (shown in respective order so  equals call, equals exit, etc.).

The Prolog interpreter takes a list of goals left to prove. When no goals are left the environment is returned. The algorithm (adapted from (Nilsson, 1984)) for the interpreter is:

```
If there is nothing to prove
then return the environment; else
    if the first-goal-left-to-prove
    is true, then
        note event: succeed, the goal that
                    was proved, env,
                    and prove the
                    remaining goals; else
        note event: goal,
                first-goal-left-to-prove,
                env, and create a player for
                the-first-goal-left-to-prove,
                and
        loop for each clause in the database
                if the head of the clause
                matches the first goal then
```

```
                create a player for
                each of the subgoals
                in the clause
        if we prove the new list of
                goals (which is the
                body of the matched
                clause appended to the
                rest of the goals)
                then return the new
                environment else
        note event: redo,
        first-goal-left-to-prove, env,
note event: failure,
        first-goal-left-to-prove, env,
        and
return failed-to-prove
        first-goal-left-to-prove.
```

In the above, the algorithm is shown in italics while the Viz event calls are shown in plain text.

Figure 3 shows a screen snapshot of the proof for the goal ?- desperate(?x) given the following Prolog database:

```
desperate(?x) :-
  name-dropper(?x),
  unemployed(?x).

name-dropper(?x) :- knows(?x, ?y),
                    famous(?y).

name-dropper(?x) :- knows(?y, ?x),
                    famous(?y).

knows(joe, mick).
knows(charles, fred).
famous(mick).
famous(charles).
unemployed(fred).
```

Bearing in mind that atoms beginning with "?" depict variables in this approximation of Edinburgh-syntax Prolog, the first five lines of the code above defines a) the relation that someone is desperate if they are name dropper and unemployed; and b) a person is a name dropper if they know someone who is famous or if someone famous knows them. The rest of the code defines five "facts" about who knows who, who is famous and who is unemployed.

Because history events are invoked by inserting hooks into our own Prolog interpreter, the Viz implementation is straightforward once the machinery for players, views, mappings, and navigators is in place. The simple implementation, as described here, can deal with non-trivial cases of tricky backtracking and unification. Coping with arbitrarily large proof trees requires the definition of a "collapsed predicate" player. The tree beneath a collapsed predicate player would not be displayed unless requested (by clicking on it with the mouse). The current set of collapsed predicates would be chosen by the user. This collapsed predicate set would correspond to the segments of code the user deemed irrelevant and thus could be "black boxed away". In order for collapsed predicates to be distinguishable, the mapping for a collapsed predicate player would be a triangle. The collapsed predicate player would in fact contain the players within its subtree. A request to show the full sub-tree would result in the replacement of the single collapsed predicate player with the players it contained.

BALSA	TANGO	Viz	Comments
Interesting (Algorithm) Events	Algorithm Operations	Events and Create Players	The BALSA and TANGO terms are virtually identical while Viz events can be arranged hierarchically, and are designed to relate to the code rather than the algorithm.
Modellers	Image, Location, Path, and Transition	States and Players	In describing a visualization's internal representation, TANGO adds to the BALSA framework by providing 4 abstract data types (geared towards animation); Viz's states and players are program execution level abstractions.
Renderers	Animation Scenes	Mappings and Views	BALSA provides a general mechanism for each view while TANGO provides reusable libraries of animation scenes; Viz discriminates between the actual images that are mapped to the screen and the style in which they are displayed (the view).
		Navigators	BALSA and TANGO don't specify any kind of user interface interactions within the framework, nor techniques for dealing with arbitrarily large programs.
Adaptor and Update Messages		History	The Viz history is a structure for the collection of events, states, and players generated during program execution. The history module includes various searching and filtering functions.

Table 2: A Comparison of Terminology

Additional Systems

To fully exercise our evolving framework across a range of players, events, states, mappings, and views, we have also used Viz to re-implement the textual visualization provided by the Symbolics™ Lisp stepper/tracer which uses layout to summarize the execution history of the Lisp evaluator (the Viz implementation actually improves on this by using colour and typography as well) and the table based vizualization of an OPS-5 style rule interpreter. We have also duplicated some of the well known AA examples from BALSA (sorting) and TANGO (bin packing) to show that the system can be used to easily construct custom algorithm animations as well.

A Comparison of Viz, Balsa, and Tango Terminology

Although each of the goals of Viz, BALSA, and TANGO are somewhat different, the importance of the pioneering work of BALSA and TANGO is such that a close comparison of terminology is warranted. Viz terminology is designed to allow existing systems to be described as well as implement new ones. Table 2 shows the systems in left-to-right chronological order, mapping the similar terminology across systems where appropriate, and highlighting differences accordingly in the "comment" column.

Conclusions

The main goal in designing Viz was to provide a descriptive mechanism for understanding and explaining the diverse notations and methodologies underlying existing software visualization environments. Our re-implementation based approach is in contrast to the current literature (Eisenstadt et al., 1990; Green, 1989; Green, 1990; Myers, 1990; Price et al., 1992) which focusses on cognitive and notational dimensions and practical categories. The amount of effort involved in our approach is of the same order of magnitude as category or dimension based approaches. Each of the example systems was constructed within 1-2 days (this included many extensive alterations to the first version of Viz) and is of the order of 100 lines of code.

By providing a descriptive abstraction for internally expressing the state of an algorithm (players, events, and states) we have augmented earlier frameworks and added to the common language for representing algorithm animation designs. Since Viz provides visualization facilities for programming languages, we have provided a framework for generalized visualization that is applicable to software engineers since it provides visualizations that are automatic and faithful to the execution model of the language. The use of Viz to implement systems which differ widely in terms of their scope, content, form, method, interaction, and effectiveness, suggests that the framework is sufficient to design and implement a wide class of software visualization systems.

Acknowledgements: This research was supported by CEC ESPRIT-II Project 5365 (VITAL), the UK SERC/ESRC/MRC Joint Council Initiative on Cognitive Science and Human Computer Interaction Project 91/CS66, and NSERC. We wish to thank Ron Baecker, Mike Brayshaw, Thomas Green, and Marc Brown for their helpful comments on early drafts of this paper.

REFERENCES

Baecker, R. M. & Sherman, D. (1981). *Sorting Out Sorting*. narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH '81. Los Altos, CA: Morgan Kaufmann.

Brown, M. H. (1988). *Algorithm Animation*. New York: MIT Press.

Byrd, L. (1980). Understanding the Control Flow of Prolog Programs. In S. A. Tarnlund (Ed.), *The 1980 Logic Programming Workshop*, (pp. 127-138).

Domingue, J. & Eisenstadt, M. (1989). A New Metaphor for the Graphical Explanation of Forward Chaining Rule Execution. In *The Eleventh International Joint Conference on Artificial Intelligence*, (pp. 129-134).

Eisenstadt, M. & Brayshaw, M. (1988). The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *J. of Logic Prog.*, **5**(4), 1-66.

Eisenstadt, M., Domingue, J., Rajan, T., & Motta, E. (1990). Visual Knowledge Engineering. *IEEE Trans. on Software Engineering*, **16**(10), 1164-1177.

Green, T. R. G. (1989). Cognitive Dimensions of Notations. In A. Sutcliffe & L. Macaulay (Eds.), *People and Computers V* (pp. 443-460). Cambridge: Cambridge University Press.

Green, T. R. G. (1990). The Cognitive Dimension of Viscosity: a sticky problem for HCI. In D. Diaper, D. Gilmore, G. Cokton, & B. Shackel (Ed.), *INTERACT '90 Conference on Computer-Human Interaction*, (pp. 79-86). Amsterdam: Elsevier.

London, R. L. & Duisberg, R. A. (1985). Animating Programs using Smalltalk. *IEEE Computer*, **18**(8), 61-71.

Mackinlay, J. (1986). Automating the Design of Graphical Presentations of Relational Information. *ACM TOGS*, **5**(2), 110-141.

Myers, B. A. (1986). Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In M. Mantei & P. Orbeton (Ed.), *CHI '86 Human Factors in Computing Systems*, (pp. 59-66). New York: ACM.

Myers, B. A. (1990). Taxonomies of Visual Programming and Program Visualization. *JVLC*, **1**(1), 97-123.

Nilsson, M. (1984). The world's shortest Prolog interpreter? In J. A. Campbell (Eds.), *Implementations of Prolog* (pp. 87-92). Chichester, England: Ellis Horwood.

Price, B. A., Small, I. S., & Baecker, R. M. (1992). A Taxonomy of Software Visualization. In *The 25th Hawaii International Conference on System Sciences*, Volume II (pp.597-606). New York: IEEE.

Stasko, J. T. (1990). The Path-transition Paradigm: a practical methodology for adding animation to Program Interfaces. *J. of Vis. Lang. and Comp.*, **1**(3), 213-236.