The Visualization of Genetic Algorithms - Related Work

Trevor Collins

The Knowledge Systems Group,
The Knowledge Media Institute,
The Open University,
Walton Hall,
Milton Keynes,
UK, MK7 6AA.

Abstract

Genetic Algorithms are robust search algorithms capable of finding multiple solutions to complex problems. In order to ensure that the algorithm is working correctly it is necessary to examine the steps involved in its execution and the results produced at each stage. It is proposed that Software Visualization may be one technique that could support this task. This review examines a number of Software Visualization systems, discusses the key features that may prove useful for visualizing Genetic Algorithms, and presents some screen representations that illustrate some possible design configurations.

Contents List

Acknowledgements

This document is the result of several months of literature searching and system investigation. I would like to thank my supervisor, John Domingue for his help and encouragement during this time.

Some of the screen snap shots used in this review were taken from the on-line software visualization taxonomy written by Blaine Price, Ronald Baecker and Ian Small (http://www-cs.open.ac.uk/~doc/jvlc/JVLC-Body.html (Price, Baecker, & Small, 1993)), thanks go to Blaine Price for his permission to use these images.

I would also like to thank Steven Reiss for supplying the images used in the review sections on PECAN and FIELD, and John Stasko for the image of the DANCE animation development tool used in the TANGO algorithm animation system.

This work was supported by a three year postgraduate studentship from the Engineering and Physical Sciences Research Council.

1. Introduction

1.1. Genetic Algorithm Visualization

Genetic Algorithms (GAs) are robust search algorithms capable of finding multiple solutions to complex problems. During their search for such solutions GAs produce vast multi-dimensional data sets, from which possible solutions emerge. In order to guarantee the quality of those solutions the user must ensure that the algorithm's parameters are appropriately tuned. This requires insight into the algorithm's execution operations and output data set. It is proposed that this process would be best supported by the use of Software Visualization (SV).

This report is split into three sections. Section 1 is aimed at introducing software visualization, it explains the structure adopted in the review, and provides an overview of each system included. The purpose in presenting an overview of each system in section one is two fold; first of all to enable those readers who are not directly interested in each and every system to pick and choose detailed sections relevant to their interests, and secondly, for those who are interested in all of the systems to support their sense of orientation whilst examining each one.

Section two contains a detailed review of each system. Finally section three presents a list summarising each system, discusses some of the features which may prove useful for GA visualization, and presents some screen representations of possible design configurations.

1.2. What is Software Visualization?

From a Psychologist's view-point Visualization refers to the use of the "third eye", an inner eye, to mentally create internal views from information received via our five physical senses. We use this internal viewing system in order to make sense of the external world. Visualization systems therefore are created with the specific purpose of supporting the user's mental visualization process.

Software Visualization has been more formally defined as: "the use of the crafts of typography, graphic design , animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software" (Price, et al., 1993), page 213. Although this particular definition focuses mainly on the use of visual support, as the technology to involve some of our other four senses is now a practical reality, it is felt that a wider basis for supporting the mental visualization process is becoming possible.

1.3. Why is there a need for Software Visualization?

As was noted in the previous section, the purpose of the mental visualization process is to enable us to make sense of our external world. Mental visualization therefore, is central to understanding, in fact, it could even be considered as a foundational part of consciousness. In the case of understanding computer software it is when our mental visualization process is unable to construct an accurate representation of the dynamic behaviour of the static code that the majority of problems arise.

The process of correcting the errors resulting from such false interpretations is known to as "debugging". The debugging process itself can be further divided into two principal activities; Instrumentation and Localisation (Lieberman & Fry, 1995).

Instrumentation is the process of finding out what the behaviour of a particular piece of code is. Traditional approaches to this process include the use of tracers, breakpoints and the manual insertion of print statements (Lieberman & Fry, 1995). All of these techniques require the programmer to have some idea of the area in which the bug i.e. error occurred. On occasions where the programmer is unsure of the bug's source this can lead to an exhaustive and time-wasting search.

Localisation is the process of identifying the individual piece of code responsible for an error. The traditional approach for supporting this activity is an execution stepper (Lieberman & Fry, 1995). A stepper enables the programmer to incrementally step through the program examining the output at each stage. A fatal flaw of the traditional stepper is that it provides no control over the level of detail shown. Typically steppers stop before the execution of each expression and allow the programmer to decide whether or not to examine the next expression in detail. This again reintroduces the problem associated with traditional instrumentation tools, i.e. that the programmer must have some idea of the source of each bug.

Although these approaches go some of the way to helping Programmers analyse their code clearly more could be done. Software Visualization seeks to build on these traditional approaches through the introduction of typography, graphic design, animation, cinematography and Human Computer Interaction technology. By using these to improve the support for our mental visualization process SV will, in effect, improve our understanding and reduce the occurrence of errors. In cases where errors do occur, SV will improve the programmers ability to locate and understand the elements at fault.

1.4. Review Structure

SV systems are often classified into one of two broad categories; Program Visualization (PV) systems, and Algorithm Visualization (AV) systems. Program Visualization is the visualization of a program's code or data i.e. the basic elements that define the program. Algorithm Visualization on the other hand is the visualization of a program's algorithm i.e. the program's fundamental operations. This distinction may be made clearer with an example. Take a program that sorts a series of numbers into ascending order, it could be illustrated either by using Program Visualization, or, Algorithm Visualization.

Program Visualization could be used to illustrate the execution of the individual program commands by highlighting each line as it is executed, the effect on the program's data set could then be shown by printing each new version created. Algorithm Visualization on the other hand, could illustrate the series of numbers using an abstract representation such as printed blocks whose height represented the value of each number. The programs' operation could then be shown by moving each block into its correct position throughout the programs execution.

In other words, PV systems are tied to the programming language and data structures used because they rely on them for their presentation format, AV systems however, use abstract presentation formats and therefore, are not directly tied to the underlying programming language or the data structures used.

Although this dichotomy covers all SV systems the area of application for each PV system varies quite significantly. Therefore, the following review is structured, not as a simple dichotomy of PV versus AV, but it also sub-divides the PV systems into three groups, based on their intended area of application. The Venn diagram shown in figure 1 illustrates the categories chosen and the systems associated with each.

The three sub-groupings adopted for the review of Program Visualization systems were chosen in order to reflect the three main stages of program development, namely; Program Design, Program Construction, and Program Debugging. As can be seen in figure 1 above some systems support all three of these stages, in the review these are referred to as "Development Environments".

The application of Visualization to the Program Construction stage refers to systems capable of supporting the Programmer in the task of writing program code. However, this should not be confused with Visual

Programming where the Programmer constructs programs purely through the direct manipulation of graphical objects (icons).

Program Debugging is perhaps one of the most obvious application areas for Software Visualization as the Programmer's ability to identify and remove errors is largely dependant on their accurate understanding of the program's operation (see Section 1.3, above).

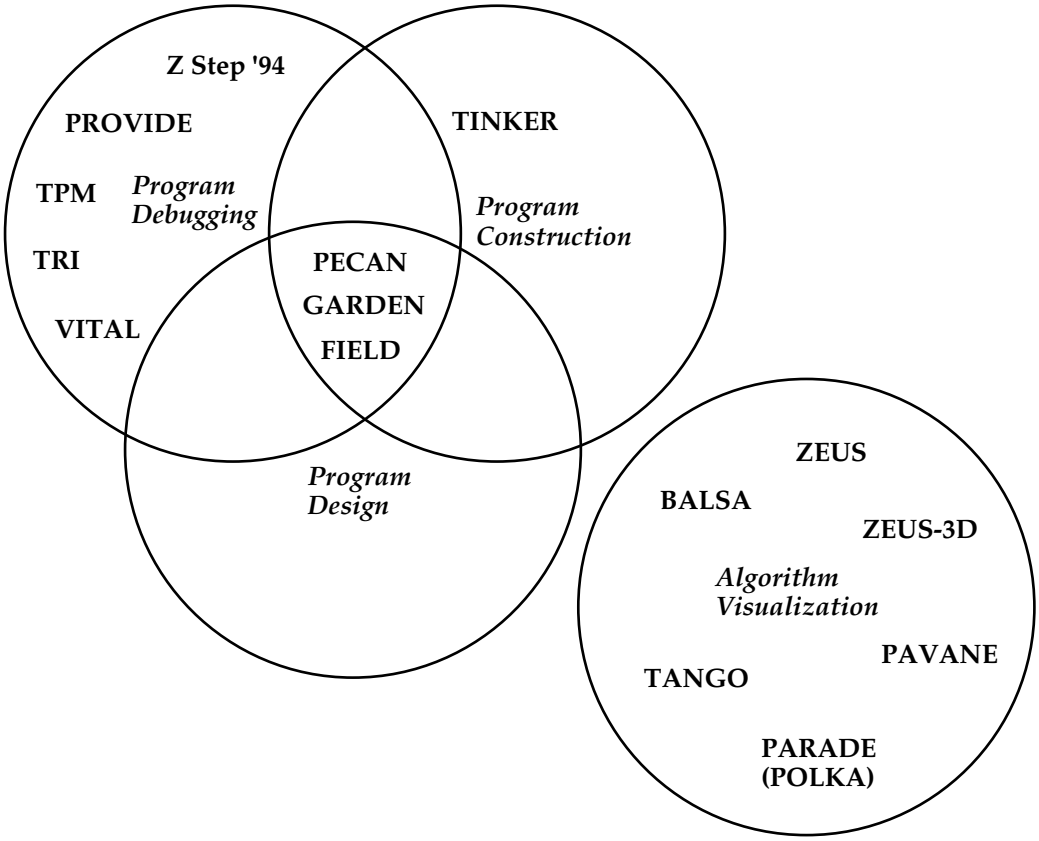

Figure 1. A Venn diagram illustrating the types of SV systems discussed in this review. The types of systems reviewed are labelled in italics and the names of the systems reviewed are labelled in capitals (except ZStep'94).

The final section of the review is that of Algorithm Visualization as previously noted the approach to visualizing a programs algorithm is quite distinct from visualizing its code or data. The review of algorithm visualization systems requires no sub-division as their area of application does not vary so widely.

1.5. Overview

This sub-section contains a brief overview of the systems reviewed in section two. This illustrates the structure used in the review and provides an initial introduction to the examined systems.

1.5.1. Development Environments

PECAN - A family of programming systems that supports the optional usage of structured templates for program commands, along with provision for maintaining concurrent graphical and textual views of a program.

GARDEN - An automated design environment to support conceptual programming. This is made up of three basic components; an object oriented framework, a multiwindow environment, and an environment support framework.

FIELD - A programming environment aimed at providing an integration framework for UNIX based tools with direct support for program and data visualization. Particularly significant for its introduction of the "selective broadcasting" integration mechanism.

1.5.2. Program Construction

TINKER - A program-by-example system that provides instant graphical and textual feedback. A function is created through a series of example steps, once completed the steps are generalised and displayed as program code.

1.5.3. Program Debugging

PROVIDE - A program evaluation and debugging environment implemented in order to illustrate the open access debugging approach proposed by Thomas Moher. This approach increased the direct engagement between people and their programs by the graphical presentation of program states at a chosen level of granularity, which the users could then directly manipulate and observe the results through immediate feedback.

Z Step '94 - This Lisp program debugging environment produces a (textual) code and (graphical) output view of the program under examination. It provides bi-directional control over both the code expression execution and output execution. It also includes the option to displays the evaluation of each program expression as it is found.

TPM - A program visualization system that is capable of automatically visualising any Prolog program ran through its interpreter. This

was one of the first systems to introduce bi-directional trace stepping. It uses a goal tree visual metaphor for both coarse grained and fine grained views and hence provides a highly consistent interface mechanism.

TRI - A very flexible rule-based program visualization system that presents a coarse-grained time specific view of the rule executions in a "rule graph", and more fine-grained rule specific views through "three way view frames", "predicate windows" and "node examination frames". Bi-directional trace stepping and proof tree views are adopted from the TPM system.

VITAL - This was a international research and development project over four and a half years aimed at the provision of methodological and software support for the development of large, industrial, embedded Knowledge-Based System applications. This project produced the "VITAL Workbench" and the "Viz" visualization framework. The most notable features of this project are the extensive provision of visualization support (via the "Viz" framework and software library) and the close integration of an assortment of design, development and validation tools through a common interface mechanism (the "VITAL Tower").

1.5.4. Algorithm Visualization

BALSA - This was one of the first real-time algorithm visualization systems. Its primary application is as a teaching aid for computer science students studying computer algorithms. The student interacts with the visualization system in a set-up and run cycle: the algorithm, parameters and display format are first set-up, and then run. The students then observe the operations of the algorithm during its execution.

ZEUS - This system uses objects, strong-typing, parallelism and supports the graphical development of views. Multi-view editing is introduced to this system through the adoption of an editor which enables the user to define, or edit, a view through either text or graphical manipulation.

ZEUS-3D - This is basically ZEUS plus 3D. The systematic approach taken with sound and colour in the ZEUS system is applied in this system to 3D. 3D is used for: the representation of additional information, the integration of two normally 2D views, and the representation of time in an originally 2D view.

TANGO - The implementation of an algorithm animation framework devised to produce smooth, colour animations. The framework has three main components; the algorithm itself, the mapping of the algorithm's operators to create a control file, and the resulting animation. The significance of this system is in the animation component where animations are created using the "Path Transition Paradigm". Four abstract data types are available within this paradigm; images, locations, paths and transitions from which all animations are constructed.

PAVANE - This is a parallel program visualization system capable of producing 3D colour animations. A shared dataspace is used to maintain the visualization of concurrent processes. A declarative approach is adopted in which program states are mapped from the state space through a proof space, object space, and animation space to produce the corresponding visualization images.

PARADE (POLKA) - A parallel program animation development environment designed to support the construction of application specific animations by programmer's with little or no experience of graphics programming. POLKA is the name of the animation component used to construct the visualizations used. POLKA is available in either 2D, on X Window systems, or 3D on Silicon Graphic's GL systems.

2. A Review of Software Visualization

The following section presents a detailed account of the systems mentioned in the above overview (section 1.5). Just like the previous section, the following is ordered with respect to the foreseen area of application. It is hoped that this will support the comparison of like with like and enable the reader to maintain a sense of perspective.

2.1. Program Development Environments

2.1.1. PECAN

The PECAN "family of programming systems" was developed by Steven Reiss at Brown University (Reiss, 1985). At the time of its development, the two main differentiating features between this system and several of its contemporaries were 1) its extensive use of the graphical facilities available on personal workstations, and 2) its support for multiple concurrent views.

Some of the key features of the PECAN system include: the provision of an undo facility whereby the user can undo and redo any action back to the beginning of the current session, the optional use of structured templates for support whilst writing the program code, and finally, the provision of a framework that enables the use of a variety of (algebraic) programming languages via the same commands.

PECAN provides support for three different types of views; program views, semantic views and run-time views. Examples of Program Views include the provision of a Syntax-directed Editor and Nassi-Shneiderman (structured flow-chart) graphs. Examples of the Semantic Views provided within PECAN include Symbol Table and Data Type views which illustrate the scope and type definition of the current symbol being edited, an Expression view that breaks the current expression down into an expression tree, and a Flow Graph view that shows the flow of control through the program on a system produced flow chart. Finally, an example of an Execution view is the Stack view this illustrates the current state of the execution stack. A series of sequential execution frames make up the Stack view each frame includes the frame reference, the variables used in that frame and their associated values.

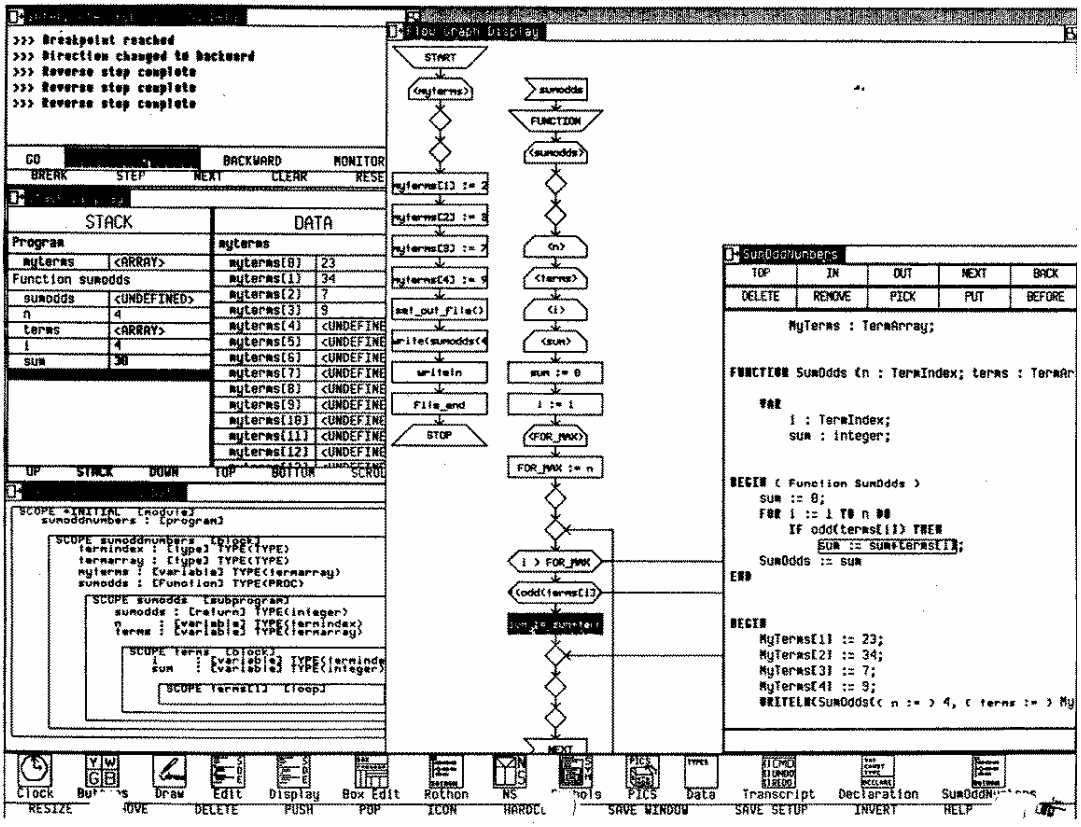

Figure 2. An example PECAN screen view. This figure shows examples of the Syntax-directed Editor (bottom right), Flow Graph (middle), Nassi-Shneiderman flow chart (lower left), and Execution and Data Stack views (middle left).

2.1.2. GARDEN

In the development of the GARDEN Automated Design Environment (Reiss, 1987) Reiss argued that a design environment should not constrain the designer to the design methods implemented therein. He proposed that designers naturally use a variety of techniques when designing their systems, and these techniques are often modified to suit the problem better with new strategies, or languages, being constructed to simplify the description of an otherwise complex design. In order to support such a free-form and non-invasive approach, Reiss developed the GARDEN design environment for conceptual programming. This environment is comprised of three basic components; an object oriented framework, a multiwindow environment and an underlying environmental support database.

The object oriented framework encourages the use of both data and control abstraction. The objects form a consistent basis for supporting multiple languages as any language can be defined in terms of its underlying constructs. Furthermore, as the objects can be used to represent programs GARDEN has no bias toward either graphical or textual syntactic forms. At

the time of publication however, the GARDEN environment provided only a single textual Lisp-like form of representation, although this is not suitable for all languages, the problem here is one of implementation rather than design.

Within the multiwindow environment three editors are available; the text editor, the graphical editor and the object-based browser. Additional windows can also be used to display interactive read-eval-print loops, user controlled system browsers, document editors, and any additional input/ output windows. Multiple instances of these display windows can be used simultaneously, with any necessary view updates being automatically maintained.

Finally the environmental support is provided by an underlying object-oriented database which is used to store all of the objects in current use, effectively saving the entire environment. With the necessary consistency checking and access control several programmers may share a common object space for collaborations. Version control is also supported by enabling users to create and restore different versions of their environment.

2.1.3 FIELD

The FIELD (Friendly Integrated Environment for Learning and Development) programming environment was designed for both teaching and research (Reiss, 1990). FIELD provides a framework for the integration of UNIX based tools into a consistent programming environment. The key concept underlying this environment is the integration mechanism called "selective broadcasting". At the beginning of a session each of the tools being used sends a message to the central message server notifying it of the messages for which they hold an interest. During use the central message server receives incoming messages from the tools and matches them to the interesting messages previously declared. The corresponding tools are then sent a copy of that message.

The FIELD environment emphasises program and data visualization. Support is provided for the automatic visualization of the user's data structures, including the dynamic updating of these structures whilst the program executes. Program execution can be monitored either by directly viewing the source code or indirectly by using a code visualization such as a call graph. Annotations may also be added in order to provide hooks for algorithm animations. An example screen view of the FIELD environment is given in figure 4.

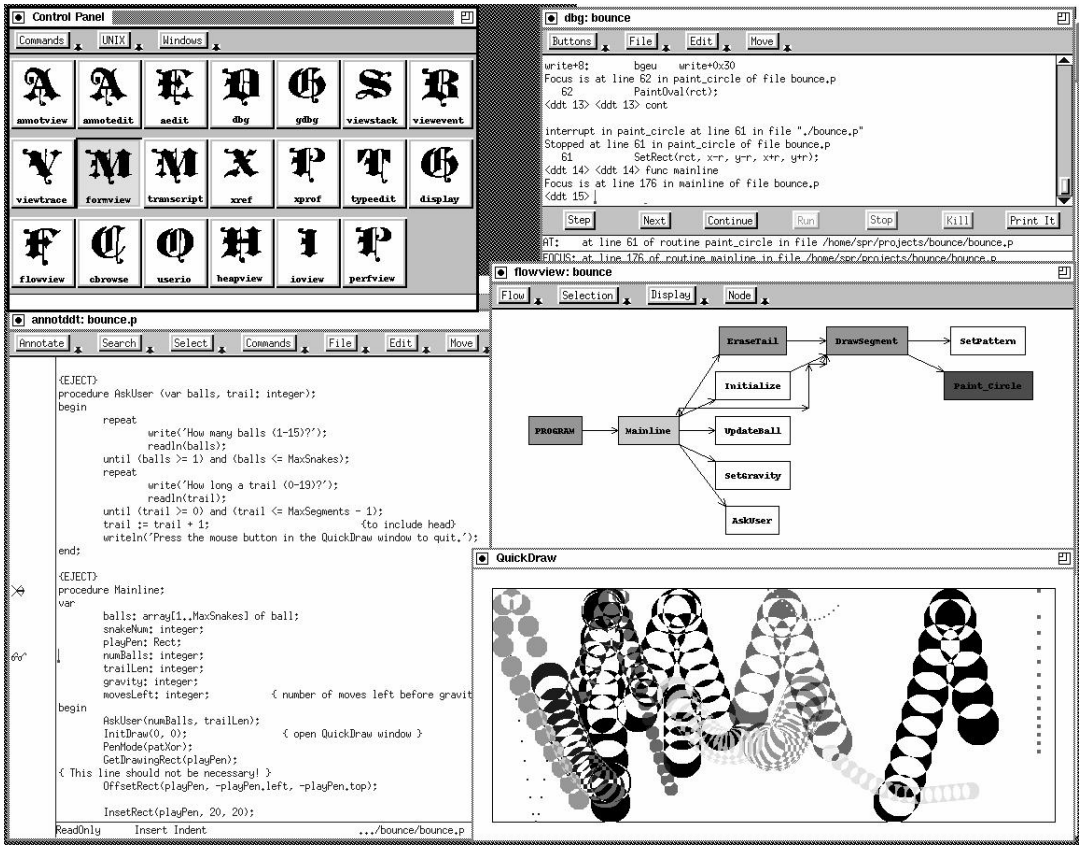

Figure 4. An example screen view of the FIELD environment illustrating: the control panel used to select the tools (top left), the debugging tool "dbg" (top right), the annotation editor "annotddt" (bottom left), the flow chart view "flowview" (middle right) and the execution view "QuickDraw" (bottom right). This figure is available from the World Wide Web research page at Brown University (see http://www.cs.brown.edu/ research/hpde/arpa-quad-94.html).

In a recent report on software visualization (Reiss & Cruz, 1994) Reiss and Cruz point out:

> "While this {the FIELD environment} provides information, it is not sufficient for real software understanding. The questions of interest often cannot be characterised by one of the canned views. Also, the amount of effort required to specify the particular aspect of the generic view that was needed was perceived as too much."

In an extension of the work originally done on the FIELD environment Reiss and Cruz are now examining the use of a visual query interface. They propose that by storing all the data produced in a virtual database, the user could then use a visual query interface to find any data of interest quickly and easily. Reiss and Cruz propose that querying for software understanding should, therefore, be at a high-level of abstraction, easy to do, and should make use of the same metaphors as the original visualization. The DOODLE

database query language was adopted for this purpose. Within DOODLE a visual constraint language is used to specify by example the data display.

Reiss and Cruz also proposed the adoption of 3D, reasoning that the use of an additional spatial dimension permits a more compact presentation of data and more flexibility for the design of new presentations. Three new tools are being created to support this, namely; PLUM, PEACH and TWIG. PLUM is a style manager which offers a variety of different, parametrized 3D presentation styles that can be combined hierarchically to form a desired view. PEACH is a hierarchical browser used as a front end for the PLUM style manager. TWIG is the name given to the specification definition language and its associated interpreter. TWIG was based on the GELO graphical editor used within the GARDEN environment and is designed to be suitable for graphical editing.

2.2. Program Development

2.2.1. TINKER

The TINKER programming environment is a Lisp-based environment in which a programmer may create a program through demonstrating its steps on representative examples (Lieberman, 1981).

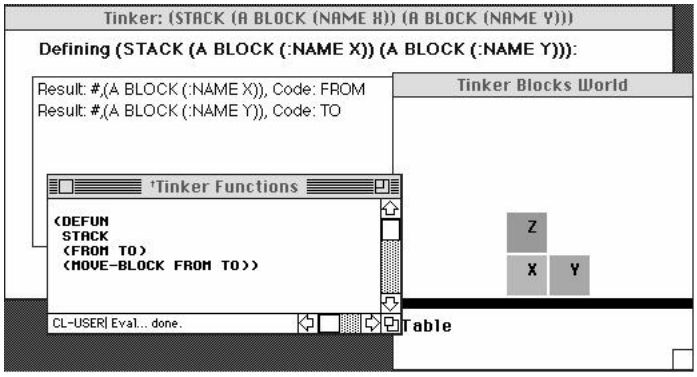

Figure 5. A TINKER screen view depicting the stack window (top-left), the function window (bottom-left) and the graphical execution window (bottom-right) of a block manipulation program. This figure was taken from (Lieberman, 1993), page 58.

As indicated in the above figure TINKER uses three types of windows; a stack window, a functions window and a graphical window. The stack window shows the previous states of the program's stack. The components of the program stack can be selected as arguments for further examples. The function window shows the generalised function code i.e. the code generalised from the user's example commands. The graphical window shows a visual representation of the current example. All three views are continuously updated throughout interaction in order to reflect the

programmer's actions. Hence, the user is provided with both code and graphical confirmation of their examples.

Once an example is completed the individual steps are generalised to form a program. It should be noted that unlike the majority of programming-by-example systems TINKER does not infer or assume anything it is not explicitly told by the user. Any conflicts which arise between examples, such as conditional conflicts, are taken up immediately with the user who must specify a predicate to distinguish between the conflicting cases.

An advantage of this incremental programming and immediate feedback approach is that the user can freely choose to program in a "top-down" or "bottom-up" manner. By constructing programs "bottom-up" the user may start with the most common base case and then further refine their program by adding more specific exceptions. The "top-down" approach supports the construction of specific examples which can be further reduced to more common cases.

An example of these two approaches could be taken from a definition for birds in which the most common case is that all birds fly, and a more specific case is that penguins and emus don't fly. A top-down definition would start with the specific example and work down to the more common base case, whereas the bottom-up definition would start with the base case and work up to the specifics. The approach adopted by the programmer will be dependent on the level of the information known.

Further details on this system are available from the World Wide Web, see http://lieber.www.media.mit.edu/people/lieber/Lieberary/Tinker/Tinker.html

2.3. Program Debugging

2.3.1. PROVIDE

PROVIDE is a PROcess Visualization and Debugging Environment introduced to demonstrate the principles behind Moher's open access approach to program execution and debugging (Moher, 1988). Standard dynamic debugging capabilities are extended by ((Moher, 1988), page 849);

- the use of computer graphics, rather than text, to depict process states;
- continuous, rather than query-driven, display of user-defined process state representations;
- direct manipulation of graphic process state representations, rather than a command language, for modifying data objects;

- random, rather than sequential and unidirectional, access to all process states arising during execution;
- interactive control over program granularity;
- state selection based on data states as well as control states, and
- automatic consistency maintenance of process state displays in the face of modifications to programs and data.

The primary goal of the PROVIDE system is to allow users to observe and control program execution at a suitable level of extraction. To this end PROVIDE enables users to specify any program objects of interest, these objects are then allocated a permanent display area during execution and are automatically maintained. Users can also modify views after compilation, as the use of deferred binding enables view editing to take place outside the traditional edit-compile-execute cycle. Another significant feature of PROVIDE is its playback facility in which users can control not only the apparent speed, but also, the apparent direction of execution (a feature also supported by Eisenstadt and Brayshaw's TPM system).

In order to implement these features Moher used a "continuous execution" technique so that whilst the program is being interpreted the state transitions are recorded in a process history database. This database is a shared resource between the interpreter and the interface. As the interface has access to all state transitions the user can identify any that may be of interest. Control over the display's apparent speed and direction during execution is not direct control over the program, but, over the accessing of that program's process history database.

When a view is modified the PROVIDE environment disposes of its future pre-computed data from the database and starts again from the current frame. As a result, although the term "process history database" may imply that the program's complete history could be viewed using any view, in practice only the history of each view is stored. Hence, visualizations of the program using a view not displayed at the original time of execution are not supported.

PROVIDE effectively demonstrated the principles behind Moher's approach to open access debugging. The major contribution made by PROVIDE is the design of a system capable of this level of user control in a practical environment.

2.3.2. Z Step '94

Z Step '94 is a program debugging environment developed by Henry Lieberman and Christopher Fry at MIT's Media Lab (Lieberman & Fry, 1995). This environment supports a "20-20 hindsight" view of debugging, in that it enables the Programmer to go back in execution-time and re-examine the

execution at varying levels of detail. A complete, incrementally generated history of a program's execution and its output is maintained by Z Step '94.

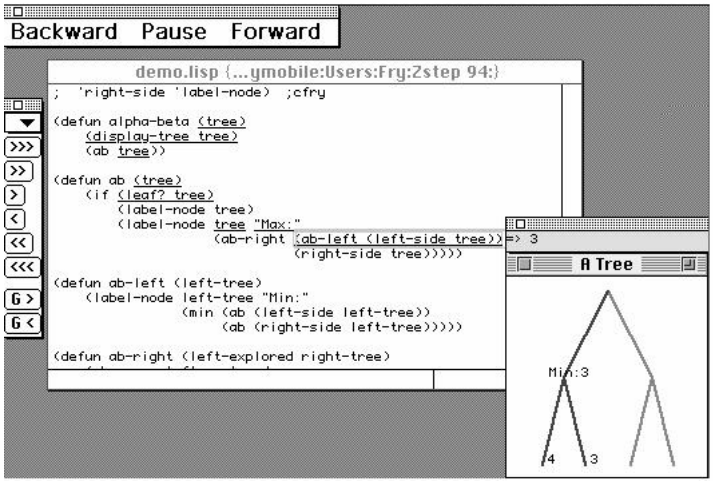

Figure 6. A screen view of Z Step '94 in which an alpha-beta tree drawing program is being debugged. There are five main components to the Z Step'94 environment; the bi-directional "video-recorder" control bar (mid-left), the "cruise control" panel (top-left), the code view (main window, centre screen), the "floating value window" (mid-right), and finally the (graphical) execution view (bottom-right). This figure was taken from (Lieberman & Fry, 1995).

A "video recorder" metaphor is used to create the control panel of a bi-directional stepper (figure 6). The upper section of the control panel enables a programmer to step through the execution expression-by-expression. The graphical stepper in the lower section enables the programmer to step through the graphical changes in the program's output. Lieberman and Fry reason that as a program's behaviour is generally considered by the Programmer in terms of the graphical output frames, the graphical stepper is, in effect, providing a stepper for controlling the behaviour of the program

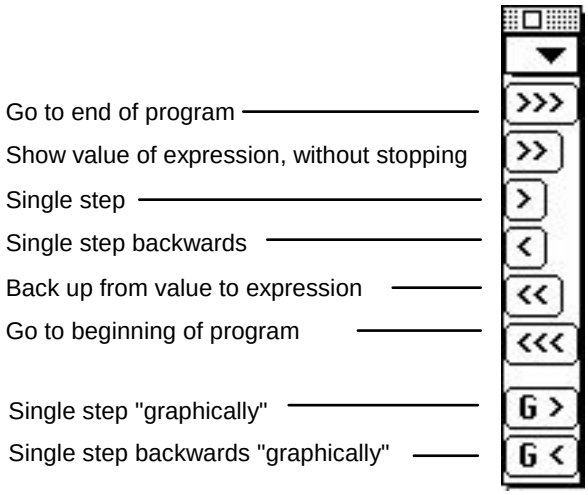

Figure 7. An illustration of the bi-directional "video recorder" control panel used in Z Step '94 (figure taken from (Lieberman & Fry, 1995)).

A "cruise control" panel is also available with which the programmer can set the execution speed of the program in either the forward or backward directions (figure 8). The cruise control window enables a programmer to click on either Backward, Pause, or, Forward buttons, with the distance from the middle of the pause button being used to define the speed of execution in the corresponding direction.

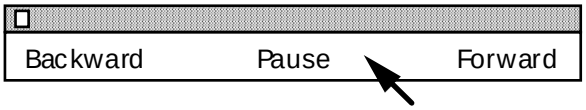

Figure 8. An illustration of the cruise control window used in Z Step '94 (figure taken from (Lieberman & Fry, 1995)).

Another feature of the Z Step '94 environment is the "floating value window". A problem with previous linear steppers and trace programs was that each expression was simply printed out as it was executed. This created the additional task of trying to match the printed expressions to the corresponding position in the code. As a method for supporting this matching some debugging systems provided a "follow the bouncing ball" interface which pointed to the current expression in the code before printing its value in another window. This however, although effective created a "ping-pong" effect as the programmer was forced to constantly switch their visual attention between the two windows of interest.

Lieberman and Fry proposes the "floating value window" as a solution to these problems. Rather than separately pointing to an expression and displaying its value, the floating value window itself moves through the code, pointing to the current expression being evaluated and displaying any values it returns. The background of the floating window is colour coded; a light green background indicates the expression is about to be evaluated, a light blue background indicates a returned value, and a yellow background indicates an error has occurred and the error message is displayed within the window.

As the floating window displays any error messages beside the expression that generated that error, the process of localisation is significantly reduced. Rather than disrupting program execution the Z Step '94 environment stops and enables the programmer to reverse over the events leading up to the error. Once a correction has been made the program can then be executed again. Note, this does not mean that the execution can continue where it left off, the entire program must be executed again for any corrections to be effective.

Four additional feature settings are discussed by Lieberman and Fry to support the instrumentation process. A “step to mouse position” mode enables the programmer to select an expression in the code and the stepper will go either forward or backward to that expression’s evaluation. The “show value under mouse” mode is a continuous mode which displays the latest evaluations for any expression pointed at with the mouse-cursor. This mode does not step forward to evaluate any new expressions it will only display those values which it has already stored.

A “current form history” feature can also be used so that an expression’s previous values can be displayed in a separate form history window when selected by the programmer. Finally, the “values filter feature” supports the display of a selected condition expression’s previously satisfying values. By further selecting an individual value the stepper will step back to the corresponding evaluation.

There is also support for correspondence between the views. As the stepper is used to step through the expressions, the execution view is updated to reflect the corresponding dynamic behaviour of the program. Furthermore, if the programmer selects a graphical object in the execution view the stepper positions the text view at the code expression responsible for that objects creation.

The significant contribution of the Z Step ‘94 environment is its integration of reversibility, code animation, and correspondence between code expressions, values and graphical output. Further details on this system may be found on the World Wide Web, see http://lieber.www.media.mit.edu/people/lieber/Lieberary/ZStep/ZStep.html

2.3.3. TPM

The Transparent Prolog Machine (TPM) is a visualization system capable of visualising any Prolog program (Eisenstadt & Brayshaw, 1987). TPM is a continuation of some of the work done on the Prolog Tracing Package (PTP) by Marc Eisenstadt in the early 1980’s (Eisenstadt, 1984).

PTP uses a technique developed by Eisenstadt known as “retrospective zooming” (Eisenstadt, 1985). By storing all of a program’s trace details internally this technique enables the user to decide upon the current level of detail exposed, and therefore customise the trace to their debugging task. This technique is also supported in TPM by the provision of two alternative viewing structures known as coarse-grained and fine-grained views.

A common visual metaphor is used for both of these views, namely; the Prolog program's "goal tree". Each node on a goal tree represents a command in the Prolog program and each layer in the goal tree represents a layer in the program . In a coarse-grained view a goal's type is indicated by the node's shape, its status is indicated by the fill colour and node-outline (see table 1).

Node Feature	*Goal Attribute*
Shape	Type
square	user-defined goal
circle	system primitive
triangle	compressed tree section
Fill Colour & Outline	Status
white (or green)	successful goal
white (or green) with a thick node outline	currently pending goal
black (or red)	failed goal
grey scale (or pink)	an initially successful goal which failed on backtracking.

Table 1. A description of the visual components of a long-grained view goal tree.

A commonly cited fault of tree-based visualizations is their ability to effectively illustrate large programs. In the case of TPM this problem is avoided by the provision of a compression facility with which the user may compress irrelevant sections of the goal tree into single nodes. These compressed sections are then indicated by a triangular node in the goal tree (table 1). Compounded with the fact that TPM supports multiple views and view scrolling, this makes the problems associated with displaying huge goal trees virtually insignificant. An example of a coarse-grained view is given in figure 9.

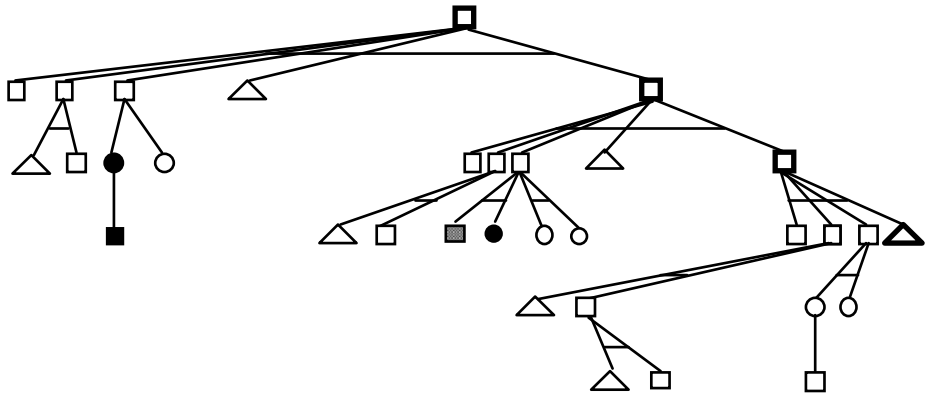

Figure 9. An example of a coarse-grained view used by TPM.

The fine-grained view of the program's goal tree provides a more detailed view of the goals. Each user-defined goal in the goal tree is replaced by a "Procedural Status Box" (see figure 10). This box illustrates the current status of each goal and its respective clauses. For both goals and clauses a tick is used to indicate success and a cross to indicate failure. In the case of system primitives a circle is used rather than a procedural status box.

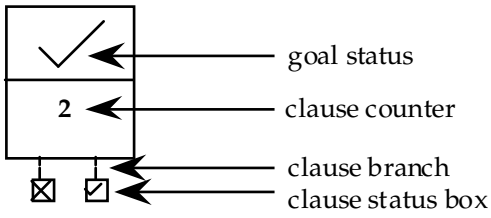

Figure 10. A Procedural Status Box.

Each status box provides additional detail about the individual goals and their respective sub-goals (commonly referred to as "clauses"). A printed representation of each goal is presented along side each status box, this includes the current variable instantiations (enclosed in lozenges) and the passing of variables between goals (illustrated using arrows). These fine-grained views are called AORTA diagrams, which stands for And/OR Trees-Augmented. An example Prolog program and it's corresponding AORTA diagram is given in figure 11.

```
older(X, Y) :-
                age(X, AgeOfX),
                age(Y, AgeOfY),
                AgeOfX > AgeOfY.

age(john, 27).
age(tom, 18).
age(sue, 24).

?- older(john, sue).
yes
```

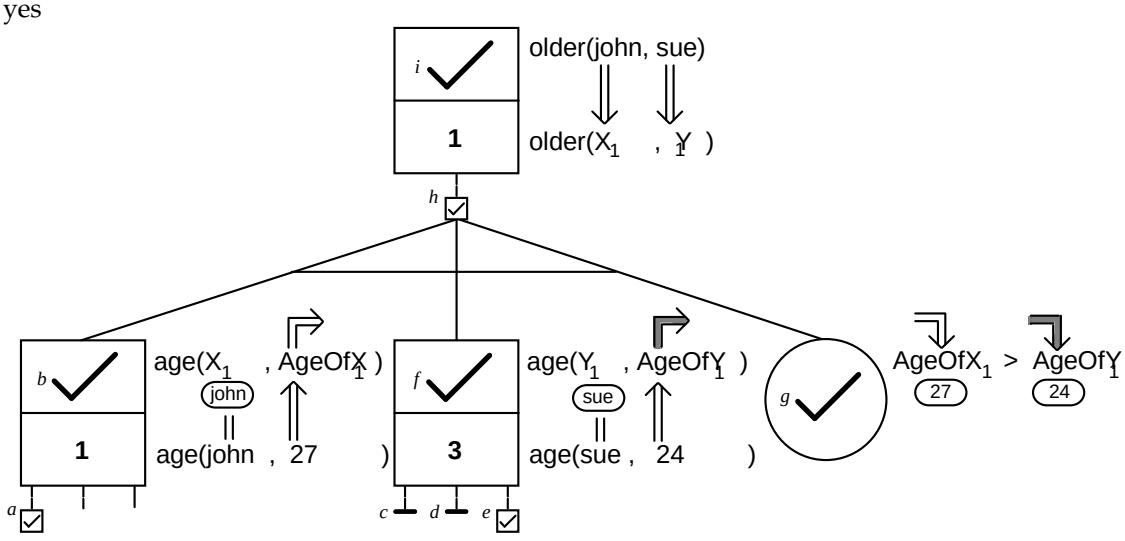

Figure 11. An example Prolog program and its respective AORTA diagram, as used for displaying fine-grained views within TPM. The small letters in italics indicate the order in which the status symbols appeared. System primitives are displayed as circular nodes.

As previously noted a Prolog program's trace details are stored internally by TPM this provides the data source for a replay panel. The replay panel within TPM can be used in order to control the current position within the trace history. In addition to the replay controls their are four additional command icons, these empower the user to; set pause ports (also known as breakpoints) for a node, refocus the display causing a selected node to be the top node in a new goal tree view, display the location of a goal in the source code, or, zoom-in to a fine-grained view of a selected node. The icons used in the replay panel are illustrated in figure 12.

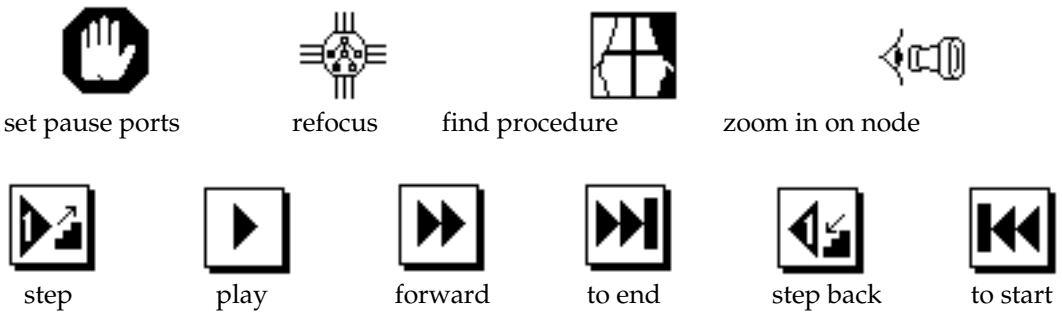

Figure 12. The command and control icons available within the TPM replay panel.

The two most important features of TPM are its ability to support synchronised coarse-grained and fine-grained views, retrospective zooming, and user directed tracing. A screen shot is given in figure 13 in order to provide an impression of a typical TPM scenario.

TPM is now a commercially available system and has been adopted by the Open University as an aid for teaching Prolog on their DM862 - Intensive Prolog course (details are available via the World Wide Web at http://kmi.open.ac.uk/courses/DM862.html). Mike Brayshaw has recently been working on a version of TPM suitable for a parallel version of Prolog known as Parlog. This system uses the same visual metaphors and replay panel as TPM but is extended to visualise the concurrent processes involved in parallel programming (see (Brayshaw, 1990)).

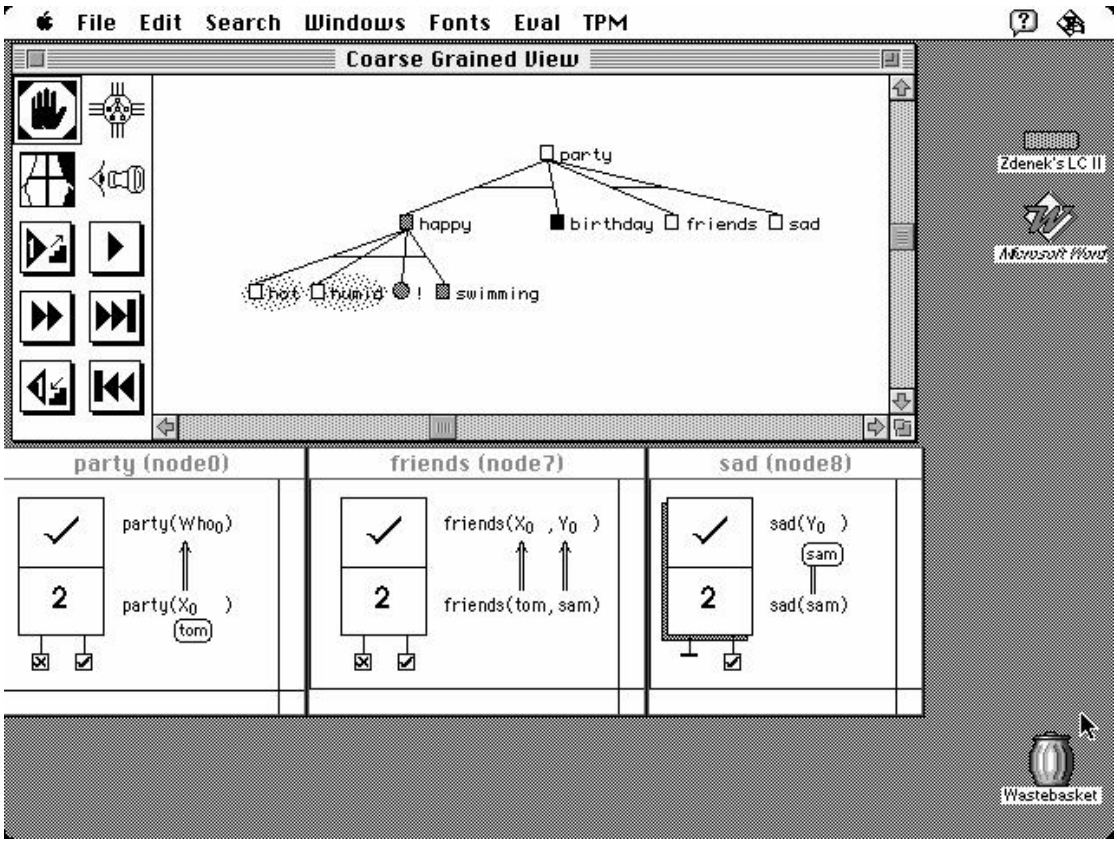

Figure 13. A screen shot taken from TPM version 1.11.

2.3.4. TRI

TRI stands for the Transparent Rule Interpreter (Domingue & Eisenstadt, 1989) which was developed by John Domingue at The Open University as part of the KEATS project (Knowledge Engineer's Assistant) (Motta, Rajan, & Eisenstadt, 1989). TRI is a maintenance tool designed for use on rule based programs written in OPS 5.

In order to debug a rule based program the Programmer must understand not only the specific components of the program but also how those components are structured to form a Knowledge Base. Therefore, a course-grained view is required in order to get an overall idea of the program's structure, as well as a fine-grained view so that the Programmer can understand the specific program details (such as rule unification, instantiation and the working memory state).

TRI supports the dual requirements of the debugging process by using a "rule graph" for coarse-grained viewing and "view frames" for fine-grained viewing. A rule graph provides an explicit representation of a program's execution cycles (shown along the x-axis) and the associated rule actions taken (shown along the y-axis, see figure 14).

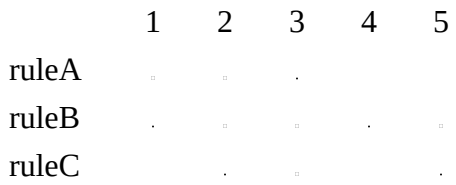

Figure 14. An example rule graph illustrating the execution of a set of three rules over five execution cycles. A plus sign indicates a rule fired. A triangle (not included in this graph) indicates that a rule fired and backward chaining occurred. A box indicates that a rule was considered but did not fire.

The view frames provide more detailed views of the program's history. Three different types of view frames have been developed; Three Way View Frames, Predicate Windows and Node Examination Frames. The view frames used are under the direct control of the user who can create, move and delete view frames at will. A high level of selectivity within these frames provides an easy and consistent method of interaction for viewing and manipulating rule components.

Three Way View Frames provide an opportunity for the user to view either the rules, predicates or working memory components of their program. These can be viewed either individually or up to all three views may be displayed at the same time. TRI also offers support for carrying out operations on any item within a frame view such as: describing a working memory element; describing a predicate, or; describing, viewing, or, editing a rule.

Predicate Windows can be used to display the working memory elements associated with a selected predicate or set of predicates. A cycle number is displayed beside each element indicating the cycle during which that element was deposited in working memory.

Node Examination Frames provide fine-grained views of the nodes in the rule graph and are intended to highlight why a rule didn't fire during a particular cycle. A rule graph node is selected using the mouse in order to create a Node Examination Frame containing the corresponding rule's definition and instantiations during the corresponding execution cycle (see figure 15).

The rule instantiations in a Node Examination Frame can also be selected using the mouse in order that they may be further examined. Rule instantiations in which backward chaining occurred are visualised using proof trees, these trees are based on the coarse-grained views used in TPM (see section 2.3.3). An example of a close-up of a backward-chaining proof tree is shown in figure 16.

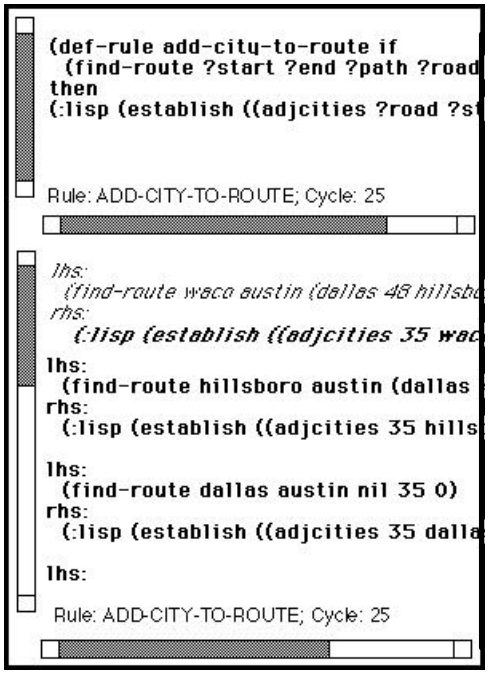

Figure 15. An example of a node examination frame from a route finding rulebase, the upper section displays the rule's definition and the lower section its instantiations. The most recent instantiation of a rule is highlighted using italics. Instantiations shown in bold indicate that when the rule fired backward chaining occurred. This view would have been produced by selecting the rule graph node corresponding to the add-city-to-route rule at cycle number 25.

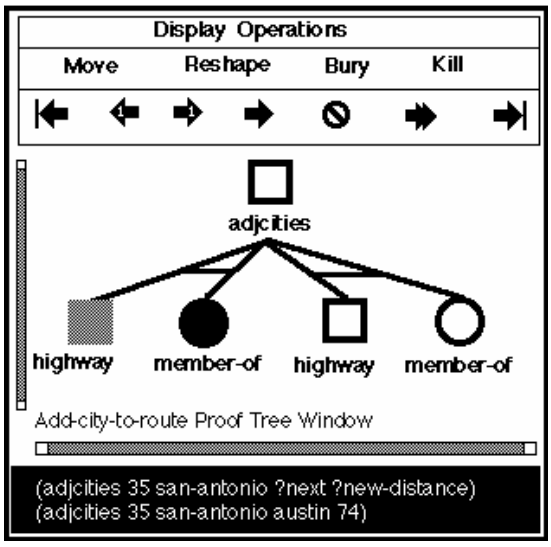

Figure 16. An example of a backward-chaining proof tree for an **adjcities** rule instantiation.

The execution control mechanism used within TRI is also taken from Eisenstadt and Brayshaw's TPM system. Using the replay panel a user can either; step forwards or backwards through the execution cycles, or, replay the program's history. The replay panel used in TRI is shown in figure 17.

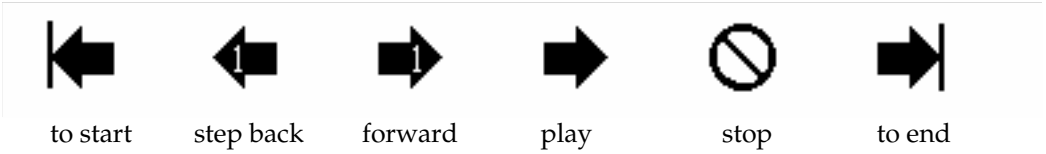

Figure 17. A representation of the replay panel used within TRI. Each control button's action is explained in the added text.

All of the views within TRI are synchronised and provide the user with both coarse-grained and fine-grained simultaneous views of their rule based programs. A screen shot containing examples of all of the views discussed is shown in figure 18.

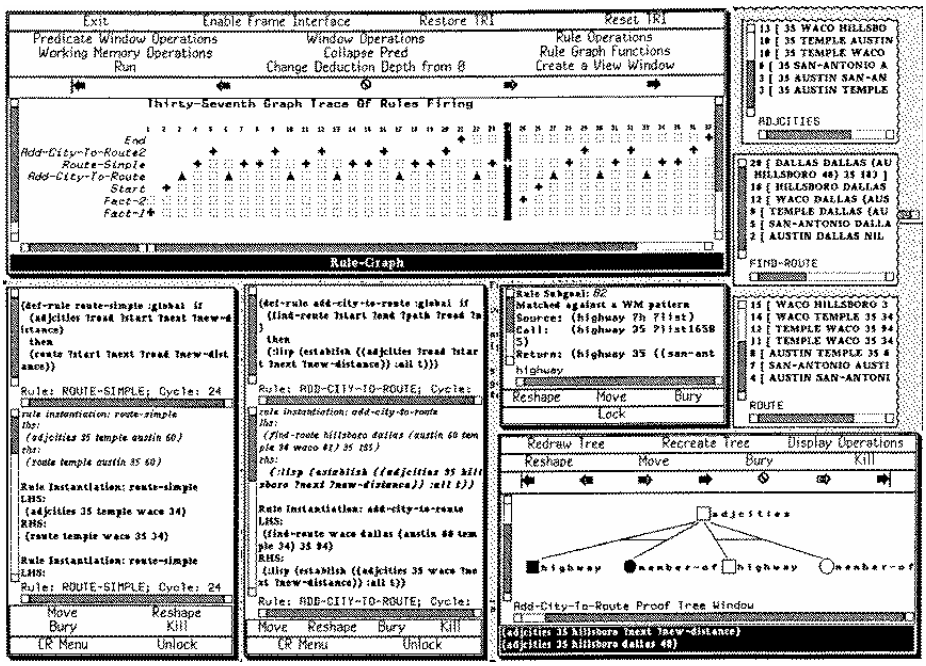

Figure 18. A screen shot of TRI, taken from (Domingue & Eisenstadt, 1989).

2.3.5. VITAL

The VITAL project was a four and a half year research and development project completed in April 1995. This was an ESPRIT II project that involved nine organisations in five different countries. Their aim was to provide both methodological and software support for the development of large, industrial, embedded Knowledge-Based System applications (Domingue, Motta, & Watt, 1993).

Software visualization was seen as an opportunity to enhance the users' control of the individual tools within the VITAL Workbench, in order to support this a separate visualization framework and software library, called Viz, was created (Domingue, Price, & Eisenstadt, 1992).

Viz enables the user (i.e. KBS developer) to define and construct visualizations of their systems using a very high level programming language. A program's execution data is stored in a history database which is used as the basis for creating different views of that program's execution. These views are then made available to the user, who can pick and choose which views they wish to see.

To orchestrate this Viz uses a story-telling metaphor in which the program's elements (i.e. functions, data structures, lines of code, etc.) can be referred to as "players". The players are identified, either manually by the user annotating the code selecting elements of interest, or, automatically by the program compiler. The program's execution is then stored in the history database as a series of "history events" (i.e. events that happen to, or are caused by, the players). Hence each player's name and state details are stored every time there is an event involving that player. An architecture diagram showing the different sub-components of Viz is given in figure 19.

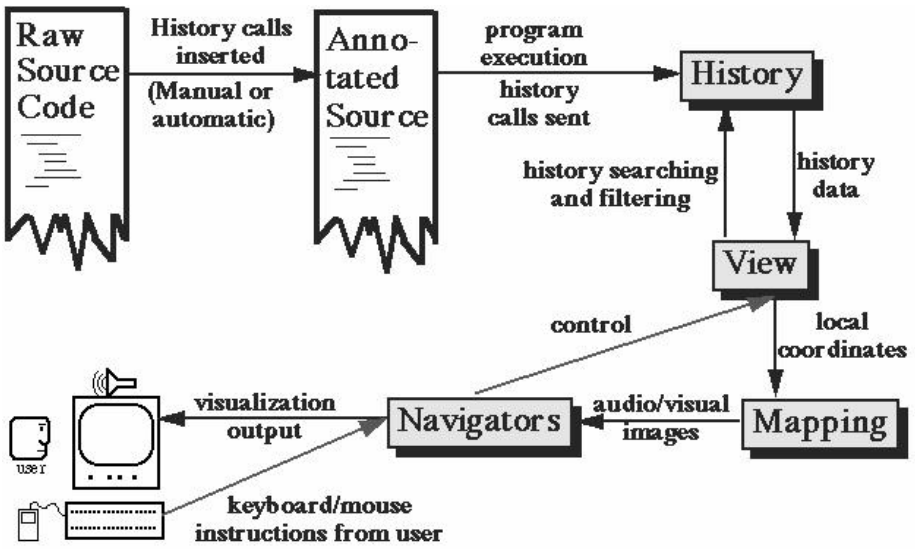

Figure 19. The architecture of Viz, taken from (Domingue, Eisenstadt, & Price, 1994), page 9.

There are four main components to Viz, the; "History", "Views", "Mappings" and "Navigators" components. The History component holds a record of all key events that occur over the duration of the program's execution. The Views component provides the styles in which a particular set of players, states, or, events can be presented. The Mappings are the encodings used to present the players' state changes, either graphically, or, aurally within each view. Finally, the Navigators are the tools or techniques used to interact with the user. They allow 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 programs execution.

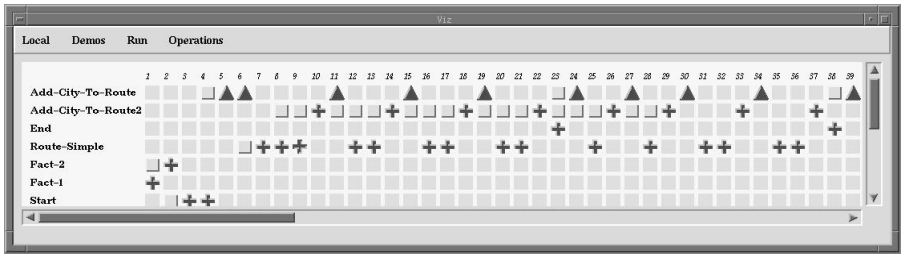

Figure 20. A screen snapshot of an example Viz visualization containing a TRI rule graph. A detailed description of the TRI rule graph may be found in the previous section (2.3.4). This view was taken from (Domingue, et al., 1993), page 14.

An example Viz visualization may help illustrate how these components are used in practice. The previous section (2.3.4) examined the development of the Transparent Rule Interpreter, a KBS maintenance and debugging tool. Figure 20 shows a screen view of a TRI rule graph created using Viz. A rule graph is a visualization of the execution of a set of rules over a series of execution cycles. The players, their possible states, program events, mappings and view details are summarised in table 2.

	Viz Definition
Players	rule, rule instantiation; *the rule player is composed of rule instantiation players*
States	failed to match working memory; *applies only to rule players* matched against working memory; *applies only to rule players* fired; *applies to rule and rule instantiation players* fired and invoked the backward chainer; *applies to rule and rule instantiation players*
Events	matching against working memory; firing; invoking the backward chainer
Mappings	failed to match working memory ->  matched against working memory -> fired ->  invoking the backward chainer -> 
Views (in decreasing order of granularity)	table: current state; *in a table based view display the rules current state* text: rule's instantiation states; *in a text based view show the rules instantiations states*

Table 2. A summary of the Viz definition used to create the TRI rule graph shown in figure 20 above. This table was also taken from (Domingue, et al., 1993), page 14.

The VITAL project was unique in that it represented an international effort to formulate a design, development and validation methodology for KBS applications. The Viz visualization framework and software library is capable of producing not only program visualizations (i.e. program data and code visualizations) but also algorithm visualizations.

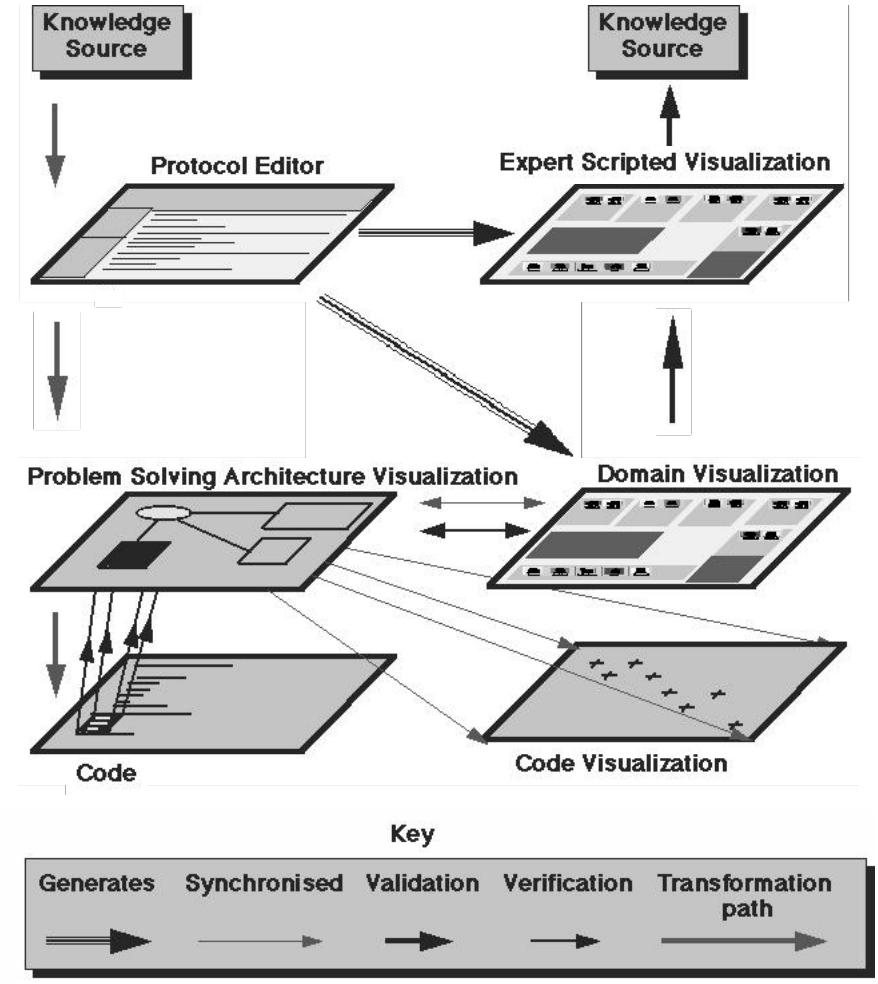

Figure 21. An illustration of the software visualization support provided by VITAL. This figure was taken from (Domingue, 1995), page 8.

The extent to which the Viz framework and library is used within the VITAL project is illustrated in figure 21. The Problem Solving Architecture and Code Visualizations are examples of program visualizations, they closely illustrate the actions of the code and states of the data being manipulated by the KBS. The Domain and Expert Scripted Visualizations are similar to algorithm visualizations where abstract representations are used to illustrate the KBS's operations.

For those interested in finding out more about the Viz framework a world wide web page maintained by John Domingue is available at The Open University, see http://kmi.open.ac.uk/~john/viz/viz.html. Further details on the VITAL Workbench are also available from The Open University world wide web server, see http://kmi.open.ac.uk/~john/vital/vital.html .

2.4. Algorithm Visualization

As noted previously in section 4 algorithm visualizations monitor the fundamental operations of an algorithm. These operations cannot be deduced for an arbitrary algorithm automatically, but must be identified by someone who has knowledge of that particular algorithm. The following sub-sections present six algorithm visualization systems and discusses their respective merits. The last two of these six systems are specifically aimed at providing support for parallel programming. This is inherently more complex than animating a serial program because of the non-deterministic nature of parallel programs.

2.4.1. BALSA

The BALSA environment (Brown ALgorithm Simulator and Animator) supports a high-level user interface that allows users to interact with the dynamically changing graphical representations of their programs (Brown & Sedgewick, 1985). The use of colour and sound is introduced to algorithm animation within this environment, although its use is not extensive an acknowledgement is deserved. BALSA interaction is based around four different user types; the Algorithm Designer, the Animator, the Scriptwriter and finally, the End User.

The Algorithm Designer provides the programs to be animated, he or she identifies any "interesting events" which need to be visualised, and contribute to the design of the graphical representations used. The Animator's task is then to implement the views that make up the graphical presentations. The Scriptwriter is the person who constructs the scripts for the animation i.e. what information is shown to the user and when. Finally the End User makes use of these scripts and views the dynamic graphical representations of the algorithms.

The interaction style for an End User is referred to as a "set-up and run" cycle (Brown, 1988). In the set-up phase the End User arranges the display layout, the algorithms they wish to view, and the parameters they want to associate with each algorithm, including its input generator and output views (figure 22). Then once set-up the End User runs the algorithm and observes the results (figure 23).

Figure 22. A pair of screen shots depicting the set-up phase of a BALSA session for a number sorting algorithm. The first screen view illustrates the display layout selection dialogue in the centre of the screen. The second screen view illustrates the parameter selection dialogue. In this particular example the user may select the initial organisation of the numbers (currently set to a random ordering), the number of numbers to be sorted and, the random number generator's initial seed value.

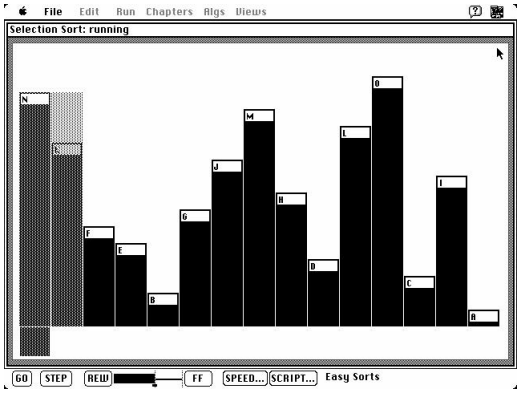

Figure 23. A screen shot depicting the run phase of a BALSA session for a number sorting algorithm. The numbers are represented by vertical columns, the size of each column represents the size of its associated number. As the numbers are sorted by the algorithm the columns move into place.

The control method for program execution is taken from Mac Pascal terminology. The End User can control the forward execution of a program by selecting one of five control commands shown in table 3.

Command	*Action*
Go	stop at the next stoppoint
GoGo	pause at the next stoppoint
Step	stop at the next steppoint
StepStep	pause at the next steppoint
Reset	reset the program

Table 3. The control commands available within the BALSA algorithm animation system, taken from (Brown, 1987), page 65. The term stop point is used to refer to any breakpoints

inserted in the program and the term steppoint is used to indicate the intermediate step positions in the code i.e. just after every command.

The End User can also edit the viewing position for each view. By selecting an individual view the user will display that view's control bars with which they can manipulate their viewing position. The End User can either pan vertically (using the right hand scroll bar), pan horizontally (using the scroll bar across the bottom of the view) or magnify their viewing position (using the left hand scroll bar).

For anyone interested in finding out more about BALSA there is a demo version for the Macintosh available via anonymous-ftp from ftp.dec.com/pub/DEC/macbalsa-demo.sit.hqx.Z and a source code version available from ftp.dec.com/pub/DEC/macbalsa-source.sit.hqx.Z. The MacBALSA user guide can also be found in ftp.dec.com/pub/DEC/ under the file name macbalsa-userguide.ps.Z .

2.4.2. TANGO

The TANGO algorithm animation system (Transition based ANimation GeneratiOn) was the resulting implementation of a framework devised by John Stasko for describing, specifying, analysing and formalising the elements involved in animating algorithms (Stasko, 1989). The framework contains three primary components; the Algorithm, Mapping and Animation Components (figure 24).

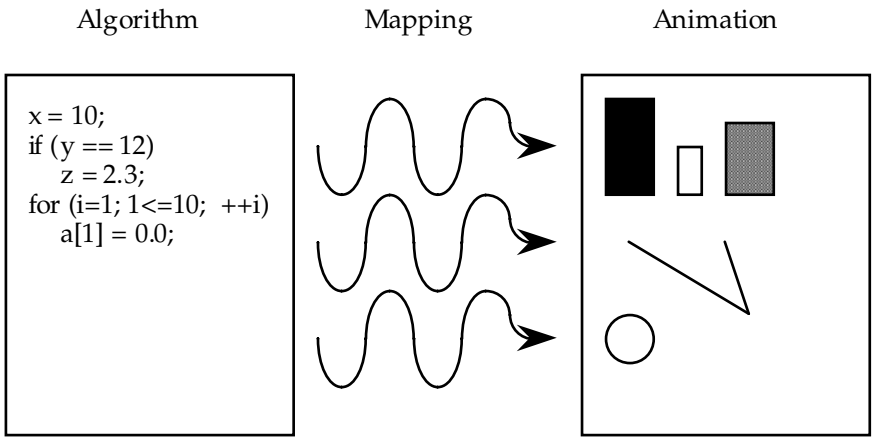

Figure 24. John Stasko's algorithm animation framework as used in TANGO (figure taken from (Stasko, 1989), page 34).

The Algorithm Component adopts an event driven approach in which any events important to the algorithm's semantics are identified by the Algorithm Designer. These events are referred to as "algorithm operators" and are used to model procedure calls mapping the algorithm to the animation, these are

then used to create the animation control file which constitutes the Mapping Component of the framework.

The Animation Component contains the graphical objects that will change location, size and colour throughout the frames of an animation and the operations that control the animation. This approach to generating animations is referred to as the "Path Transition Paradigm" (Stasko, 1990). Four abstract data types are used within the Path Transition Paradigm; images, locations, transitions and paths.

Images are either Primary Images such as lines, rectangles, circles and text, or Composite Images which are collections of primary images with specified geometric relationships. Locations are simply positions within the animation co-ordinate system, identified by an (x,y) co-ordinate pair. The Path is an ordered sequence of (x,y) co-ordinate pairs where each pair designates a relative offset from the previous position, and a relative time component used to control the smoothness of the animation. Finally the Transition component provides the animation with actions to modify the attributes of the image. Three action types are available; move, visibility and fill, these can also be used to form synchronous group transitions. A screen example of a TANGO animation is given in figure 25.

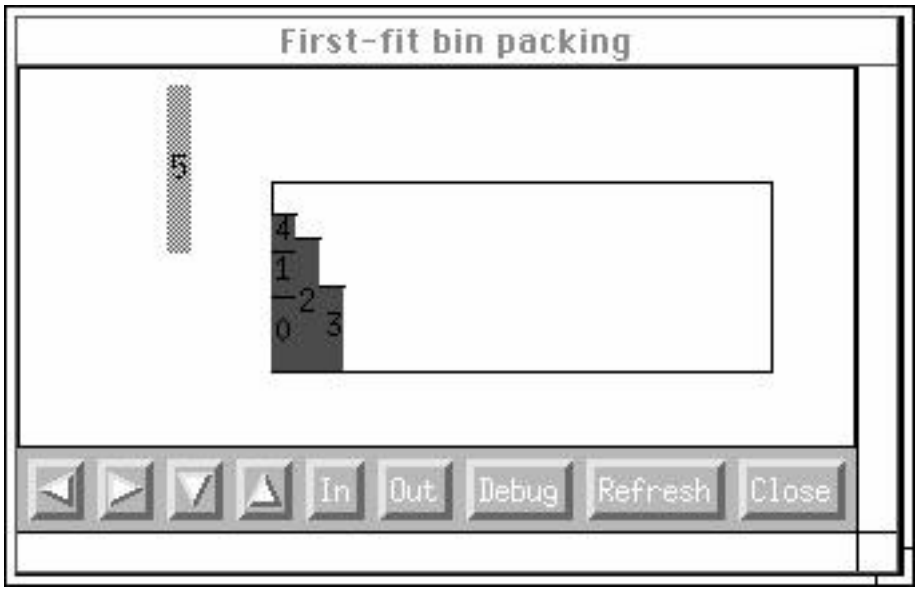

Figure 25. A TANGO animation of a first-fit binpacking algorithm. The elements are inserted into the rectangle and tried against each column position until a large enough free-space is found to house them. The control bar shown at the bottom of the figure allows the user to pan around the view, zoom in and out, switch the debugger on/off, alter the refresh rate, and close the view.

Within TANGO the Algorithm Designer can identify the "algorithm operators" and generate the corresponding animation control file using either

a standard text editor or the Annotation Editor available within Steven Reiss's FIELD programming environment (Stasko, 1989). The advantage of using the Annotation Editor being that the control file can be edited dynamically without having to recompile.

An additional tool available within the TANGO system is DANCE (Demonstration ANimation CrEation), a demonstrational tool for defining the actions to occur in the animation scenes. This promotes ease-of-design and rapid protoyping, hence increasing design experimentation. The graphical editor allows designers to create and manipulate instances of the four data types available within the Path Transition Paradigm to build up sections of the animation referred to as "scenes". Once created a scene can then be automatically converted into animation code. An example screen shot is given in figure 26, showing a typical DANCE scenario.

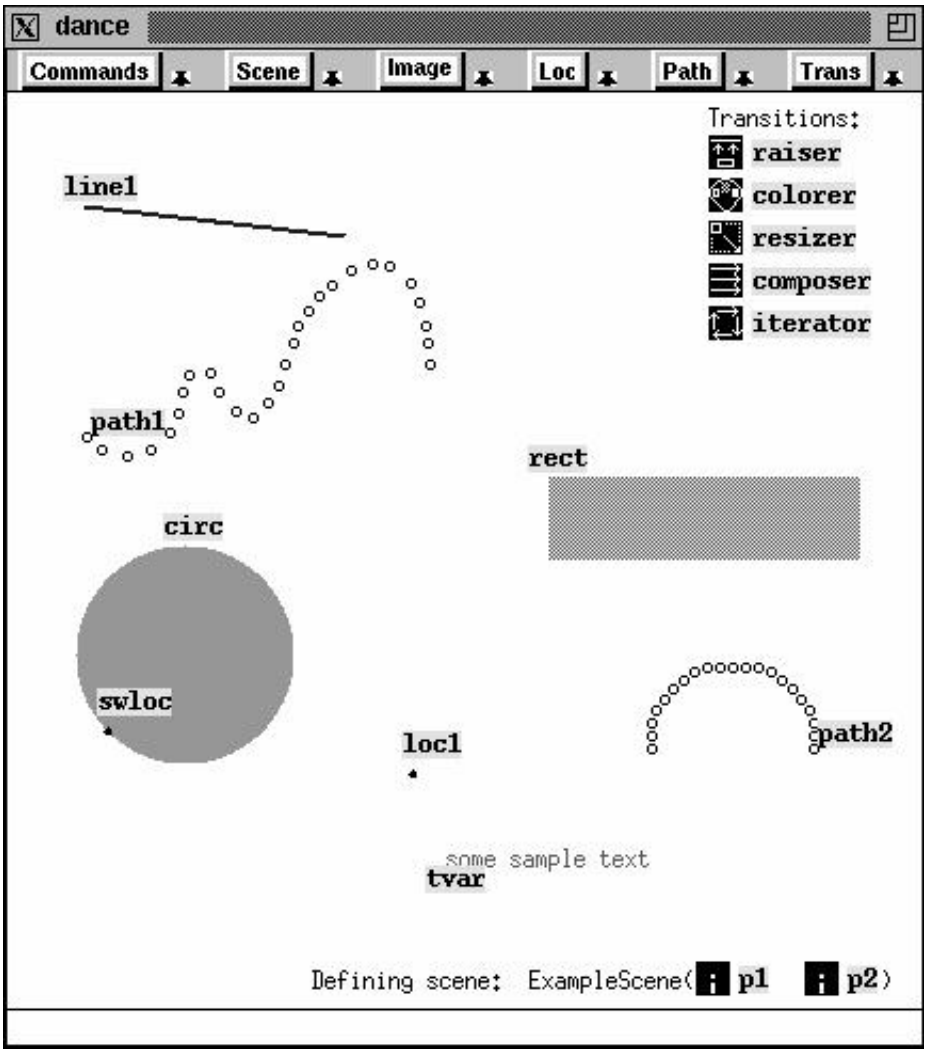

Figure 26. An example of the DANCE algorithm demonstration environment being used to create a new animation scene.

2.4.3. ZEUS

Marc Brown's second algorithm animation system, ZEUS, was designed to provide support for both algorithm animation and multi-view editing. The use of annotations to indicate "interesting events" in an algorithm is still used, however, added features include the use of objects, strong-typing, parallelism and the graphical development of views (Brown, 1991). The use of objects encourages the reuse of code and facilitates the construction of composite views. The introduction of a graphical editor aids the construction of new view components and the adoption of strong-typing provides an opportunity for generating automatic visualizations. A screen shot taken from a ZEUS binpacking animation is given in figure 27.

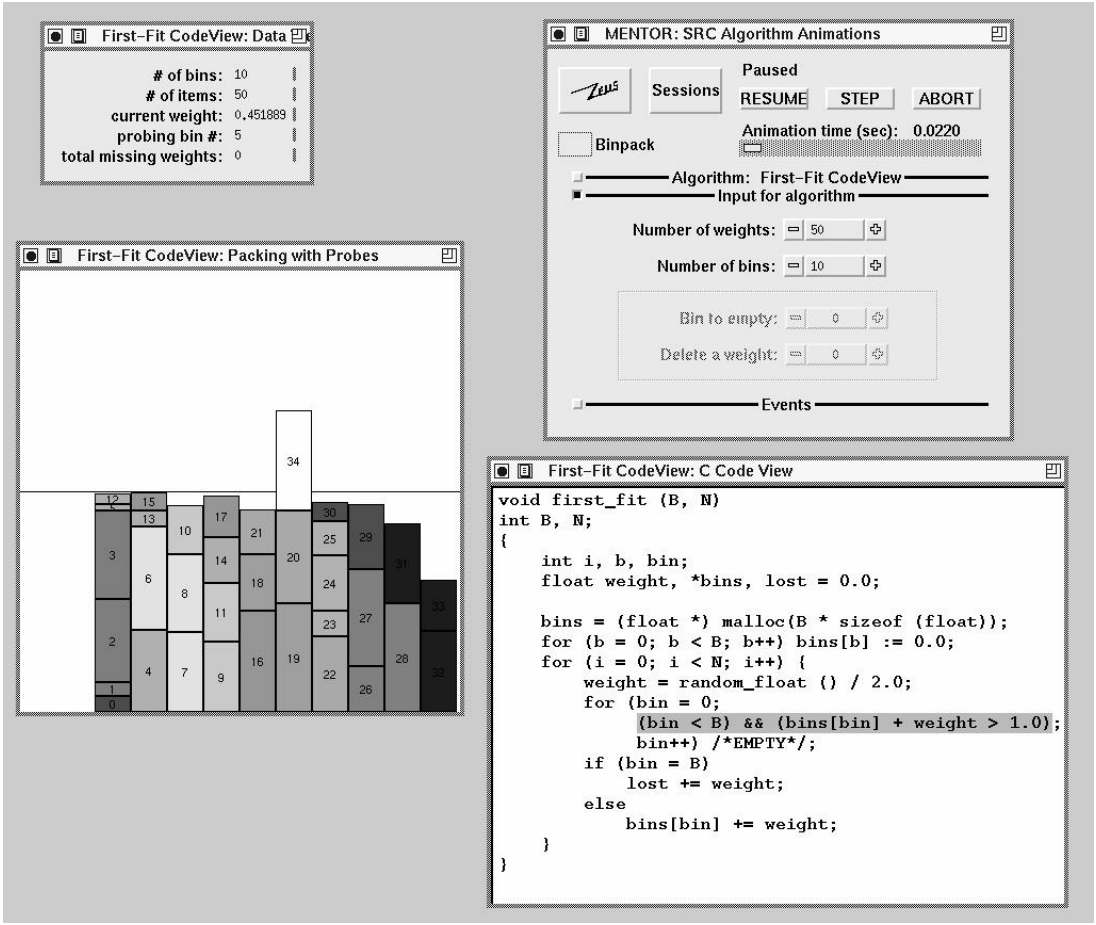

Figure 27. A screen shot taken from a ZEUS binpacking algorithm animation.

Initially the user is presented with a control panel through which they may configure the system and select the appropriate interpreter settings for their task (see top right-hand quarter of figure 27). The configuration facilities allow the user to select which algorithm to run, which views to use, and the data to give the selected algorithm. The interpreter settings allow the starting,

stopping and single stepping of an algorithm, and provide some control over the algorithm's speed of execution. The configuration facilities also permit the user to write a snapshot of the state of the system to a file and restore the system from a previously stored file.

Scripts were not included in the ZEUS system as, unlike BALSA, it was never intended for use as an educational aid, but, as a software engineering tool. Some utility views are generated automatically though from the list of "interesting events". For example a "Transcript View" displays each event as a symbolic-expression as it is generated, the control panel also displays a view of the events as a set of selectable buttons with the appropriate widgets for specifying each parameter. These utility views enable the design and debugging of new views even without the underlying algorithms.

Multi-view editing is supported by the inclusion of the "FormsEdit" editing tool. FormsEdit is a tool developed as part of the FormsVBT User Interface Development Environment (Avrahami, Brooks, & Brown, 1989). FormsVBT was developed within DEC to support a two-view approach to constructing user interfaces from an extensive library of interactor objects. An Interface Designer using FormsVBT can either create an interface by direct programming using the FormsVBT language, or, interactively by using the FormsVBT editor; "FormsEdit" (as is the case with ZEUS).

Forms Edit provides three simultaneous views for the interface designer; a Text View, a Graphical View and a Results View. The Text View incorporates a conventional text editor with which the designer can edit the FormsVBT syntax code, this takes the form of symbolic expressions such as;

```
(Border (PenSize 4) (PenPat "Grey")
        (Border (PenSize 8) (PenPat "White")
                (VBox (Width 200)
                        (HBox Fill "*General Options*" Fill Fill)
                        (HBox "Mail Check: " Fill (Numeric %mailInterval))
                        (HBox "News Check: " Fill (Numeric %newsInterval)))))
```

Figure 28. An example of FormsVBT syntax which produces a *General Options* dialogue box.

The Graphical View not only displays a graphical representation of the interface but also supports graphical editing (see figure 31). Both the text and graphical editors use a shared dataspace for storing the interface design. That dataspace takes the form of a parse tree. Each node in the parse tree holds an element of FormsVBT syntax, the tree is organised in order to reflect the hierarchical structure of the code. Figure 29 illustrates a section of the parse tree representation for the code given in figure 28.

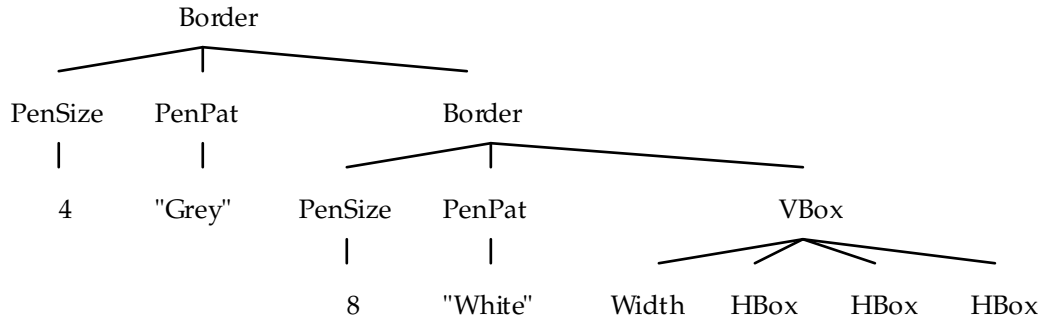

Figure 29. The parse tree representation of the example program given in figure s, only the first four layers of the tree are illustrated here.

The third and final view within FormsEdit is the Results View which produces a fully interactive interface model (see figure 31). Any editing carried out within either of the editors is relayed back to the parse tree and once the tree has been updated the changes are then sent to update all of the views. An architecture diagram showing the structure of the FormsVBT system is given in figure 30.

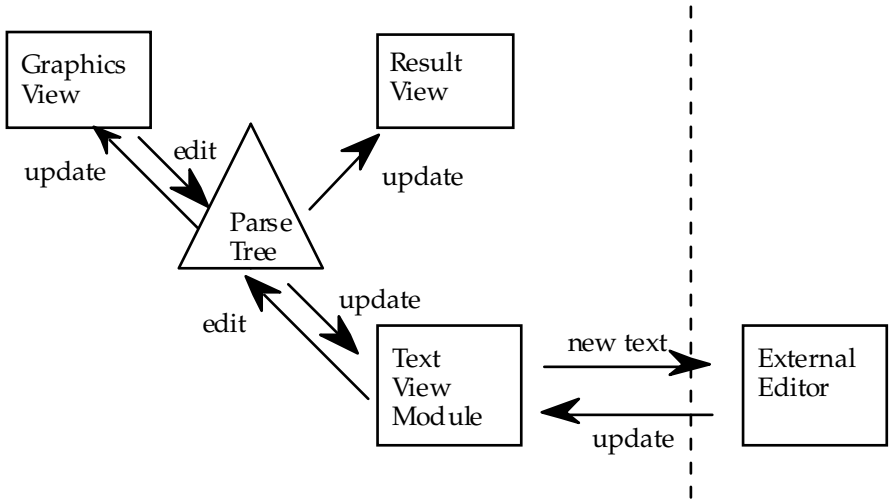

Figure 30. The structure of FormsEdit. The shared parse tree data-space has two-way communication links with the two editing views and a single one-way updating link with the results view. Any external text editor can also be used to edit the Text View Module.

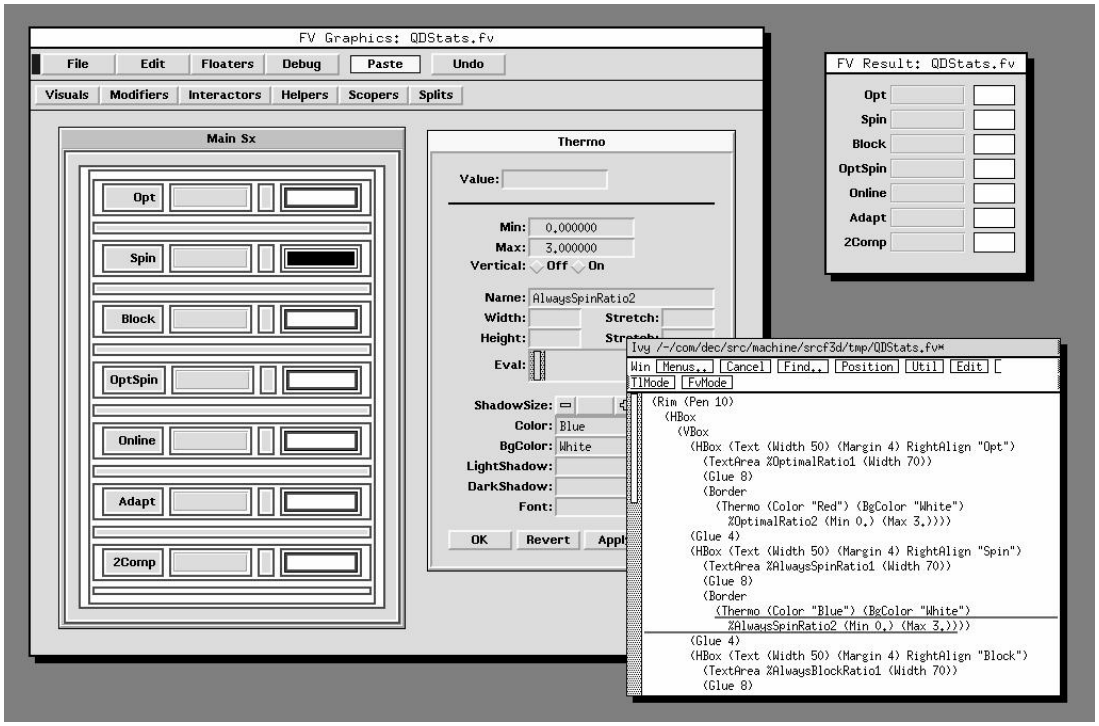

Figure 31. An example screen shot taken from Zeus which illustrates the use of multi-view editing with the FormsEdit editor.

Although Marc Brown's work on the BALSA system did introduce the use of sound and colour to algorithm animation, it is the ZEUS system which further explores the use of these two additional dimensions (Brown & Hershberger, 1992). Brown and Hershberger refer to the use of sound in visualization as "auralization". Sound has been used in a series of sorting algorithm ZEUS animations in order to; reinforce the visuals, convey patterns in the data, replace (some) visuals and for the signalling of exceptional conditions. These animations are based on a musical score metaphor in which the sorting operations are represented by notes and each pass through the algorithm by a bar line. Figure 32 shows one example in which the musical score is presented visually as well as aurally.

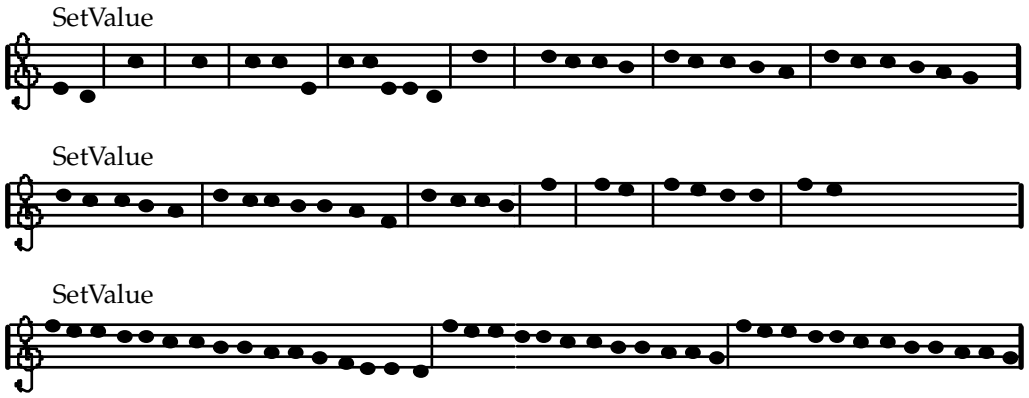

Figure 32 The visual component from a ZEUS "auralization" of an insertion sort algorithm running on a 20 element list. Each bar line indicates the completion of one pass through the array (figure taken from (Brown & Hershberger, 1992), page 60).

The ZEUS system also extends the use of colour from illustrating the state of data structures and highlighting points of interest to emphasising patterns within the data, illustrating an algorithms history, and the tying of views together. The extension of colour usage within this system is based on the principles of graphic design although the design issues involved are beyond the scope of this review it is their application which is the foundation for effective colour usage (see (Tufte, 1990) or (Bertin, 1983)).

2.4.4. ZEUS-3D

In as extension to the ZEUS system Brown and Najork (Brown & Najork, 1993) examined the use of three dimensional visualization in order to; represent additional information geometrically, integrate two normally two-dimensional views, and, for the representation of time within what was originally a two dimensional view.

The addition of a high level object oriented graphics library on top of the 3D graphics extension to X Windows provided the basis for extending ZEUS into three spatial dimensions. The resulting system, ZEUS-3D, provides a platform for presenting 3D perspective visualizations along with support for the rotation of a selected view about any of the three axes (x, y or, z).

Two examples of ZEUS-3D animations are illustrated below, figure 33 shows three tree-based views of a heapsort visualization and figure 34 shows three bar-chart based views of an insertion sort visualization. The three views of each system are intended to communicate the use of the 3D viewing perspective.

Figure 33 is an example of the use of the third spatial dimension to integrate two normally two-dimensional views; the tree-based view shown in the first

view, and the bar-chart view shown in the central view. A tree-based view places the numbers to be sorted on the tree nodes in the same order as their position in the heap then, as the heap becomes sorted, the larger numbers (indicated as brighter nodes) work their way toward the top of the tree. The second way of representing this type of sort algorithm is to use a colour bar-chart, in this case the set of numbers are illustrated as a set of horizontal bars. Each bar's position is used to indicate the respective number's position in the set, the length and colour of each bar is used to indicate the number's magnitude, as the numbers are then sorted the bars are rearranged. In order to integrate these two views Brown and Najork used the z-dimension to indicate the magnitude of each number in the tree-view illustrated in the x-y dimensions.

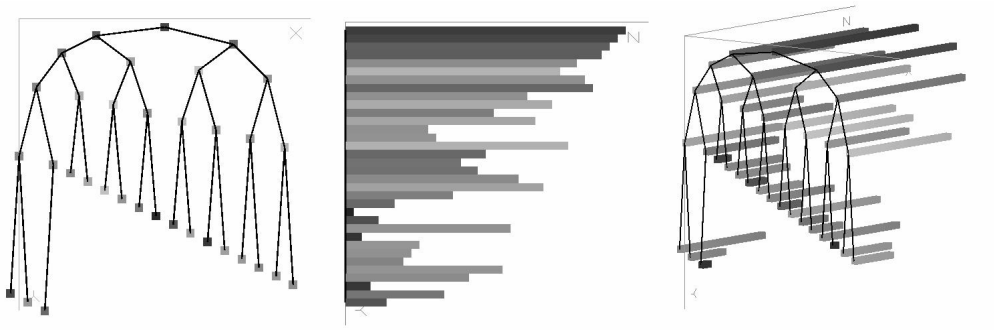

Figure 33. A series of images taken from a ZEUS-3D animation of a heapsort algorithm. A tree representation is used to illustrate the heapsort algorithm with each node's colour and size (i.e. length in the Z dimension) indicating the magnitude of each element in an array.

Figure 34 on the other hand uses the third spatial dimension for the representation of time. The view used in this example is based on a coloured bar chart that indicates each number as a bar, with the number's magnitude being illustrated using both the bar height and colour. As the numbers are sorted the bars are rearranged to form a wedge shaped spectrum. The third spatial dimension (z) is used here to represent time. By reducing the bar chart to a flat strip of what are referred to as paint-chips Brown and Najork place each of the previous passes through the algorithm side by side along the z-axis. Hence it is possible to trace a number's movements throughout the execution of the algorithm by watching how that number's coloured paint-chip changes position as it is plotted along the z-axis.

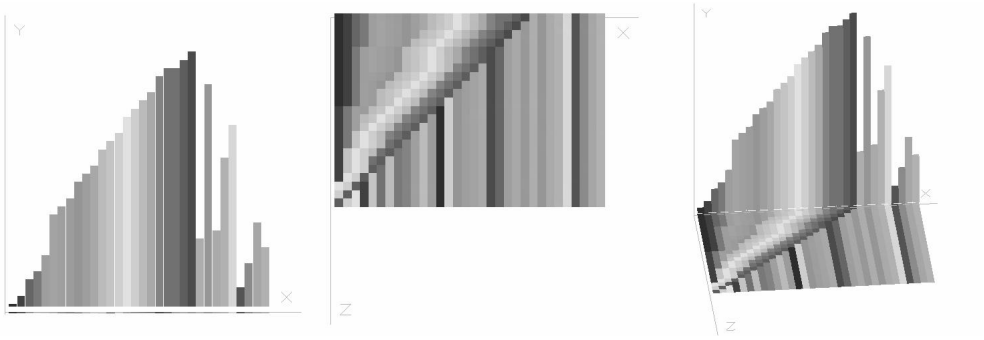

Figure 34. A series of images taken from a ZEUS-3D animation of an insertion sort. This time the elements are represented by adjacent sticks whose colour and size (i.e. height in the Y dimension) are again used to indicate the element's values, however, in this view the Z dimension is used to indicate the algorithms history i.e. previous element positions. In order to provide a clear view of the current array all previous versions are flattened and shown as "paint chips".

These and other examples of the animation work done at DEC can be found on the World Wide Web at http://www.research.digital.com/SRC/zeus/home.html.

2.4.5. PAVANE

The PAVANE visualization system was developed by Gruia-Catalin Roman, Kenneth C. Cox, Donald Wilcox and Jerome Y. Plun at Washington University, St. Louis (Roman, Cox, Wilcox, & Plun, 1992). The PAVANE system was designed in order to help users to understand programs consisting of large numbers of concurrent processes. Roman, Cox et al. argue that the most common approach to visualization, that of using program calls to invoke visualization mechanisms each time changes occur, is unsuitable for concurrent processes. They reason that the parallel nature of the concurrent processes can not be accurately represented in this sequential fashion.

In order to appropriately visualise concurrent processes they propose a declarative approach to visualization. Rather than seeing visualization mechanisms as being called by a control structure each time a change occurs they propose that visualization should be considered as a mapping between computational states and graphical object states rendered by a display device. For this purpose Roman et al. propose the use of formal declarative mappings.

In their system a shared dataspace is used to store all the program state values as content addressable cells. This dataspace is partitioned into three subsets: the *tuple-space* is a finite set of data tuples representing passive data, the *transaction-space* is a set of finite transactions representing the program's actions, and finally the *synchrony relation* is a mechanism for specifying that selected actions are to be executed either synchronously or asynchronously.

Visualizations are specified by defining a mapping from a program state to an image state. Three types of mappings are supported:

> *Simple Mapping* - a functions which given an input space produces an output space.
>
> *History-Sensitive Mapping* - a function which given a primary input space consults the previous version of its output space and produces an output space, enabling the incorporation of historical data.
>
> *Differential Mapping* - a function which given a primary input space consults the previous input space and produces an output space, enabling the detection of any changes.

The mapping between a program state and its corresponding image state is divided into four sub-mappings; a proof mapping, an object mapping, an animation mapping and a frame generation mapping (see figure 35). The proof mapping is a *history-sensitive mapping* which removes any irrelevant details from the program states, the result is then held in the proof space. Object mappings are also *history-sensitive*, they map the proof space states into the object space, this is a 3D world of geometric objects. The animation mapping is a *differential* mapping; it detects visual events, i.e. changes in the object space, and translates them into sequences of image changes in the animation space. The final component is the frame generation mapping, this is a *simple* mapping which interprets the image changes in the animation space and produces a corresponding set of animation frames. The number of frames produced is constrained by the time available between image changes.

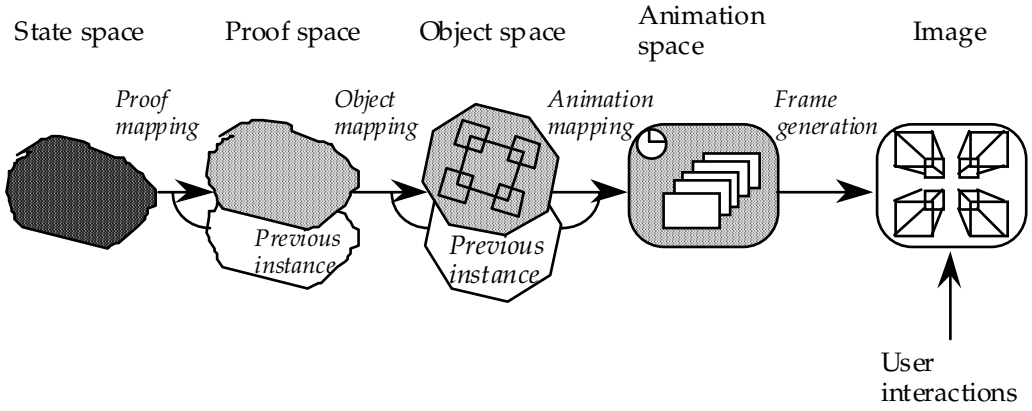

Figure 35. Visualization mapping decomposition in the PAVANE system. This diagram is taken from (Roman, et al., 1992), page 169.

The resulting frame animation system produces concurrent 3D colour animations of an underlying parallel program. An example visualization of a shortest-path algorithm is shown in figure 36. A better impression of this system may be gained by examining its World Wide Web home page at http://swarm.cs.wustl.edu/pavane.html and running a demo.

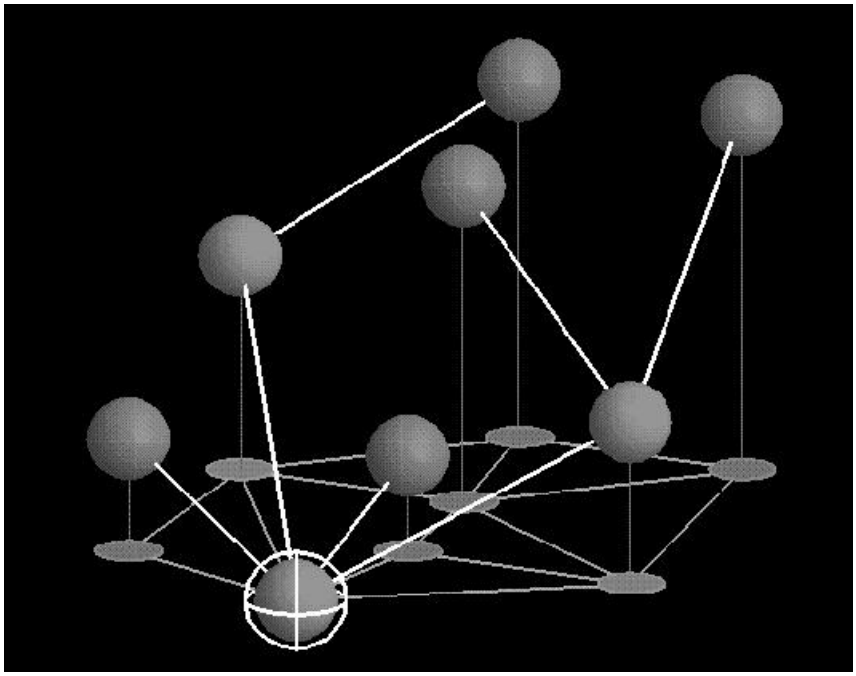

Figure 36. A three-dimensional animation of a shortest-path algorithm showing computed distances and paths from a particular node (marked with circles). The algorithm is operating on the depicted planar graph. Currently-known paths and distances are shown by the lines and spheres above the graph. Colours represent the status of a node in the computation, either scanned (green/light grey) or unscanned (red/dark grey). This diagram was taken from ((Price, et al., 1993), page 233).

2.4.6. PARADE

PARADE is the PARallel program Animation Development Environment, currently under development by John Stasko and his colleagues at the Georgia Institute of Technology (Stasko, 1995). The focus of this work is to use "application-specific" visualization to assist the debugging and correctness checking of parallel programs. Application-specific program views are defined as views that illustrate the program's semantics, its fundamental methodologies and the inherent application domain (Stasko, 1995).

PARADE is made up of three components (figure 37). The first component, the "Parallel Program" component, extracts the necessary program information on which to base the views. The second component, referred to as the "Animation Choreographer", is responsible for the gathering of the program information and its subsequent organisation into a preferred

structure identified by the user (via the Visualization Paradigm). And finally, the third component is the “Visualization Paradigm”. This presents the choreographed program details in a smooth animated form and passes the user’s actions back to the Animation Choreographer.

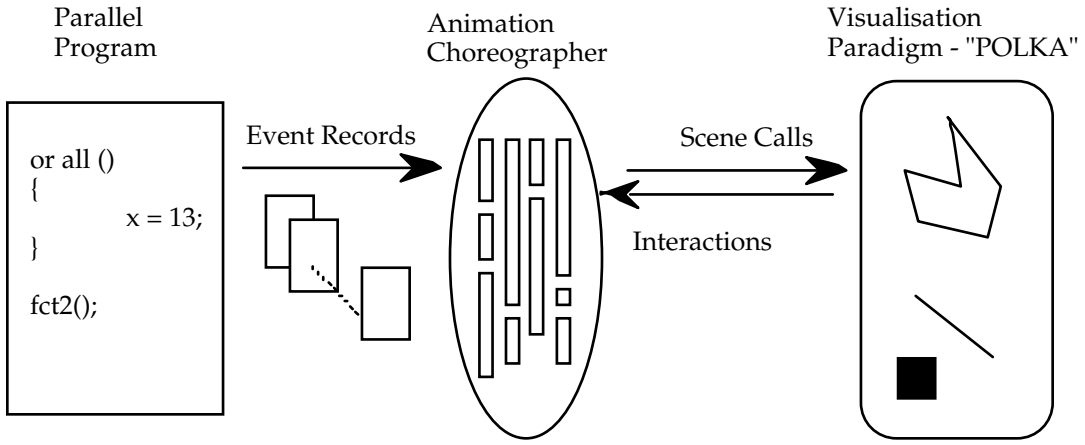

Figure 37. An overview of PARADE highlighting its three major components; the parallel program component extracts the information required for producing the visualizations, the Animation Choreographer gathers the program information from the Parallel Program component and organises it into a preferred format, and finally, the Visualization Paradigm takes the choreographed program details and presents them in an apparently continuous smooth animation to the user. Any user interaction is passed to the Animation Choreographer by the Visualization Paradigm where it is acted upon. This figure is taken from (Stasko & Kraemer, 1992), page 4.

As previously noted the first component extracts the necessary program information to build the visualizations (labelled in the above figure as the Parallel Program component). In order to generate the program information for these visualizations, the PARADE user can utilise any one of three different software instrumentation techniques, namely; via hand annotation of the program code, by overriding the standard communication library, or, through the modification of the resident parallel communication library.

Although annotating the source code with output statements is time consuming and error-prone this is the most general approach of the three, in that the user has a completely free rein over what is shown and the amount of detail presented. The overriding of the standard communication library in PARADE is done by the replacement of the C library called “pthreads”, these include the basic process control and communication calls, with “gthreads”. The gthread library contains a set of simple macros that firstly writes a trace event of each process control or communication call to a trace file before carrying out the associated pthread call. Although this is less general in that it can only produce process control and communication calls it is a lot less time consuming and a lot less Programmer intensive.

The final method for extracting the necessary program information is to actually modify the resident parallel communication library. “Conch” is an experimental heterogeneous network computing system that has been used for this purpose. It contains all the communication primitives used by system with modifications to activate and deactivate the trace facility. Although the replacement of the computer systems parallel communication library may seem rather extreme it results in an almost seamless program-data extraction technique. Once the “Conch” system is in place the user is no longer required to perform any programming tasks the visualization data is automatically recorded during execution. All of the above information extraction techniques time stamp each trace element. This is to ensure that each element will be presented in the program animation at its relevant time position.

The second component of PARADE is the “Animation Choreographer”. This takes the program information in the trace file and maps it to any associated animations. The Animation Choreographer uses the time stamped data of each element to present a directed acyclic graph, the graph’s nodes represent the individual program events indicated by the trace elements.

An “Ordering” menu is used to select the temporal ordering applied to the animation. Currently four options are available the graph can be ordered by: the time stamped data (“Timestamp”), serially by the causal order (“Serial”), as the events occur in global time but with any problems in logical or causal ordering being resolved (“Minimal distortion”), or, as the events would occur to generate maximum concurrency under their causal ordering (“Maximum concurrency”). The Animation Choreographer then allows the user to manipulate the graph until an acceptable ordering is found. A “run” option is available to start the animation using the chosen temporal ordering.

POLKA (Parallel Object-orientated Low Key Animation) is the name given to the third and final component of PARADE. This is the Visualization Paradigm developed specifically for PARADE (Stasko & Kraemer, 1992). POLKA is an object-orientated system for the creation of visualizations and animations which includes both high-level graphical-object, and motion primitives. POLKA is implemented in C++ and is available with either 2D graphical support (on the X Windows system) or 3D graphical support (on top of Silicon Graphics’ GL system).

The POLKA animation methodology is a combination of principles from the Path-Transition Paradigm, (previously developed by Stasko for algorithm animation, see section 2.4.2), and more traditional production 3D animation systems. Figure 38 illustrates the hierarchy of a POLKA animation. An animation is made up of a series of Views with each view being made up of Locations, Actions and AnimObjects.

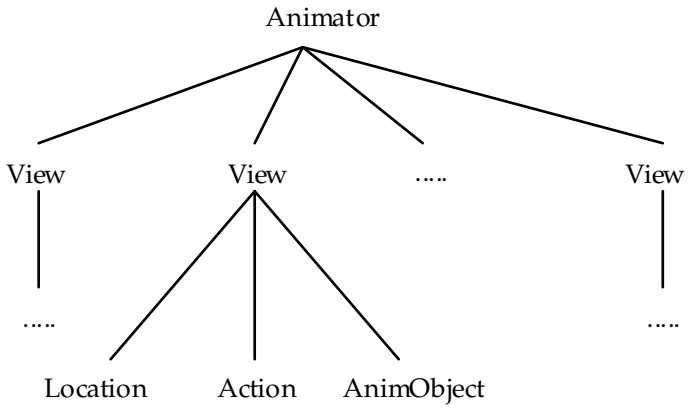

Figure 38. A hierarchy diagram illustrating the structure of a POLKA animation. The Animator module controls the smooth animation of all the Views by ensuring that each animation action is allocated a time-frame. This figure was taken from (Stasko & Kraemer, 1992), page 5.

An AnimObject is the base class for all graphical objects (either 2D or 3D), objects are created by the "Originate" method and deleted by the "Delete" method. Locations in POLKA can be used to reference and remember important positions for later use. They are real-valued (x, y) markers in the View co-ordinate system. Finally the Action class supports the simple movements or changes to be made to the AnimObjects. An Action object has a type such as "MOVE", "COLOR", or "RESIZE" and a list of (x, y) offset pairs defining a two dimensional sequence in the View co-ordinate system.

The most significant feature of the POLKA system is its support for concurrent animation that accurately illustrates parallel program concurrency. This is enforced by the programming of each AnimObject with Actions to occur at particular View frame times. The "Animate" method within the Animator class then checks all of the AnimObjects for each View and ensures that any Actions programmed to occur at the current frame time are executed and the appropriate "Update" and "Draw" methods invoked.

POLKA maintains the simple modification of graphical objects along paths approach cultivated in Stasko's previous Path-transition Paradigm, but adds the capability to program actions into objects at desired animation times. Although POLKA is only one part of the PARADE environment it is currently the most complete component, two screen images illustrating both 2D and 3D visualizations from POLKA are shown in figures 39 and 40, unfortunately no screen images are currently available to illustrate the other two components.

John Stasko maintains a World Wide Web home page for the work done on this and other SV systems by members of the Graphics Visualization & Usability Center at the Georgia Institute of Technology, see http://www.cc.gatech.edu/gvu/softviz/SoftViz.html.

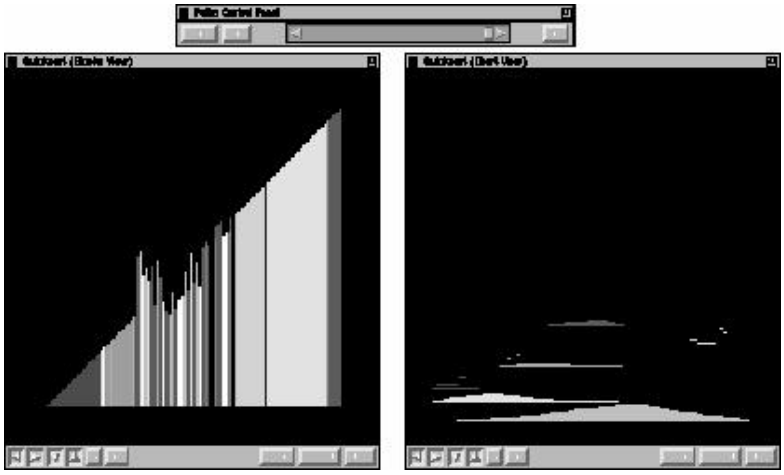

Figure 39. A screen shot taken from POLKA showing a 2D representation of a quicksort algorithm. The view on the left is a "blocks view" showing each element in an array as a block whose height indicates the element's value, and horizontal position indicates its position in the array. The view on the right is a "chart view" in which the horizontal lines are used to represent the swapping of elements, the start and end points of these lines indicate the positions of the elements being swapped. Colour is used in both views to indicate the partitioning of the array. This and other images are available from the web site at the Georgia Institute of Technology (http://www.cc.gatech.edu/gvu/softviz/parviz/polkaanims.html).

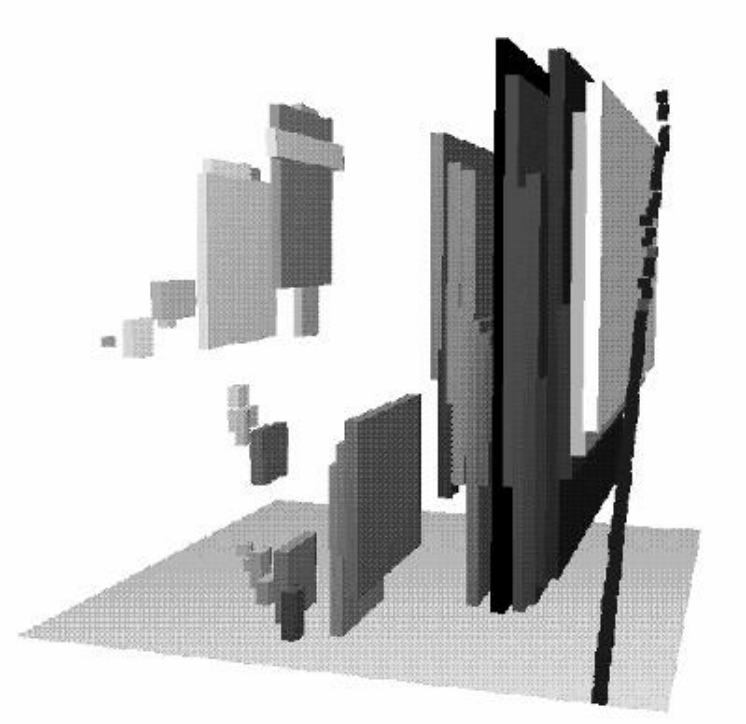

Figure 40. A POLKA 3D representation of a quicksort algorithm in which the smaller blue (dark) boxes to the right represent the values of the elements being sorted, and the multicoloured (greyscale) planes to the left provide an impression of the history of the exchanges in the program. The corner positions of each history plane are defined by the

element's blue block positions along a diagonal plane. This illustration is taken from (Stasko, 1992), page 13.

3. Conclusions

Section one of this review introduced the author's intention, to apply software visualization to support peoples use of genetic algorithms. It also discussed what software visualization means to the author and why it is needed. This first section closed with a description of the structure chosen for the review and a brief overview of the systems included.

The second section then detailed, what are considered by the author, to be the most significant features of each system. In this, the concluding section of this document, an attempt is made to draw from the systems described which features may be of importance for genetic algorithm visualization. The section opens with a brief summary of the systems reviewed along with their associated sources of reference. Some of the more important features are then discussed with respect to GA visualization. Finally the section closes with some examples of possible GA visualization views, illustrating a few of the possible uses of the highlighted software visualization features.

3.1. Summary

The following list provides a summary of the systems reviewed in the previous section. It is hoped that this may form a useful reference guide. Each system is listed with a brief synopsis of its key features, the references used in this review, and any known ftp or world wide web sites where further information may be found.

3.1.1. Development Environments

PECAN - Supports the optional use of structured program command templates.
- Provides concurrent graphical and text based views (a significant contribution at the time).

(Reiss, 1985)

GARDEN - Support for free-form conceptual design and development.
- Object-orientated framework, multiwindow environment and environmental support database.

(Reiss, 1987)

FIELD - Integration framework for a variety of UNIX based tools.

- Support for automatic data structure visualization.

(Reiss, 1990; Reiss & Cruz, 1994)

3.1.2. Program Development

TINKER
- Lisp based programming by demonstration.
- Example demonstrations are generalised to form functions.
- Supports both top-down and bottom-up programming approaches.

(Lieberman, 1981; Lieberman, 1993)

http://lieber.www.media.mit.edu/people/lieber/Lieberary/Tinker/Tinker.html

3.1.3. Program Development

PROVIDE
- Presents graphical views of a program's process states that can be altered via direct manipulation.
- Supports bi-directional stepping through a program's execution.

(Moher, 1988)

Z Step '94
- Supports the bi-directional stepping of code expressions and execution output.
- Presents concurrent displays of the code expressions, their values, and associated graphical output.

(Lieberman & Fry, 1995)

http://lieber.www.media.mit.edu/people/lieber/Lieberary/ZStep/ZStep.html

TPM
- Automatic visualization of Prolog programs.
- Supports bi-directional stepping through a program's execution.
- Provides views at two different levels of visual abstraction, both united by the common visual metaphor of a "goal tree".

(Brayshaw, 1990; Eisenstadt, 1984; Eisenstadt, 1985; Eisenstadt & Brayshaw, 1987)

http://kmi.open.ac.uk/projects/projects.html#2.2

TRI - Rule based program visualization.
- Supports bi-directional stepping through a program's execution.
- Presents a coarse-grained time specific view (i.e. a rule graph) along with several fine-grained rule specific views.

(Domingue & Eisenstadt, 1989)

VITAL - The VITAL Workbench provides methodological and software support for the design, development and validation of knowledge based systems.
- Uses the Viz visualization framework and software library to produce code, design and domain based views.

(Domingue, et al., 1994; Domingue, et al., 1992; Domingue, 1995; Domingue, et al., 1993)

http://kmi.open.ac.uk/~john/vital/vital.html
http://kmi.open.ac.uk/~john/viz/viz.html

3.1.4. Algorithm Visualization

BALSA - Allows users to interact with the dynamically changing graphical representations of their programs' execution.
- Intended for use mainly as an educational aid for the teaching of computer algorithms.

(Brown, 1987; Brown, 1988; Brown & Sedgewick, 1985)

ftp://ftp.dec.com/pub/DEC/macbalsa-demo.sit.hqx.Z
ftp://ftp.dec.com/pub/DEC/macbalsa-source.sit.hqx.Z
ftp://ftp.dec.com/pub/DEC/macbalsa-userguide.ps.Z

TANGO - Uses the Path Transition Paradigm to produce smooth frame animations where an image is moved by small increments along a path in a succession of animation frames.

(Stasko, 1989; Stasko, 1990)

ZEUS - Multiview editing provided via the FormsEdit tool which supplies a text editor, graphical editor and a resulting program output viewer.
- The Zeus system examined the use of colour and sound as additional communication channels capable of delivering information not contained in the graphical or text based views.

(Avrahami, et al., 1989; Brown, 1991; Brown & Hershberger, 1992)

http://www.research.digital.com/SRC/zeus/home.html

ZEUS-3D - Basically ZEUS plus 3D perspective. Here the third spatial dimension is recommended as an aid to illustrating additional information, the previous states (i.e. history) of a two dimensional view, or, as an integration mechanism for two related two dimensional views.

(Brown & Najork, 1993)

http://www.research.digital.com/SRC/zeus/home.html

PAVANE - Proposes the declarative approach to parallel program visualization. Where the views of a program are created through a series of formal declarative mappings.
- Produces two or three dimensional, colour rendered, animations.

(Roman, et al., 1992)

http://swarm.cs.wustl.edu/pavane.html

PARADE - Supports the production of application-specific parallel program animations.
- Extends the Path-Transition-Paradigm to include the occurrence of concurrent operations.
- Can produce either two or three dimensional colour animations.

(Stasko, 1995; Stasko & Kraemer, 1992)

http://www.cc.gatech.edu/gvu/softviz/SoftViz.html
http://www.cc.gatech.edu/gvu/softviz/parviz/polkaanims.html

3.2 Discussion

In the course of the above review it has became increasingly obvious that software visualization is much more than pretty-printing or fancy pictures. It is the principled application of techniques adopted from a variety of disciplines including graphic design, psychology, and animation. Perhaps one of the most influential factors for software visualization has been the rapid increase in computing power over the past two decades. This has nurtured the use of 2D and 3D graphical views, colour and sound, making software visualization a practical reality.

The use of multiple, two and three dimensional colour views are common place in most of the currently available systems. The effective application of these features however can not be left to chance, but must be based on some idea of effective representation. Several of the systems' authors cite the seminal works of Edward Tufte and Jacques Bertin who have written extensively on the principled application of graphic design techniques for representing information (Bertin, 1983; Bertin, 1987; Tufte, 1990).

However as well as bring out the common place approaches to viewing information, the systems reviewed have illustrated many more novel approaches. The work done by Brown and Hershberger on the effective use of sound indicates the extent to which sound may be used (Brown & Hershberger, 1992). Although no evaluation work was done on their "auralizations", the use of sound to reinforce the visual displays and signal exceptional conditions is now a standard feature of the majority of direct manipulation user interfaces. Whether audio can in fact be used to replace some of the visual images is unconfirmed. This would involve the user attending to both visual and audio messages concurrently. As these would be communicating different information this would effectively double the user's cognitive load.

The use of bi-directional control mechanisms to step through a programs execution is another interesting approach for supporting the user. The Open University's TRI, TPM and Viz systems all support bi-directional stepping as does Steven Moher's PROVIDE and Henry Lieberman's ZStep'94 systems. This feature could be used within genetic algorithm visualization to support the users examination of an algorithm's evolution. Providing the user can view the data produced after each pass through the algorithm (i.e. each "generation"), a bi-directional control mechanism could be introduced to navigate through the algorithms evolution. Furthermore, at any backtracked position in an algorithms evolutionary data set, the user could either step forward to the next generation in the stored data set, or, restart the algorithm again and let it re-evolve, quite possibly to a better solution.

This idea of evolutionary control through restarting an algorithm is even more intriguing given that if the user were allowed to control the algorithms parameters (as in the BALSA system) they could actually step back and forth re-starting an algorithm with differing parameters. Given time this process could be used to find the optimal set of problems for the attempted problem.

The use of a pause and directional control mechanism also introduces the idea of version control. As the visualization system would be storing the algorithms output data set, a simple extension of this would be to maintain different versions of the GA's evolution for different parameter settings. These could then be used either to illustrate the influence of the parameters over an algorithm, or to guide future efforts on similar problems.

The effective use of multiple views is a non trivial problem. In systems like TRI and TPM there are specific fine-grained and coarse-grained views intended to support different components of the user's task. The FIELD environment supports the integration of various Unix based programming tools, and the VITAL workbench supports the use of task-specific tool sets. All of these are tailored to the needs of the user and the requirements they have for each particular task. Similarly a GA visualization system must consider the tasks involved in applying GAs and support these accordingly.

For example, a GA visualization system to be used as an educational aid to teach how GAs work, the views presented should illustrate the operations of the algorithm. Hence algorithm visualization should be used to visually present the generation, evaluation, selection and reproduction of the representative strings (i.e. "chromosomes"). This could easily involve a roulette wheel image as used by (Goldberg, 1989) to illustrate the selection procedure, or a cut and paste image to illustrate the crossover genetic operator.

In order to illustrate an algorithm's execution the views should reflect the data sets changes over successive generations i.e. it should present data visualizations. Finally for the process of programming GAs, like any programming task, the effective application of code visualization techniques may best support the user's needs.

Is the best solution therefore to develop a task specific visualization system? It is the author's opinion that it is not. Like TRI, TPM, FIELD and the VITAL Workbench it is the views that are unique to the task, not the system. Provided a consistent framework can be made available within which a library of views can be used, there is no foreseeable reason why a visualization system cannot support all these tasks. Furthermore, providing the user interface is flexible enough to accommodate a wide range of experience in its users, the one visualization system could be used by the

same user as they become more familiar with GAs and progress from learning about GAs, to developing and applying GAs of their own.

The idea of a software library containing different views is used in several of the above systems. A library of possible views gives the user the freedom to choose their preferred combinations of views. This includes the freedom to create duplicate views and uninformative views. However, the use of default combinations of views suited for particular tasks enables the provision of recommended, complimentary view combinations (like those presented in the BALSA demo package).

A library that cannot be extended however, is limited. As new views are devised some simple means of extending the library becomes necessary. This may be done using a tailored high level command language, as is the case in Viz, TANGO and POLKA, or through the use of a graphical editor, as is the case with the DANCE animation tool, and the FormsEdit editor used in ZEUS.

As has been discussed in this section the key features which seem most appropriate for the development of genetic algorithm visualizations are as follows:

- The principled use of 2D, 3D and colour views.
- The use of sound to illustrate additional information, indicate exceptional conditions and reinforce the visuals.
- The provision of a bi-directional control mechanism for stepping through an algorithms evolution, with the added possibility of recording multiple evolutions.
- The provision of an extendible view library with recommended task orientated view combinations.

The following final sub-section, shows how some of these features could be presented in a series of design models.

3.3. Design ideas

This section examines some design ideas for the inclusion of the key features highlighted in the previous sub-section.

3.3.1. Principled 2D, 3D and colour views

As previously noted the design of the individual views is a crucial factor in how useful a visualization system will be (section 3.2). It is therefore, essential that some effort is made to ensure the views used are as informative as possible. Design principles, such as those proposed by Edward Tufte and

Jacques Bertin, provide some guidance for this (Bertin, 1983; Bertin, 1987; Tufte, 1990). A complete description of such guidelines is beyond the scope of this review, therefore the reader is merely recommended these texts with the hope that they may be of help.

3.3.2. The use of sound

Sound could be used within a GA visualization system to indicate the algorithm's operations. Unless the user is specifically interested in the individual chromosomes' selection, reproduction and evaluation operations they may not be visually presented. However, sound could be used independently of the visual images to communicate these operations. This may be useful in illustrating the execution speed, the occurrence of crossovers, mutations, or other genetic operations, and the discovery of an acceptable solution.

An possible method for this would be to use a clock metaphor. An hour-chime could be used to indicate each pass through the algorithm, and a tick could be used to indicate each operation. As there are different types of operations within each algorithm (i.e. selection, crossover, mutation, etc.) a variation in pitch could be used to indicate each operation's type, these could be further categorised with the use of distinctive chimes to indicate the beginning of each algorithm component. An alarm could be used to indicate the discovery of an optimal, or acceptable, solution. A simple example illustrating these ideas is given in table 4 below.

Algorithm Components	*Audio signal*
1. Start	
2. Initial Population Creation	Chime once
2.1. generate random chromosome	tick
2.2. evaluate chromosome	tock
3. Population Reproduction	Chime twice
3.1. select two chromosomes	tick
3.2. crossover chromosomes	tack
3.3. evaluate chromosomes	tock
4. Is the optimal solution reached?	
No - goto 3.	Hour-chime
Yes - Stop	Alarm buzz

Table 4. An example clock based audio visualization suitable for indicating a simple genetic algorithm's internal operations.

The use of sound to indicate exceptional conditions and reinforce visual images should also be considered as additional techniques for improving a visualization system. No specific designs for reinforcing sounds are given here as their design is dependent on the visual images presented. One example of sound as an indicator of exceptional conditions has already been

given in the clock based audio visualization. There sound was used to indicate the discovery of an optimal solution. Sound benefits from the fact that it can be used as an attention grabbing form of communication and can therefore be used to draw attention to any important, or exceptional, events.

3.3.3. A bi-directional control mechanism

A genetic algorithm's evolution is a strongly guided but essentially random-based search. The initial population, the position of crossover points between mating chromosomes and the occurrence of mutation, are all random events. If two identical GAs were executed side by side there is no reason why they couldn't evolve completely different solutions.

Therefore, in the case of a controller used to step from one generation to the next the difference between stepping forward one generation in an algorithm's trace data-set and executing the algorithm to generate the next generation must be made explicit. The following example design illustrates the use of separate control buttons for the execution (labelled "play") and the stepping (labelled "step") of a GA (see figure 41).

Figure 41. A Bi-directional control panel suitable for use within a genetic algorithm visualization system.

The use of a single step stepper to navigate through an execution trace can be frustrating when the user wishes to step to the start, end or any specific position in between. Therefore, in order to step through a program at an appropriate rate additional buttons can be made available to rewind to the start, fast-forward to the end, or step N positions forward or backward (where N is a positive integer). A second example illustrating this is given in figure 42.

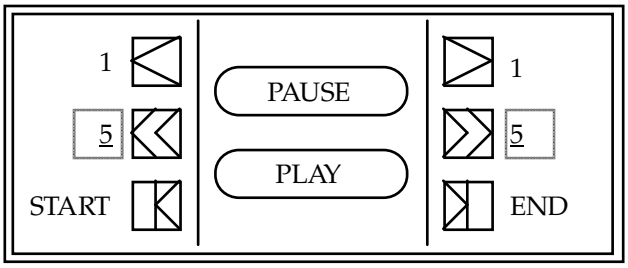

Figure 42. A bi-directional control panel capable of multiple-step stepping. The user can edit the number of steps taken in the outer-middle two stepping buttons, which are currently set to 5

(generations). The outer-bottom two stepping buttons enable the user to go directly to the first generation i.e. the start, or the latest generation i.e. the end.

These are two sample designs illustrating how a control panel may be used for navigation. Further extensions can be added to permit version control over stored execution traces. An example addition to the control panel shown in figure 42 is given in figure 43, below. Here the user can either select their data set from a previously loaded data library, or, record new data using their own algorithm.

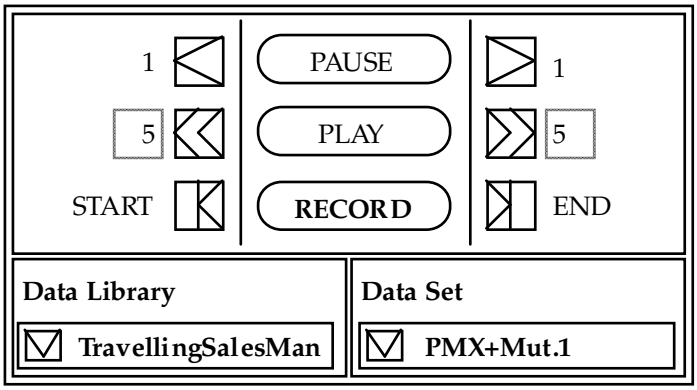

Figure 43. A bi-directional control panel and data selection dialogue. The user's data library and data set are selected from a pull-down menu, new data library or data set labels can be typed directly into the Data Library and Data Set boxes, respectively.

3.3.4 An extendible view library

A view library would permit the user to select views suited to their particular task. The use of default task orientated view combinations may be one method of encouraging inexperienced users to use complimentary views, then when the user gains more experience they may create their own combinations based on their experiences. A design for a view selection and parameter allocation dialogue is given in figure 44.

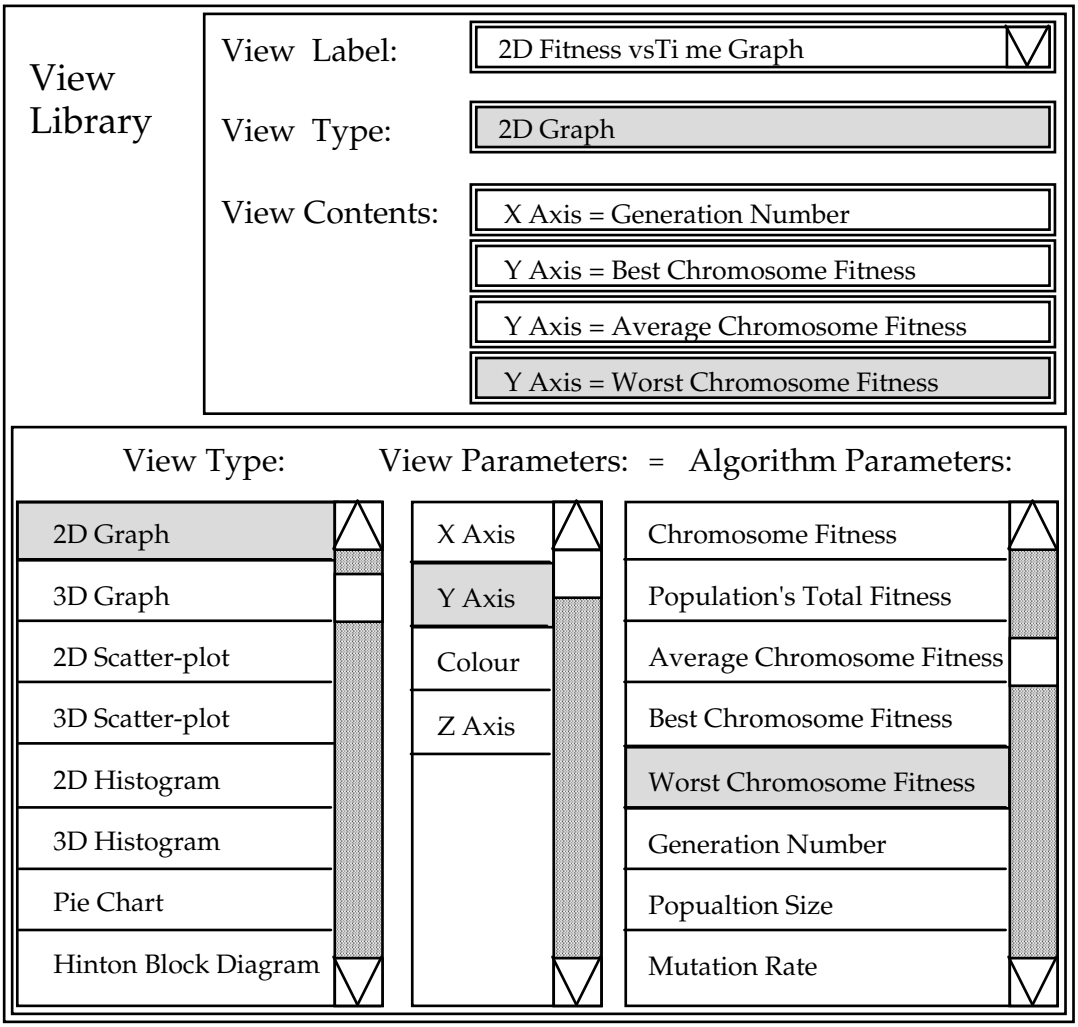

Figure 44. A view selection and parameter allocation dialogue suitable for use within a GA visualization system. The user can either select a view from the library using the pull-down menu on the View Label box, or type in a new label and then define it using the selection menus in the lower half of the dialogue. Each views type and contents are displayed in the upper half of the dialogue, below the View Label.

As noted in the previous sub-section (3.2) a library should be extendible in order to allow the creation and application of new views. Selecting an appropriate method for view definition is a difficult task, visual languages can suffer from being too specific and as a result prevent the user from freely expressing their ideas, conversely command languages can be too much bother for non-programmers to use freely.

A compromise to this problem may be to support both graphical and text editing (as is the case in ZEUS), or to use a visualization specific high-level language that reduces the amount of programming effort required (as in the case in Viz, TANGO, and PAVANE). Providing a standard protocol is used for the definition of each new view's parameters, new views could be incorporated into the view library by simple cut and paste, or drag and drop operations.

References

Avrahami, G., Brooks, K. P., & Brown, M. H. (1989). A Two-View Approach to Constructing User Interfaces. Computer Graphics, 23(3), 137 - 146.

Bertin, J. (1983). Semiology of Graphics (Berg, W.J., Trans.). Wisconsin: The University of Winsconsin Press.

Bertin, J. (1987). The Graphic Sign System (A Semiological Approach to Graphics). In Graphics and Graphic Information Processing (pp. 176 - 232). Berlin: Walter de Gruyter & Co.

Brayshaw, M. (1990). Visual Models of PARLOG Execution (Technical Report No. 64). The Open University.

Brown, M. H. (1987). Algorithm Animation. Cambridge, MA: MIT Press.

Brown, M. H. (1988, May 1988). Exploring Algorithms Using Balsa-II. Computer, p. 14 - 36.

Brown, M. H. (1991). Zeus: A System for Algorithm Animation and Multi-View Editing. In IEEE Workshop on Visual Languages, (pp. 4 - 9). Kobe, Japan.

Brown, M. H., & Hershberger, J. (1992). Color and Sound in Algorithm Animation. Computer, December 1992, p. 52 - 63.

Brown, M. H., & Najork, M. A. (1993). Animating Algorithms Using 3D Interactive Graphics (Technical Report No. 110a). Digital SRC.

Brown, M. H., & Sedgewick, R. (1985). Techniques for Algorithm Animation. IEEE Software, January 1985, p. 28 - 39.

Domingue, J., & Eisenstadt, M. (1989). A New Metaphor for the Graphical Explanation of Forward-Chainig Rule Execution (Technical Report No. 52). The Open University.

Domingue, J., Eisenstadt, M., & Price, B. (1994). The VITAL Visualisation Tool Report (Technical Report No. T-3.4.2). The Open University.

Domingue, J., Price, B. A., & Eisenstadt, M. (1992). Viz: A Framework for Describing and Implementing Software Visualisation Systems. (Internal Report), The Open University, Human Cognition Research Laboratory.

Domingue, J. B. (1995). Software Visualization Based KBS Validation (Technical Report No. KMI-TR-8). The Knowledge Media Institute, The Open University.

Domingue, J. B., Motta, E., & Watt, S. (1993). The Emerging VITAL Workbench. In N. Aussenac, G. Boy, B. Gaines, M. Linster, J. G. Ganascia, & Y. Kodratoff (Ed.), Knowledge Acquisition for Knowledge-Based Systems 7th European Workshop. EKAW '93., (pp. 320-339). Toulouse and Caylus, France.: Springer-Verlag.

Eisenstadt, M. (1984). A Powerful Prolog Trace Package. In European Conference on Artificial Intelligence. Pisa, Italy.

Eisenstadt, M. (1985). Tracing and Debugging Prolog Programs by Retrospective Zooming (Technical Report No. 17). The Open University, Human Cognition Research Laboratory.

Eisenstadt, M., & Brayshaw, M. (1987). The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. (Technical Report No. 21A). The Open University, Human Cognition Research Laboratory.

Goldberg, D. E. (1989). Genetic Algorithms in Search Optimization and Machine Learning. Addison Wesley.

Lieberman, H. (1981). Tinker: Example-Based Programming for Artificial Intelligence. In International Joint Conference on Aritificial Intelligence, 2 (pp. 1060). Vancouver, B.C.

Lieberman, H. (1993). Tinker: A Programming by Demonstration System for Beginning Programmers. In A. Cypher (Eds.), Watch What I Do: Programming by Demonstration (pp. 48 - 64). Cambridge, MA: MIT Press.

Lieberman, H., & Fry, C. (1995). Bridging the gulf between code and behaviour in programming. In CHI 95.

Moher, T. G. (1988). PROVIDE: A Process Visualisation and Debugging Environment. IEEE Transactions on Sofware Engineering, 14(6), 849 - 857.

Motta, E., Rajan, T., & Eisenstadt, M. (1989). Knowledge Acquistion as a Process of Model Refinement (Technical Report No. 32). Human Cognition Research Laboratory, The Open University.

Price, B. A., Baecker, R. M., & Small, I. S. (1993). A Principled Taxonomy of Software Visualisation. Journal of Visual Languages and Computing, 4, 211 - 266.

Reiss, S. P. (1985). PECAN: Program Development Systems that Support Multiple Views. IEEE Transactions on Software Engineering, SE-11(3), 276 - 285.

Reiss, S. P. (1987). Working in the Garden Environment for Conceptual Programming. IEEE Software, November 1987, p. 16 - 27.

Reiss, S. P. (1990). Interacting with the FIELD environment. Software-Practice and Experience, 20(S1), 89 - 115.

Reiss, S. P., & Cruz, I. F. (1994). Practical Software Visualisation. In M. Brown, J. Domingue, B. Price, & J. Stasko (Ed.), Software Visualisation Workshop - SIGCHI 94.

Roman, G. C., Cox, K. C., Wilcox, D., & Plun, J. Y. (1992). Pavane: a system for declarative visualisation of concurrent computations. Journal of Visual Languages and Computing, 3, 161 - 193.

Stasko, J. T. (1989) TANGO: A Framework and System for Algorithm Animation. PhD Dissertation, Brown University.

Stasko, J. T. (1990). The Path-transition Paradigm: a Practical Methodology for Adding Animation to Program Interfaces. Journal of Visual Languages and Computing, 1(3), 213 - 236.

Stasko, J. T. (1992). Three Dimensional Computation Visualisation (Technical Report No. GIT-GVU-92-20). Georgia Institute of Technology.

Stasko, J. T. (1995). The PARADE Environment for Visualising Parallel Program Executions: A Progress Report (Technical Report No. GIT-GVU-95-03). Georgia Institute of Technology, Graphics Visualisation and Usability Centre.

Stasko, J. T., & Kraemer, E. (1992). A Methodology for Building Application-Specific Visualisations of Parallel Programs (Technical Report No. GIT-GVU-

92-10). Georgia Institute of Technology, Graphics Visualisation and Usability Center.

Tufte, E. R. (1990). Envisioning Information. Cheshire, Connecticut: Graphics Press.