Visual Techniques for Parallel Processing

Preliminary Written Ph.D. Comprehensive Examination

for

James Arthur Kohl

July 25, 1991

Ph.D. Committee:

Thomas L. Casavant - Chair

Jon Kuhl

Michael Lyu

John P. Robinson

Teodor Rus

Outline

I. Introduction & Problem Statement

This Comprehensive Examination consists of an accumulation and analysis of research on the use of visualization in computing systems over the past decade, as well as recent efforts specifically in the area of software development for parallel processing. The goal of the examination is to determine the relationships among the references located, and their cumulative effect in directing the course of future research in the field of visualization.

The examination includes a creative portion in which the various uses and approaches for visualization are to be classified via a taxonomical system. This classification will identify the central issues which differentiate the visualization environments for developing parallel software. In addition, a quantitative assessment of these environments will be constructed which presents a more concrete evaluation and categorization technique.

The following sections constitute my solution to the examination in the following manner. Section II describes briefly the broad literature survey, including the approach to the search and a summary of the literature references.

Section III presents a detailed analysis and summary of recent work on selected visualization tools and environments for parallel software development. These tools and environments will be analyzed with respect to their range of utility or capabilities, as well as cost-effectiveness or usefulness.

Section IV combines the overall view, of the broad and focused analyses, to construct a taxonomical system for categorizing and evaluating visualization tools. A broad taxonomy of the use of visualization in general will be provided, along with two special detailed taxonomies for parallel software tools. In addition, several quantitative metrics will be described which concisely compare and classify visualization tools for parallel software development.

Section V will draw conclusions about the current state of the field, based on the previous sections, and will further explore possibilities for promising future work. Section VI lists the collection of relevant references compiled in completing this comprehensive examination.

II. Broad Literature Survey

The broad literature survey was approached in two distinct phases. The first phase involved a very broad search of a wide range of recent literature from various science and engineering fields. This portion of the search intended to provide a perspective on where visualization for parallel software development fits into the world of visualization in general.

The second phase of the search was more refined, yet reached further back into the past ten years. This phase was restricted specifically to tools for visualization in computer software development, with an emphasis on parallel software. This research was based partially on correspondence with colleagues from various research projects, and in some cases included experimentation with the actual tool environments. From the collection of references and contacts, the more focused examination of recent efforts was selected. This examination will be discussed in detail in Section III.

The research found in the broad survey indicates that visualization is becoming more widespread across all fields, and has been growing in popularity for the past few years [Jon90]. Many fields have adopted computers as assistants in manipulating information and solving problems, and visualization techniques provide natural organization and representation for these purposes. The great diversity in the types of information and problems has produced appropriately diverse visualization approaches and uses. Far beyond the commonplace uses of computer graphics for elaborate computer games, visualization has been applied to many fields of the science and engineering community, as well as to art, education, and many business and professional fields.

Computer graphics has been introduced to the world of art, both in a pure aesthetic form and in advertising. Education has used visualization to manage and deliver instruction in novel new ways. Some professional fields such as architecture and home remodeling have used visualization as a cost-effective time-saving prototyping tool. The business community has organized office information and assisted office management, controlled and designed manufacturing and production systems, and organized legal functions using visualization.

The use of visualization in technical fields comes as no surprise, and may be seen prevalently in many scientific applications. There are many uses in mathematics for displaying masses of numerical data values, and for representing complex theoretical functions and surfaces. The medical field uses visualization to assist in presenting electronically generated images of human and animal anatomies for analysis and diagnosis of conditions as well as surgery. Fields such as geography and seismology use visualization to characterize the surface of the earth for mapping features and properties. Environmental scientists and meteorologists use visualization to analyze and understand the weather and the environmental condition of the world. Many of the physical sciences use scientific visualization to assist in analyzing and understanding the actions of chemicals, molecules, biological and physical structures, and genetics. In computer science, visualization is used extensively in interaction with databases and artificial intelligence, notably with knowledge acquisition in expert systems and relational programming languages.

In the engineering fields, visualization is also used effectively in modeling, designing, and representing systems and information. In Computer-Aided Engineering (CAE), many engineering fields use computers to analyze and process complex types of information for typical engineering applications such as electromagnetics, communication and signal processing, and robotics.

One of the oldest visualization categories is Computer-Aided Design (CAD). CAD has been used for years in the design and modeling of mechanical systems from automobiles to supersonic jets, and has been extensively applied to the design and manufacture of VLSI integrated circuits for electronics and engineering of computer hardware systems.

A relatively new category in visualization is that of Virtual Reality. In this category, graphics are used to visualize a fabricated or fictitious multi-dimensional world to model and simulate desired behavior or functionality. Virtual Reality systems are used to simulate worlds or environments with more than three dimensions, to simulate the manipulation and sculpting of three-dimensional volumes and objects, and to animate the simulation of mechanical motion or physical interaction. Typically, a Virtual Reality system will utilize tactile stimulation with visualization to create a realistic interface between the user and the environment, as the name implies.

Another important category of engineering visualization includes many fields which pertain directly to the design, programming and use of computers themselves, or software engineering. In a broad sense, this category may be considered Computer-Aided Software Engineering, or CASE, although this particular use of the term may be slightly more general than the typical context. In this perspective, CASE applications include everything from software creation and analysis to documentation, maintenance, configuration management, as well as computation modeling, interface development, and design team or groupware management.

It is in this last category of software engineering that the area of interest and the central focus of this examination is found - the creation and analysis of parallel software. This area uses visualization from two distinct viewpoints, *visual programming* and *program visualization* [Mye88]. Visual programming refers to any system or environment which in some way uses a graphical means other than pure one-dimensional text to specify or create computer programs. This specification may be of the control or data flow, or may represent some hybrid approach based on visual semantics or graphics.

Program visualization, on the other hand, takes an implemented piece of software and visualizes some aspect of its execution or its interaction with the system. Program visualization may be static or dynamic in nature, and may represent information in varying levels of graphical sophistication from simple graphs and charts to elaborate animations.

These two visualization approaches are more essential in the programming and use of parallel or distributed computer systems than in sequential systems. This is primarily due to the added complexity of multiple threads of control interacting during execution. The massive amounts of information necessary for analyzing concurrent software require the organizational features of visual representation, as well as the intuitive understanding which visual displays provide.

There have been several tools and environments constructed over the past few years to assist in both visual programming and program visualization. These research projects constitute the core of the literature search for this examination.

III. Focused Literature Analysis

The focused survey of the literature unearthed a large number of visualization tools and environments for computer software development, including many which are specifically applicable to parallel software development. These tools and environments are separated into two distinct groups or categories, namely program visualization and visual programming as discussed previously. Beyond this high-level distinction, there are several subdivisions of these two categories into more specific groups which share similar characteristics. These subdivisions are based on the extent to which each tool or environment assists in the various stages of the software engineering design cycle, as shown in Figure 1.

Visualization may be applied at nearly all of these stages to provide useful information and guidance to the user. The method of grouping visualization systems emphasizes the perspective that each system takes in displaying information at a given stage. It should be noted that each visualization system may not necessarily apply to all or even more than one stage of the cycle, and this fact serves to assist in distinguishing such systems.

III. A. Program Visualization Tools

The tools for program visualization fall into several groups according to where they apply along the cycle. Beginning with the early stages of software development, the first stage involves problem analysis before algorithm and implementation issues are considered. There have been no tools yet designed specifically for this purpose, but there are several tools in a separate, more general category which are flexible enough to be applied here.

There are, however, a few tools which have been specifically designed for algorithm development. In fact, some of the earliest work in program visualization involved such tools. The **BALSA** (Brown university ALgorithm Simulator and Animator) and **BALSA-II** systems [BrS84,BrS85,Bro88] consist of a selection of different views for animating algorithms and abstract data structures. The emphasis of the **BALSA** tools is to display the actions of algorithms in an intuitive manner using the abstract data structure representations typically used on paper in planning for an implementation. **BALSA** views include literal graphical representations of trees and graphs, as well as intriguing visual representations of the progress of lists during sorting. Unfortunately, the **BALSA** systems require extensive development steps to produce

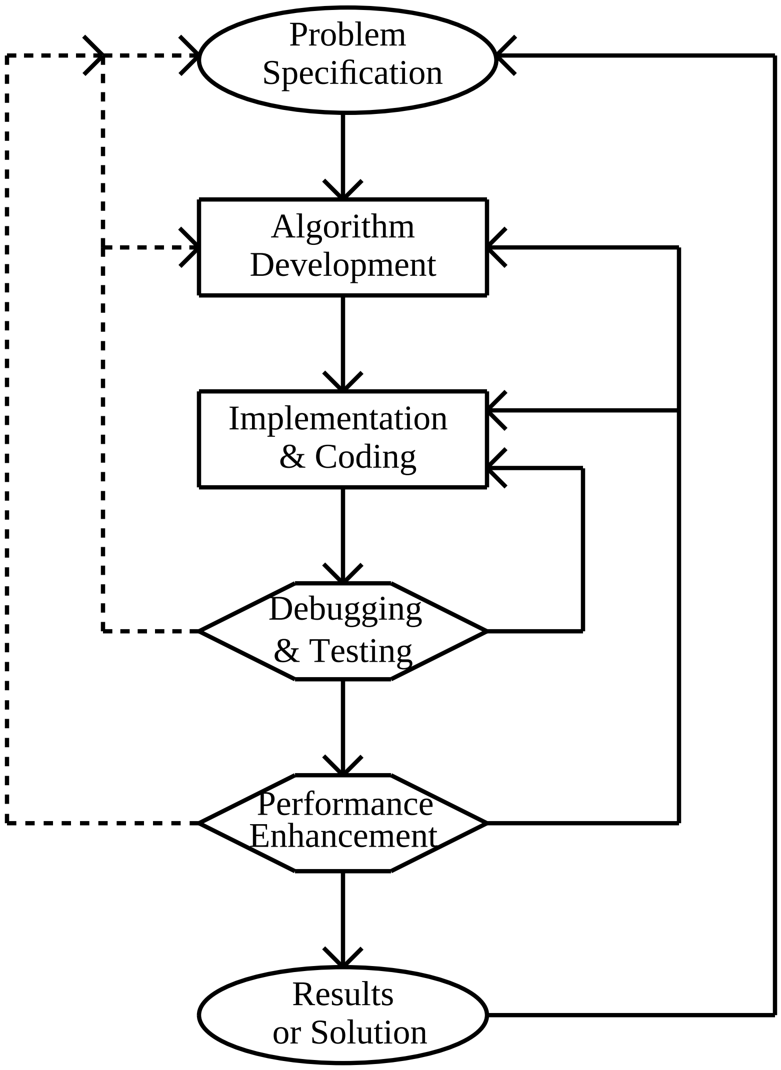

Figure 1: Software Engineering Cycle

each single animation. While effort may be saved by modifying past animations, the overall development cost is still high.

Another algorithm animation system is based on the **Smalltalk** system and actually provides a modularly constructed object-oriented animation specification environment for animating algorithms [LoD85]. The user creates an animation by integrating controller, model, and view classes to form custom animations. The resulting animations include abstract graphical objects which move and interact according to the rules relating the controller, model, and views. Again, as was true with **BALSA**, the development time for this animation is rather involved and requires development of a complete animation program.

The boldly-named "**Program Visualization**" system [KrB83,BrC85,Rae85] is another early tool designed for algorithm animation which includes several displays for abstract data structures, sorting algorithms, and other illustrative diagrams. This system, too, involves complicated animation program construction, but utilizes some library routines to reduce development overhead.

Although program visualization proves useful in some of the earlier software engineering stages for problem analysis and algorithm development, most program visualization tools have been developed for use with implemented programs which already execute on real computer systems. These visualization tools are often applied to debugging the implemented program or enhancing its performance. There is often little distinction between the types of displays used for debugging and performance enhancement, because many of the same displays benefit both types of analysis. Therefore, these tools are combined together into a single category. This category is composed of three groups which are distinguished by their perspective or approach to the visualization. These groups are *application-oriented*, *system-oriented*, and *independent performance metrics & statistics*.

Application-oriented visualizations emphasize the behavior and interactions of the software components of the application. These components may represent processes, tasks, program modules, subroutines, code segments, or data structures. Often, the views for this type of tool will display progress along a call-graph or a data-dependency graph, or will display interaction and communication at an abstract process or routine level.

One early example of an application-oriented tool is the **Incense** system [Mye83] which is an integrated system for displaying program data structures. Displays include visual representations for linked-lists, circular buffers, arrays, and even a thermometer-like iteration index display for the percent of loop iterations completed. Views are created using predefined *Artists* and user-defined *Prototypes* which control how information is displayed by specifying graphical functions. The default displays are limited but relatively straightforward to implement, whereas the development of the more flexible user-defined displays requires the less feasible creation of custom graphics driver programs.

Recent work similar to **Incense** has been done on the **VIPS** system [ShI91], which displays detailed information on linked-list data structures. The **VIPS** system takes a large step beyond

Incense by automatically extracting linked-list information at run-time via an interface to the **DBX** symbolic debugger in Unix. Though limited in general usefulness, **VIPS** is simple and easy to use.

Other automatic data structure displays have been created by [DiM90], and some iconic languages have actually been designed specifically for the purpose of displaying and visualizing data structures [SeR90].

Aside from the data structure visualization of the **Incense** and **VIPS** systems, there have also been several other tools and environments which have utilized different kinds of application-oriented displays. One of the most common displays seen recently involves the animation of program call-graphs. The **IPS-2** system [MiY89,MiC90] provides an active, animated display of a parallel program's call-graph as it executes. This system analyzes parallel programs built on a hierarchical software structure which includes instrumentation for the monitoring of events. The animated call-graph or *Program Activity Graph* is produced automatically from the monitored trace information, along with other information concerning the critical program path. By building programs within the framework of the layered **IPS** software structure, the visualization capabilities are implicitly developed along with the rest of the program. Hence, the results may be considered automatic. The **IPS-2** visualization systems are descendants of the earlier **DPM** system for visualizing distributed programs [Mil88] and the **IPS** monitoring system [MiY87].

The integrated **Faust** system [GuG89] includes a tool which provides separate windows for the call-graphs of each processor in a multiprocessor computer system. The **Faust** system uses program source, assembler, and object code, along with execution traces, to automatically generate the visual displays.

The **PIE** system [SeR85,Har90c,LeB90] is a programming and instrumentation environment, for developing parallel software, which includes some visual displays. One display format is used to display static program parse trees and animated invocation trees. The invocation trees grow dynamically as the program executes, showing the layout of concurrent activities across program modules. Another **PIE** display shows the activity of the program over time in a horizontal Gantt-like time-line which shows the active states of each module. All information necessary for these displays is generated through **PIE**'s instrumentation system which automatically captures execution traces at run-time.

Several other tools use program call-graphs or data dependency graphs for visualizing program execution, such as **CODE**, **E/SP**, **GRAPE**, **SCHEDULE**, etc, but these tools emphasize visual programming aspects of visualization and will be described in more detail with other such visual programming tools.

There are several new types of application-oriented displays currently under development which are more creative than simple graphs or data structure displays. The **Gauge** system under development at Aerospace Corp. by Carl Kesselman utilizes certain mathematical theories to efficiently derive parallel program execution information from a minimal program instrumentation with little intrusion. Very low-overhead counters are automatically inserted into the user's program and then incremented throughout program execution as the program flow passes over them. Upon completion of execution, these counters are inserted into a special differential equation which is then solved to derive performance data. The resulting performance data contains information concerning aggregate execution time, idle time, and other performance statistics for each subroutine or program module. This information is displayed in two- and three-dimensional (using color as a third dimension) displays which allow interactive viewing of statistics on a linear or logarithmic scale for a single procedure or all procedures, and for a single processor or all processors. The relative values among procedures and/or processors reveal performance bottlenecks as well as synchronization and program flow errors. The intrusion into the program's execution is less than five percent, hence the instrumentation may be left in production codes. Though the information provided is very useful and easy to obtain, it is also very specific and there is no way to maintain user-defined event information.

The **Ravel** tool, under development at Argonne National Laboratory by Ewing Lusk, provides an intuitive visualization of interprocess or interprocessor communication. Based on execution traces, the **Ravel** tool dynamically creates and places colored circles to represent processes or processors, and then animates flashing arrows between pairs of circles to represent communication or the transfer of messages. The color of each circle corresponds to the current state of that process or processor, as is determined by pairs of trace events (one event signals the beginning of a state and a matching event signals the end of that state). Note that though **Ravel** can display interprocessor communication, a seemingly *system*-oriented characteristic, the view is still oriented towards the structure of the software application and not the underlying system architecture. The **Ravel** tool does not take part in the generation of program traces, but provides a simple and easy-to-use interface for displaying provided traces.

Various other application-oriented techniques are appearing, as new paradigms for parallel programming are developed. The **HeNCE/PVM** system being developed by Jack Dongarra at Oak Ridge National Laboratory and the University of Tennessee provides an environment for constructing distributed applications on groups of heterogeneous machines. Currently under development is a visualization system for displaying actions and interaction among the resulting distributed processes.

The **TupleScope** tool [BeC90] visually displays the abstract tuples in Linda-based parallel programs. This system provides a view of the Linda "tuple-space" and uses small balls and arrows to reveal the creation and flow of tuples.

Aside from the large set of application-oriented visualization tools, there is another large category of tools which emphasize or display characteristics of the target computer system architecture, or the system-oriented tools. These tools display the behavior of the target computer architecture, including both hardware and operating system features, in response to a particular user application or program. Typically, these tools display behavior or activity, as well as interaction among system components, from low-level hardware through higher-level operating system components and constructs.

At the simpler, but not necessarily less useful, end of these system-oriented tools are those which visualize characteristics of systems using only bar graphs, charts, and other visual diagrams. Although similar to tools with general statistical displays, these tools are somewhat more specific and complicated, and so are included with the system-oriented tools. One example of such a tool is the **Axe** environment [Nic90] for analyzing concurrent systems. **Axe** provides several statistical displays including system load and interprocessor communication traffic, and also includes a processor status grid which dynamically visualizes each processor's activity during execution. Because the information displayed is easily obtained from system-provided statistics or from other simple modeling parameters, **Axe** is able to provide its analyses with minimal user involvement.

The **Simple/Care** system [Nic90] produces simple plots and bar graphs of system statistics, and includes a diagram which displays message activity over a topologically-organized display of processors. Each processor is represented by a square which is shaded in to show the processor state, and lines are drawn between processor pairs to represent messages.

Another tool along similar lines is the **PARET** tool [NiE88] for modeling multicomputers. This tool's central display is similar to the application-oriented call-graph displays, but the emphasis is towards system characteristics. The call-graph display includes information about which processor executes each node and how work is distributed among different processors. Surrounding the central display are smaller graphs and plots which show various processor status information. The information displayed by **PARET** does not come from execution traces but rather from user-defined models of the target system and application which **PARET** uses to simulate the behavior. The construction of these models appears straightforward but somewhat involved.

There are some tools which produce more realistic visual depictions of system components and information. The **DV-DRAW** package [Gai90] provides a visualization, of shared-bus multicomputers, which is more representative of the user's physical view of the system. The display appears as would a mental picture of the machine, with a central placement of address and data busses and various collections of small diagrams to represent each processor and each memory module. The small diagrams display current status, status history, bus contention, plus information on memory accesses. **DV-DRAW** is actually a highly configurable graphics editing system for constructing visual displays, and uses previously collected trace data from various sources to animate its displays. It appears that development of each particular system's displays requires a substantial effort.

The **Q+** visual performance analyzer [Nic90] (a revised version of the **Performance Analysis Workstation** project [MeM85]) uses user-specified parameters to model and simulate network traffic and behavior. The user inputs a model visually, as if drawing a simple diagram of nodes and arcs, and then sets various parameters to model probabilistic behavior, priorities, blocking, routing, etc, to complete the model specification. Then, **Q+** reads user-provided execution traces as inputs and visually simulates the behavior of the network. The configuration of the network models appears to be relatively straightforward and is supported well by the graphical interface.

A similar but more specific network simulation tool is the **Gecko** package [HaF90] which displays transputer networks based on input from the **Transim** transputer simulator. **Gecko** automatically produces topographical displays of transputer networks, with nodes labeled according to which processes are currently executing. There is also a simpler display which visualizes the framework or mapping for the sharing of results among transputer nodes. This display shows the general flow of information through the system. Due to the special-purpose nature of this tool, the visualizations are completely automated and easy to use.

Beyond visualization in the image of the physical system, some other new techniques have been introduced for more creative use of visualization in analyzing system behavior. The integrated **TOPSYS** system [BeE88,Bem88,BeB90,BeB91] contains tools for debugging, performance tuning, and animating parallel program execution. The performance tuning tool, **PATOP**, produces only simple graphical plots of performance statistics, but the animation tool, **VISTOP**, actually animates the behavior of a system using several graphical objects. These objects represent tasks, mailboxes, and semaphores in a parallel system. During execution, these objects move around the display, flashing arrows between themselves to represent interaction and synchronization activity. The graphical objects may be selected individually for closer examination, and can open up into windows with textual descriptions of their current states. The **TOPSYS** environment is integrated with the **MMK** operating system to allow efficient and automatic extraction of the necessary information.

Other interesting displays have been constructed for viewing rather vague or complex system characteristics which have no concrete physical representation. Typically this type of display uses fabricated abstract views which intuitively reveal conceptual information. The **Multiple Views** approach [LeM87,LeM90] includes several displays for revealing elusive or vague system characteristics. One such display uses colors in large squares with varying shades to represent the progress of matrix operations. The memory used to store each matrix is depicted as a two-dimensional area, and the progress through a matrix manipulation is indicated by appropriately flowing patterns of colors which represent the corresponding memory accesses. Another display shows processor interaction using a *Moviola* diagram, which represents processes as vertical lines along a vertical time axis and draws lines between processes to indicate communication over time. The location of the two endpoints of each communication line indicates the moment in time when the communication begins and ends. This system also includes several interesting displays such as a space-time diagram and some statistical plots and graphs. The system is driven by externally captured execution traces which are obtained from instrumentation of software libraries using the **Instant Replay** system.

Another novel visual display technique is implemented in the **SHMAP** system [BrD88,DoB90] for visualizing matrix algorithms on shared-memory multiprocessor computers. Memory accesses are displayed on two-dimensional canvas grids which represent matrices - similar to the above matrix display, but with separate canvases for displaying reads and writes. As a memory location is accessed to examine or modify a matrix element, the corresponding grid square is colored in on the appropriate canvas. The color of the access corresponds to the processor which made the access, and the accesses fade to black with time. This fading quality results in an intuitive visualization of program locality through the flowing memory access patterns. The speed of fading, as well as the speed of the overall animation, can be adjusted to allow different levels of perspectives. The tool also includes the capability for multiple shared caches with similar access coloring of special cache canvases, plus an additional marking of cache accesses to distinguish hits and misses. **SHMAP** simulates parallel execution by allowing the user to configure the number of processors and the specific cache system. The number of caches and the size of each cache are flexible, as is the cache replacement policy. The animation is driven by execution traces obtained via instrumented linear algebra libraries for sequential Fortran programs. The instrumentation of matrix functions includes specification of concurrently executable operations to allow animated simulation using the various **SHMAP** multiprocessor configurations. Traces are obtained easily by linking in the appropriate libraries.

Yet another intriguing approach to visualization of system-oriented characteristics is that of the **Pictures of Performance** views [Rov89,RoW90,RoC91,Rov91]. This approach combines the technique of scientific visualization with massively parallel computer systems. Parallel computers with large numbers of processors are displayed in two- or three-dimensional cubes, where each processor is associated with a particular grid point throughout the cube. Execution trace information is used to compute various processor status and statistics which are then displayed, on a processor-by-processor basis, using coloring of the grid points. The space between the processor grid points is filled in with an average color/value of the surrounding processors, producing a smooth and continuous display of the overall multiprocessor activity. The resulting view resembles scientific visualization, similar to what would be seen in visualization of heat flow or stress across a solid material. This approach has been applied in several visualization tools to produce views for multiprocessors with over 16000 processors.

Another approach to system-oriented visualization may be seen by the **TMP** (Test and Measurement Processor) hardware monitor [HaW90], which supports a visualization system for viewing the actions of parallel software on a parallel or distributed system. With the **TMP** hardware installed with each processor in a parallel computer, program execution information may be captured and displayed in real-time at a central graphics workstation. The visual displays show various performance metrics and status information for each processor in the system. Each processor is allocated its own small window which includes several displays in the form of bar charts and Kiviat diagrams [Mor74]. Information concerning the overall system is displayed in similar diagrams. The visualization is all produced automatically and highly efficiently with the assistance of the extra monitoring hardware, but this extra hardware is clearly more expensive.

The next category of program visualization perspectives includes tools for the display of independent performance metrics and statistics. The two types of visualization in this category are similar in their intent and style, so belong to the same category, yet are distinct in their level of theoretical complexity and sophistication. Independent performance metrics refer to very particular statistics which seek to reveal the quality of the relative mapping of application to architecture, but without emphasizing either the application or the architecture explicitly. This concept has been explored outside the field of visualization by Jamieson et al [JaG87a] and Martin [Mar87], but has only just begun to appear in actual visualization tools.

The **PAWS** (Parallel Assessment Window System) performance evaluation tool [PeG91] uses visualization to compare a common application on various existing, prototype, and even conceptual machine architectures. **PAWS** applies separate analyses to each of application and architecture and then provides a consistent independent analysis, with a graphical user interface and representation. The application is transformed, with the user’s assistance via a graphical interface, into a visual representation which emphasizes the control and data flow. The architecture is characterized by combining analytical models with benchmark results to form a set of timing values. Next, the application model and the architecture model are combined and compared by a performance assessment tool which produces the resulting graphical output. This graphical output is in the form of numerical plots which display the maximum application parallelism over the course of its execution, and also plot the appropriate speedup which would be observed on the chosen architecture. These statistical values are independent of the application’s programming language or any specific architecture components. Once the user has specified the necessary application characteristics, the remainder of the analysis is automatic and easy to use.

Aside from the above tool category with more complicated theoretical statistics, there are a number of tools which visually display standard performance metrics and statistics. The most complete and flexible collection of such statistical displays is provided by the **ParaGraph** tool [Hea90,HeE91] for animating the performance of parallel programs. **ParaGraph** supports a multitude of different statistics and display techniques for visualizing them. Nearly all displays are animated over time or displayed on a sliding horizontal time axis, and effective use is made of color and shading techniques. The display views include a Kiviat diagram for dynamically displaying processor utilization and load balance in a circular diagram, with darker shading representing instantaneous values and lighter shading representing "high-water" marks. A horizontal "space-time" display shows processor interaction with angled lines along the horizontal time axis, and this same information is also animated dynamically using appropriately shaped architecture topology graphs with communication arcs between processor nodes. Gantt charts display colored processor activity over time, and animated sliding histograms and dynamic bar charts visualize a selection of other statistics. The information for computing the various statistics for **ParaGraph** is obtained from the **PICL** instrumentation library which automatically captures events from message transfers among processors, as well as any other desired user-defined events. The methodology is straightforward and easy to use, yet provides a great deal of useful information and the capability for expansion as necessary.

Another diverse collection of statistical displays is provided by the **Hyperview** performance data collection, analysis, and visualization environment [MaR89,Nic90]. This integrated system for analyzing the iPSC/2 hypercube extracts communication information in a way similar to that of **ParaGraph**, and then produces a number of revealing visual displays. There are animated dials, bar charts, LEDs, Kiviat diagrams, matrix views, x-y plots, contour plots, strip charts, and a general purpose graph display. The displays are configured and controlled via a high-level interface which selects filters and displays for the desired data. The filters and displays may be dynamically and arbitrarily combined to produce a highly flexible selection of views.

The **Preface** tool [BeB89] provides several displays of standard performance metrics and utilization distributions. Data is extracted by instrumenting code on a shared-memory multiprocessor system, and is then analyzed for visual display. There are displays which plot instantaneous speedup over time with an average speedup water-line, and displays for percent utilization over time and aggregate activity distribution per processor.

Aside from the above tools, there are also several visualization tools which include general statistical displays amongst their more sophisticated ones, such as the previously mentioned **IPS-2** system which includes several performance metric plots.

The last stage or phase of the software engineering cycle deals with the results or solution of the problem under consideration. This stage may utilize visualization, but not necessarily in the context of the above analysis tools. In this phase, the use of visual rendering for results falls more appropriately into the category of scientific visualization or other non-software-oriented categories, such as CAE or visual modeling and simulation. In fact, the result of the problem computations may very well be an animated simulation or rendering. The visual information obtained in this stage is problem-specific, and serves only to generate new problems or modifications to existing ones. Discussion of this type of visualization will be included in the sections which describe the overall, general taxonomy.

Finally, there are several program visualization tools which transcend the previous distinguishing perspectives of application, system, and statistics, and even the boundaries into the realm of visual problem and algorithm analysis. These visualization tools apply equally well to all of these areas and categories, and prove to be generally useful across the entire software engineering cycle. They all have in common the fact that they are driven by program execution traces, but these tools place no predetermined meaning into the trace events or their origin. Rather, they provide general frameworks for analyzing arbitrary user-defined events and information.

It should be noted that all of these tools are explicitly driven by the event traces, and that in most cases the collection of such traces is not considered or assisted. The traces need not even come directly from actual program executions, but may be taken from simulations or probabilistic synthesis techniques. In fact, nearly all tools which use execution traces might be driven by such fabricated traces.

The generally useful visualization tools fall into two distinct sets with drastically different levels of sophistication. The first set of these tools is rather straightforward and simple, yet has a high value in analyzing parallel programs. They are very general event display tools which visualize the events themselves by interpreting pairs of events as signifying the beginning and ending of states of activity for processes or processors. These displays are often attributed to Gantt's work in Industrial Management in the early 1900's [Gan19]. The form of the displays is usually a collection horizontal lines which represent the activity of each process or processor over time. Colored or textured bars appear along the lines to indicate the current states, and the arrangement allows easy relative comparison of process or processor status. Information regarding synchronization and idle time are easily and intuitively derived visually, and the state histories can be used to verify proper flow of execution through each thread in parallel programs.

The **Upshot** tool [HeL91] is one example of a tool with this type of Gantt event display. **Upshot** produces a static display but allows scrolling forward and backward in time, as well as zooming in and out by adjusting the time scale to provide varying degrees of detail. It is also possible to display each beginning and ending event number, along with the colored state bars, to add extra detail as to the precise location and ordering of individual events. **Upshot** also provides a graphical interface for specifying state event pairs and the corresponding desired colors.

Other tools in this category with similar displays include the **PM** system profiler from Parasoft [Nic90] and the **Jed** event display tool from Malony at the University of Illinois [Nic90]. In addition, several more sophisticated visualization tools include Gantt-like event displays in their view selection, such as those previously mentioned in the **ParaGraph** tool, and the view from the **Impact** event-display tool in the integrated **Faust** environment.

The remaining set of generally useful visualization tools are the most elaborate and sophisticated of all visualization tools, and provide environments for arbitrarily visualizing and animating event traces. These tools utilize the most sophisticated types of animation and graphics techniques available to produce complex visual models which can portray behavior from all the categories: problems, algorithms, applications, systems, and statistics. These portrayals may be either literal visual representations of concrete physical entities, phenomena or characteristics, or may be highly abstract visual depictions of fabricated objects, mental constructs or concepts. These abstract, general tools are clearly descendents of earlier work on algorithm animation, but they have gone beyond the limited perspective of those earlier tools in flexibility and ease of use, as well as scope of utility and function.

The **Voyeur** system [BaS88] was perhaps the first of the new generation, and provided a framework for defining various animated views of executing parallel algorithms, applications, and systems. An interesting algorithm animation produced by **Voyeur** displayed shark and fish icons interacting in a two-dimensional grid space. The behavior of the algorithm was viewed as the visual movement and activities, such as "eating" and "giving birth", of the sharks and fish. Another **Voyeur** view dynamically displayed a collection of data vectors in a two-dimensional space. The length and direction of an application's data vectors were numerically plotted instantaneously throughout a computation to locate a numerical instability problem. Another view showed the communication traffic among processors in a computer network. Processors were arranged in a mesh topology with lines of varying thickness to indicate message traffic along the mesh. Unfortunately, use of **Voyeur** views require development of several program modules and interfaces for each view, so extended popularity and use has not been evident.

More recently, the **Tango** system [Sta90] has been used to animate the execution of various computer activity. **Tango** provides a systematic approach to animation creation which involves designing a mapping between an application and the desired animation. Animations are constructed using *images*, *locations*, *paths*, and *transitions* to specify the flow and view of the animation. The mapping between application and animation is accomplished via *associations* which relate animation images to various program parameters and behavior. The user is able to interactively instrument and annotate a program via a special **Tango** interface. The resulting animations can visualize any aspect of program execution, from algorithm features to application entities and physical system interactions. The visual interfaces expedite the creation of views making **Tango** a more feasible approach to analysis than some other such systems.

The integrated **Triplex** parallel-execution monitor system [CoK90,Nic90] provides a configurable visualization environment for visualizing parallel software on the NCUBE hypercube system. The system consists of three separate tools, **Simplex**, **Seenote**, and **Seeplex** which integrate together to produce the desired visualizations. **Simplex** is a parallel operating system for the NCUBE which handles the instrumentation of program execution with very low overhead (less than 5 percent). The **Seenote** tool provides a raw textual display of event histories obtained from **Simplex**. The **Seeplex** tool provides a real-time execution animation with visual displays. The visual displays of **Seeplex** are generated via an interactive graphical editor which constructs a *Trellis*, or acyclic graph, for specifying a sequence of filters with display outputs. The trace information flows through the Trellis filters and is transformed, summarized, and organized to be output in various display views. These displays use sophisticated visual techniques to represent and animate information for large numbers of processors. Groups of processors are grouped or *binned* together based on similarities to result in concise displays which represent the overall status without unnecessary redundancy. The displays include various three-dimensional organizations of hypercube nodes with flexible visualization of values and statistics. The system is relatively easy to configure and has grown in popularity among NCUBE users as a means for visualizing parallel program execution.

The **PARADISE** environment [KoC91] provides a methodology and a "meta-tool" for constructing custom visualization tools for animating parallel software execution. The meta-tool classification applies because **PARADISE** is itself a visualization-oriented tool which stands as a foundation and provides a structure for developing other visualization tools. With **PARADISE**, an abstract visual model may be defined and constructed, with the assistance of a graphical editor interface, to produce animated simulations at arbitrary abstraction levels. The user specifies the desired visual model by constructing modular objects which organize and encapsulate individual aspects of the overall system functionality. These objects are visually connected to form an overall model structure. The visual model is object-oriented, but the objects have such well-defined interfaces that they actually become visual objects and may be completely manipulated visually. With a execution trace of events to drive the simulation, the visual model is animated, showing the behavior and interaction of the visual objects as a result of the trace information. Though the interface for designing the structure and functionality of visual objects requires some effort and is still under development, the methodology is extremely flexible and allows animation at any abstraction level or combination of levels from problem to algorithm to application to system. The choice of visual model is arbitrary and is left to the discretion of the user as far as complexity and graphical sophistication are concerned. There is much potential for expansion of this environment to improve its user interface and its capabilities for visualization.

Another similar, general-purpose, customizable system is the **Macaw** system [Nic91] which attempts to provide a general interface for custom animated simulations. This tool uses the concept of an *event manager* and *data probes* to gather and display event sequences. A central issue discussed in this work is the recurring problem of animation specification and its complexity.

III. B. Visual Programming Tools

The tools for visual programming apply primarily to the first half of the software engineering design cycle, for the creation and implementation of an application and its subsequent mapping and partitioning to the target machine architecture. In the broad perspective, these visual programming environments may be divided into two groups based upon their fundamental approach to the application of visual or graphical techniques. The first group includes those tools which utilize graphics only to organize program modules or routines which are written in ordinary sequential text, and the second group includes those tools which go beyond this simpler approach to provide abstract visual languages or use graphical objects with visual semantics. Note that because the emphasis here is more oriented towards program visualization, only a brief survey of the more well-known visual programming environments is provided.

The first group of visual programming tools orient towards assisting the user in connecting together modules of sequential, textual code to form concurrently executing programs. The central approach of these tools is to utilize a data or program dependency graph to specify parallelism, the idea being that the portions of the computation which are not directly dependent upon each other can execute in parallel.

This type of visual programming tool generally leaves the responsibility of identifying parallelism to the user, but assists the user by providing graphical interfaces and any available automatic guidance. Often, these tools will automatically generate an initial dependency graph and then allow the user to apply knowledge about the application to manipulate the graph for increased parallelism.

The **CODE** (Computation-Oriented Display Environment) system [BrA89] is a visual programming tool which assists the user in programming with the **ROPE** (Reusability-Oriented Parallel programming Environment) software system [BrL90] for parallel programming. **CODE** allows creation and design of parallel program dependency graphs, where the nodes represent subroutines or code segments for computation and the arcs connecting them specify the dependencies. The resulting graphs include visual semantics for subgraphs and input/output filters in addition to the notation for computation segments. Elements of the **ROPE** component library may be used as the computational nodes for **CODE** to assist in constructing parallel programs an reduce unnecessary textual coding. Future work on **CODE** and **ROPE** will include an animation of program execution using the dependency graph, but this feature is not currently available. No guidance is provided to the user to assist in constructing the dependency graphs at this time.

Another slightly different approach may be seen with the **SCHEDULE** tools [DoS87,JaG87b] for developing and analyzing parallel Fortran programs. With **SCHEDULE** the user still specifies a dependency graph, but with a higher grain size - using processes. Each process is assumed to be a sequentially executing section of code made up of one or more tasks. The dependency of execution of these processes is specified graphically and then used to *schedule* their execution on a parallel or distributed computer system. During execution, **SCHEDULE** produces an animated call-graph of the processes, depicted as small circles, with subtask execution animated using a row of rectangles below each process node. To use the **SCHEDULE** mechanism, the user's Fortran code must be modified, potentially dramatically in some cases, to organize the information into independent tasks which can be scheduled. The more complex dependency information is specified graphically, so the overall transformation is straightforward.

The **Poker** parallel programming environment [Sny84,Nic90] uses a graph-based approach to visual programming in which the user specifies the structure of communication among processes. Parallel computation is modeled in *phases* which are represented by the process communication graphs. In these graphs, the processes are represented by nodes, and the communication channels betweens processors are represented by arcs. Processes are visually organized and connected, with named communication ports which are referenced during communication in the textual source code for each process. The textual programming language used for **Poker** programs is extended to handle the necessary special operations involved. The overall programming task is a mixture of text and graphics.

The **GRAPE** (GRAphical Programming Environment) tool [LaE90,EnL91] allows construction of parallel programs in a similar manner to the above tools, with graphical specification at a high level and textual specification at the lower level. **GRAPE** includes the addition of graphical as well as textual libraries for constructing visual programs, and several code generation features are available. Visual interfaces are provided for interactively controlling applications at run-time through the target machines communication network, and tools are under development for animating the execution of the visual programs.

Another system [PuJ91] uses a **Module Interconnection Language** to connect program modules, and then produces an execution visualization of the resulting graph using built-in instrumentation. This approach is designed primarily for distributed systems.

A different type of graph-oriented visual programming approach involves the modification and manipulation of existing data dependency graphs. A tool which uses this approach is the **E/SP** tool [BrS89,SrM89,Har90c] for the parallel structuring of Fortran programs. **E/SP** automatically provides the user with a data-dependency graph for an existing Fortran program and then guides the user through a restructuring for increased parallelism. By providing performance information regarding the critical portions of code, **E/SP** leads the user to the code which will produce the most gains by increasing its performance. The user then applies knowledge of the application to resolve those ambiguities which limit parallelism. The restructuring of programs is done at the level of loops and subroutine calls to introduce medium grain parallelism. Use of **E/SP** has led to parallelism increases close to those achievable by manual coding techniques but in substantially less time, and have far surpassed automatic techniques by allowing utilization of user knowledge.

A similar set of tools for parallelizing Fortran are found in the **Start/Pat** toolkit [ApS89,Har90c]. The **Pat** tool is similar to **E/SP** in producing a dependency graph and allowing the user to modify and manipulate it for increased parallelism, however **Pat** also goes further to guide the user by actually suggesting alternatives for parallelism based on knowledge of parallel primitives. In addition, **Pat** continually verifies that the original program data dependencies remain intact and are not violated. The **Start** tool provides a static analysis of the resulting parallel program to visually reveal the concurrency obtained through a *Concurrency History Graph* (CHG). The CHG shows a call-graph history with concurrency state nodes and synchronization arcs. **Start** also displays a *Synchronization Augmented Flow graph* (SAF) which shows control flow of only those portions of the program which deal with synchronization.

Another example of this type of tool may be found in the **PRETS** (Parallel Reverse-Engineering Tool Set) environment [Har90c] which uses the **Parallax** graph editor to similarly restructure serial Fortran programs for parallelism.

A slightly lower-level dependency graph programming tool has been developed for the Navier-Stokes Computer (NSC) [ToC88]. This visual programming tool allows creation of low-level code for controlling the complex vector-pipelines in the NSC hypercube. The user graphically specifies a series of pipeline diagrams which indicate the desired interconnection and function of the pipeline units. Each of these diagrams corresponds to a single program "instruction", and the tool automatically produces the necessary micro-code for each resulting diagram configuration. The diagrams are similar to expression trees and link together sets of inputs through functional operator units to produce the desired outputs.

The second type of visual programming environment utilizes more sophisticated programming techniques such as abstract visual languages, theoretical models, and graphical objects. These techniques often allow more flexible generation of parallel programs with minimal or no textual specification, and typically introduce more involved formal analysis capabilities.

The **XPAT** system [PaC90,PaC91] uses Petri-Nets to specify parallel or distributed computer programs. **XPAT** utilizes several mathematical transformation techniques to assist in specifying complex computation with the Petri-Nets, and maps the resulting model into program code. The Petri-Nets themselves are specified visually via a graphical interface environment. Use of the Petri-Nets allows application of several formal, theoretical analysis techniques to parallel software structure and flow, including deadlock and reachability analyses. Research is in progress to support the use of code generation schemes.

The **VERDI** (Visual Environment for Raddle Design and Investigation) environment [ShR90] is also based on an extended version of Petri-Nets, but with extended graphical forms which embody the expressive power of the textual **Raddle** language. The emphasis of **VERDI** is on modeling the control and communication structure of the design, and so a program is specified as a composition of asynchronously interacting processes and procedures. The visual language specification utilizes various graphical shapes and constructs to represent the semantics of a program. These constructs identify primitive actions and their grouping or sequencing into processes, procedures or teams. Programs are created via a graphical editor, and may subsequently be executed and animated using color and motion with the graphical objects. **VERDI** provides a completely graphical language approach to parallel programming.

More general graphical languages are supported by the **Graspin** development environment [MaM89,Har90c] which provides a framework for utilizing structured, general, graphical languages. **Graspin** supports formal techniques for constructing, verifying and validating the development of concurrent and distributed computer systems through visual language specifications. A flexible internal abstract machine representation is used to build an interactive visual programming and development environment for a wide range of visual and textual languages. A graphical language is represented as a collection of graphic shapes or symbols which have specific visual semantics and syntax for their arrangement. Resulting visual programs may be formally analyzed for proper semantics, algebraic verification, and other useful properties of both static and dynamic behavior.

A similar tool, **VPW** (Visual Programmer's Workbench) [RuW90] provides an environment for programming distributed computer systems using visual languages. **VPW** introduces more detailed specification of the programming environment architecture and the execution model using the distributed processing paradigm. The abstract spatial structure of programs is visually parsed to construct universal relations which represent the programs. Various analyses such as those of **Graspin** are possible, although somewhat less formal. **VPW** grew from early work on textual language synthesis environments like **Gandalf** [Med82] which also utilized knowledge of the languages for manipulating structure and semantics.

Other types of tools for parallel program construction have been created to utilize various visual representations and techniques. The **PPSE** (Parallel Programming Support Environment) parallel construction system [Har90c] provides graphical interfaces for specifying task-level data and control dependencies, target machine architecture descriptions, and task mapping and scheduling. In addition, the system includes a parallel code generator and a performance analyzer. Tasks are organized and scheduled according to their dependencies and the target architecture description, and the schedule is displayed visually using a vertical Gantt-chart representation. The system is still in prototype form.

Some of the earlier work in visual programming resulted in other short-lived visual programming approaches which captured and visualized various aspects of program design. The **PECAN** program development system [Rei84,Rei85,Rae85] utilizes several different graphical program views in assisting program development. As the user specifies portions of a program, **PECAN** incrementally compiles and analyzes the program to create graphical views of the symbol table, data types, arithmetic expressions, and control flow. The **PegaSys** system [MoH85,Rae85] also uses this type of visual display technique. With **PegaSys**, graphical images are used to build a hierarchical specification of control and data dependencies for data objects, program modules and processes in a complete program. The visual representation is converted internally to a calculus which organizes and forms constraints for the specification.

Some simpler early tools include **Omega** [PoL83,Rae85] and **PiP** (Programming-in-Pictures) [Rae84,Rae85] which used various graphical means for visually specifying programs. **Omega** used a primarily menu-based technique to select program elements and assemble programs, while **PiP** used a slightly more advanced graphical object approach for specifying relatively simple programs. **PiP** approached programming as a simple connection of data object type and function, and provided a visual framework for specifying these in a graphical form.

IV. Taxonomical System and Quantitative Metrics

This section will build on the discussion and organization of Section III to produce a full taxonomy of visualization tools for parallel software development, but will first begin by classifying the uses of visualization tools in general to reveal the context into which these particular tools fit in the overall field of visualization. In addition, several quantitative metrics will be defined and described. These metrics will serve to provide a method of classifying visualization tools beyond the qualitative taxonomy.

IV. A. Overall General Visualization Taxonomy

Based on the broad survey of literature concerning the general application of visualization in computer systems, a taxonomy has been constructed which includes a wide range of fields and disciplines which have utilized visualization in some form as a means to represent and understand complex information. The overall view of this taxonomy is provided in Figure 2.

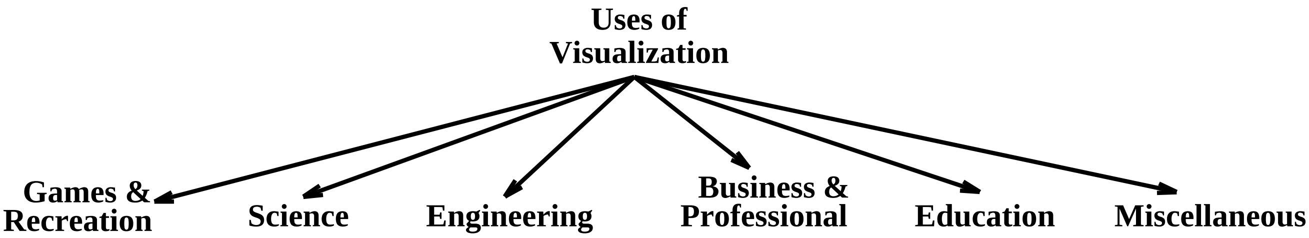

Figure 2: Overall Uses of Visualization

At the top level of the taxonomy there are several classes which distinguish the general fields or disciplines which have used visualization. Most visualization systems are substantially application-specific, so the classification of the field or area of work serves as the best means for distinguishing systems at this level. The different fields are organized into the following groups: *Engineering*, *Science*, *Business & Professional*, *Education*, *Games & Recreation*, and *Miscellaneous*.

In the general area of engineering, there are of course many specific engineering fields which utilize visualization. These fields are organized and grouped by their general manner of applying visualization, in particular the impetus for visualizing or the type of desired visual result or benefit. Engineering is split accordingly into four specific classes: **CAD**, **CAE**, **CASE**,

Modeling & Simulation, and **Other**, as shown in Figure 3.

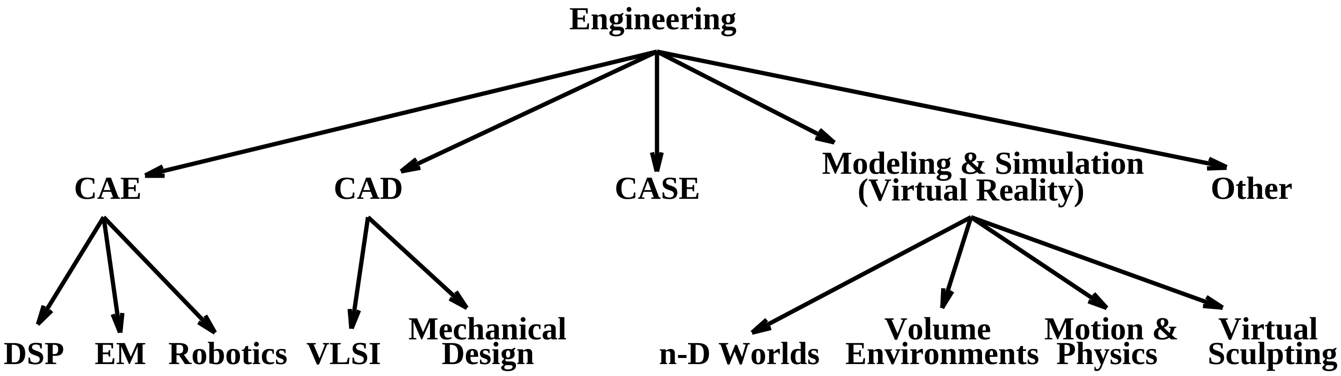

Figure 3: Engineering Visualization

CAD refers to Computer-Aided Design and includes some of the more well-known applications of visualization. The more prominent CAD applications are **VLSI** design and **Mechanical Design** and modeling. VLSI design utilizes visualization for many assorted tasks beyond simple channel routing on semiconductor chips. Visualization has been used in **SHORTFINDER** for locating short circuits in VLSI layouts [Gan90], in **UGRAF3** for analyzing and measuring VLSI device parameters [BaB90]. Other VLSI design functions may be found in [AnP89,ChM89a,KuO90,LiB89,Ven90]. Beyond chip design, CAD systems have been created for analyzing and prototyping computer networks [DuS90] and local area networks [PeR89], and in configuring hardware systems [LeL90]. Visual techniques have also been used for other electronic design functions, including analog circuitry [Hin90].

Mechanical design uses visualization to display the physical structure and analysis of machines or parts for various mechanical items, including automobiles, aircraft, etc. Visualization is applied to allow efficient and low-cost design-specification and modeling using a computer before actually constructing expensive physical prototypes. The **COSAR** system utilizes 3-D graphics to model complex three-dimensional objects using a finite element approach [BeA90]. Other applications use object features to visually model mechanical devices during design [ChP90]. Mechanical engineering also uses visualization in its design processes [JoM90].

CAE refers to Computer-Aided Engineering, a very general realm of engineering applications in which visualization assists engineers in observing theoretical or experimental phenomenon. Examples of this type of visualization include **Robotics** simulations [KuB91a], electromagnetic (**EM**) wave depiction [Cen90], and animating signals in digital signal processing (**DSP**) [Mat90]. These types of visualization generally display experimentally obtained information to provide observation of trends or patterns, either for theoretical reasons or to examine an engineered device or project in operation.

CASE refers to Computer-Aided Software Engineering, wherein lies the central issues of interest for this comprehensive examination. Visualization applies in a wide range of software engineering functions and operations [BuJ89,ChN89,Oma90a,Tyu91]. These operations include **Configuration Management**, **Design Team or Groupware Management**, **Documentation**, **Software Maintenance**, **User-Interface Design and Analysis**, and, of course, **Software Development** - specifically **Program Visualization** and **Visual Programming**. This portion of the taxonomy is shown in Figure 4.

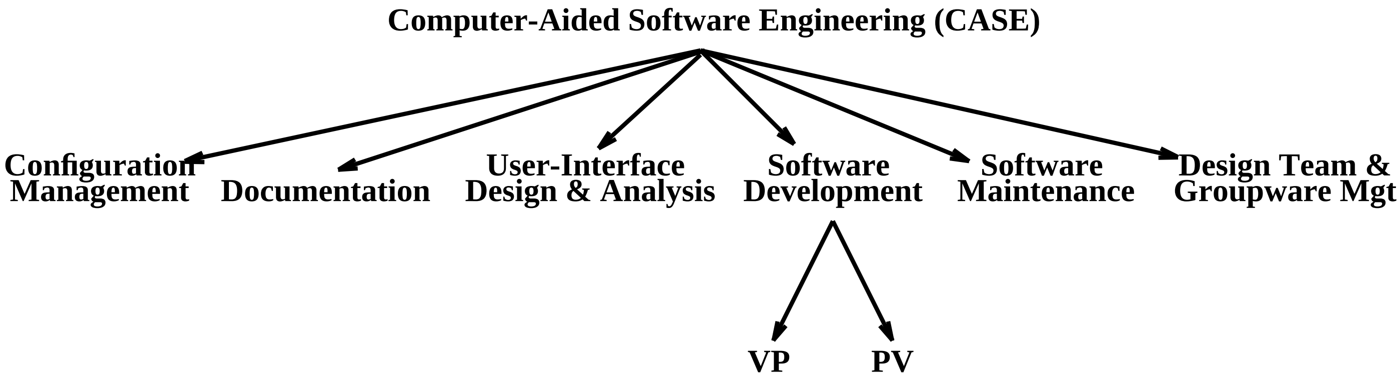

Figure 4: CASE Visualization

Configuration management refers to the organization of multiple software versions. Visualization has been applied here to manipulate working contexts in [Kob90].

Design team or groupware management refers to the organization of human and computer resources in the development of a group software project. A visually assisted analysis of such groupware activities may be seen in [MuS91]. Visualization has been applied to production team management as well [Dip90].

Software documentation benefits from visual techniques which assist in layout design using graphics. An example of such a system may be found in [IwK90].

Software maintenance refers to the task of keeping software working properly by fixing software bugs as they appear, modifying software to adapt to environment or systems changes, etc. Visualization has been used for this task in many different systems [KaF88,Oma90b]. The structure and complexity of software has been visually analyzed in [ScA89].

User-interface design and analysis is in many ways a microcosm of the entire visualization field. Determining the ways to use graphics and visual representations to assist computer users (definition of visualization), might even be considered by some to be a subclass of user-interface design. In reality, user-interface design and analysis work is a part of the visualization field at a fundamental level because it deals with the inherent issues in communicating human information to computers and back again. Approaches to visualization should stem partially from or be based upon results obtained in analyzing user interfaces in general. Research along these lines has been in progress for years, however some recent relevant examples and discussion may be seen in [EbH90, Fis89, FoK89, HaL90a, Har89, HiL90, HiS91, Hix89, HuL89, HuS89, JoS90, KaS89, KuB91b, Lee90, Mar90, MyG90, Mye89, NeM90, and PhV90].

Finally, software development refers precisely to the use of visualization in the process of creating and analyzing software applications and the behavior of the systems on which they execute. In a broad sense, the initial construction and specification of software applications utilizes visual programming techniques, and the subsequent analysis and modifications of the execution of the applications utilize program visualization. These two areas will be described in greater detail below after the general discussion of the overall visualization taxonomy.

After CASE, the remaining specific engineering classification is visual modeling and simulation. This area uses visual techniques to explore complex physical or theoretical environments and systems in an encompassing encapsulated manner so as to create a *Virtual Reality*, in some cases supporting interaction with more than one individual simultaneously [BlB90]. In Virtual Reality, the user is thrust into a new universe with fabricated laws of nature and behavior. The visual interface is usually accompanied by tactile input devices which allow the user to physically interact with the visual environment [Iwa90,MiO90]. There are many applications and approaches to Virtual Reality [WaK90].

Virtual Reality is used to display multi-dimensional **n-D Worlds** or environments, not normally conceivable, by using visual techniques, coloring, etc., to create an imaginary rendering. Tools such as those in the **n-Vision** system allow such exploration of n-dimensional worlds [FeB90].

Other applications allow three-dimensional shapes and volumes to be manipulated visually in **Volume Environments**. Systems of this type may be seen in [Fru91,KaY90]. These environments are sometimes enhanced with the assistance of tactile inputs such as electronic glove devices in the **GROPE** project [BrO90]. Geometric models of sculpted objects are illustrated using a similar graphics systems [DoC90,LuR90].

Often, mechanical devices and vehicles are simulated, as if interacting in a pseudo-laboratory, to imitate and approximate the **Motion & Physics** behavior of real and fabricated worlds. Tactical simulations with tanks and other moving platforms has been done in [PeM88,ZyM90]. Biomechanical motion in athletes has been analyzed [Per90]. The concept of *actors* has been used extensively along these lines for analyzing interaction of animals and human beings [MaD89,McP90]. The study of adaptation also uses such visualization for its analyses [CrS90].

Visually controlled interfaces are used to construct and sculpt forms from visual "clay" without ever getting the user’s hands "dirty" in **Virtual Sculpting**. The **ThingWorld** environment allows such sculpting with the use of tactile input, or *modal forces*, to form objects [PeE90].

Beyond the plethora of engineering applications for visualization, there are a multitude of scientific and other commercial uses. In the general area of science, there are many uses for visual representations [Wri90]. Such fields include **Computer Science**, **Mathematics**, **Medical**, **Environmental & Meteorological**, **Organic Chemistry**, **Seismology**, **Geography**, and many **Other** fields. The portion of the taxonomy for scientific uses is provided in Figure 5.

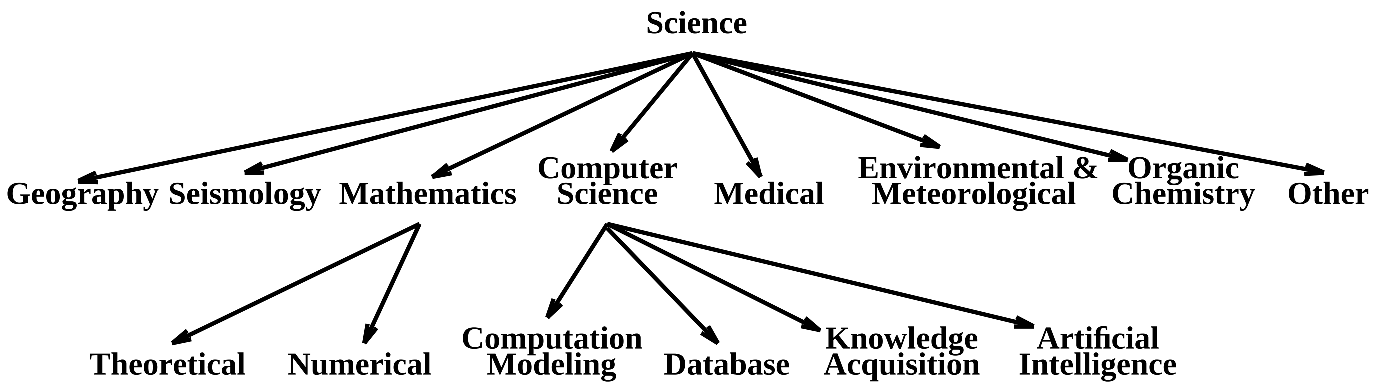

Figure 5: Scientific Visualization

In Computer Science, there are a number of applications and research areas which benefit from visualization, including **Computation Modeling**, **Databases**, **Knowledge Acquisition**, and **Artificial Intelligence**.

Computation modeling consists of abstract analyses of theoretical algorithms and computations, including visualization of finite-state machines and other graphical computation models. The **STATEMATE** system uses an abstraction called *statecharts* to analyze fundamental computations [HaL90b].

Databases use visualization as a means for representing or organizing massive or complex sets of information. Visual techniques assist in handling queries and exploration of databases, such as using graphs to explore and report on databases [Bar90], and sometimes information is retrieved through the use of visual query languages, such as the **QDB** system [AnC90] and other visual language systems [Cha90]. Databases for tool integration use visual techniques to coordinate various tools [BiL88]. Visualization is use in the gathering and retrieval of information from databases, and there are also several uses of visual interfaces for image and multimedia databases [KaK90].

Knowledge acquisition for expert systems has utilized visual techniques for decision support [HwW90], expert system construction and programming [ChF89,Sch88], and knowledge engineering and acquisition in general using visual forms [EiD90,KrU89,Mus89,StD90,TsR88]. The organization and coordination of multiple experts has been accomplished via visual knowledge acquisition techniques [Sha89]. Some knowledge acquisition tools actually support a visual language for interacting with the knowledge base [PaN89]. Knowledge acquisition tools such as **Viewpoint** assist in extracting expert experience in troubleshooting applications [CaR89]. Other such tools gather knowledge for fault diagnosis and repair using graphical interfaces [RoP89]. Knowledge acquisition for software specification has been assisted by the visually-oriented tool **PRISMA** [NiM89].

Other artificial intelligence uses include the use of visual icons in specifying logic programs in languages which use predicate forms [IcH90,Kar88]. Such visual icon programming may be found in the **HI-VISUAL** system [HiT90]. Other relational programming may be accomplished with the joint assistance of visualization and database techniques, such as in [Bra90,CeC90]. Graphical icons and interconnections are used to visually specify objects and relations [KaK91].

Another scientific field which uses visualization extensively is mathematics, both for the display and analysis of **Theoretical** shapes, surfaces, and functions [DoN90,LaM90,PiM90], and for **Numerical** plotting and visualization of data [MAT91,Nee90,Spe90]. Data is analyzed visually to assist in deriving properties [Dur90,Rac90], and collections of data are explored and emphasized using color and other visual approaches [Har90a,Hoa90,RhT90]. The concept of visual *volume probes* has been used to explore data on complex, arbitrary grids [SpK90].

Medical scientists use visualization to enhance electronic images of body parts in electromagnetic imaging systems. Color and three-dimensional graphics are applied to manipulate and emphasize characteristics in the data [Fit91].

Organic chemists use visualization as a means for assisting in computer-aided synthesis. The **PSYCHO** system is an example of such computer-aided synthesis in organic chemistry [JaL87].

Seismologists use visualization to examine characteristics of the earth’s surface. One such analysis uses visual techniques to locate oil reserves beneath the surface of the ocean [Har90b].

Geographers use visual techniques for displaying large amounts of topological data and for constructing maps, such as with the geographic information systems of [Ben89], and the visually assisted map generation of [Wil90b].

Environmental researchers and meteorologists use visualization for analyzing ecological and weather information. Characteristics of natural features are analyzed, such as a three-dimensional model of the turbulence in Lake Erie [MaK90], and weather and atmospheric systems are modeled for prediction and analysis [Mic90].

In addition to the technological uses of visualization, the business and professional community also uses visualization techniques extensively for interfacing human beings with various computer-assisted activities. Visualization may be seen in use with **Art**, **Architecture and Home Modeling**, **Civil Services**, **Legal** assistance, **Manufacturing and Production**, **Multimedia** interactions, and **Office Information and Management**, just to name a few. This classification is shown in Figure 6.

The art world uses visualization and graphics to create drawings and animations for various purposes [McW91], including commercial advertising. Visual interfaces are also used to control animation systems [HeP89,JoW89] and edit and manipulate documents which include graphic images [CoO91]. Visual techniques are used in the **SIAS** system to introduce "brush strokes"

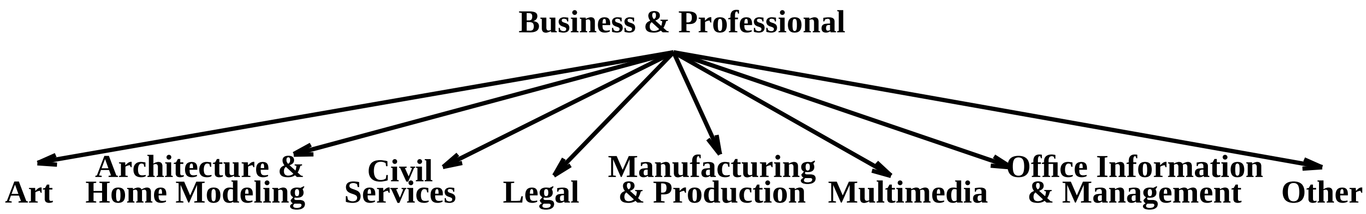

Figure 6: Business & Professional Visualization

into computer-aided animation sequences [Ber89]. Three-dimensional drawing and artwork may be created using the **3D Paint** system [Wil90a].

Architects use graphical systems to design buildings and structures, and also provide the visual renovation of historic buildings [Bla90]. Home modeling design may be accomplished entirely on the computer before hammering the first nail using visual techniques [PlP89].

Fire-fighters in the western United States are developing techniques which use visualization to display correlated infrared and visual information to help locate fire sources through thick smoke [CNN91].

The manufacturing and production world uses visualization to model the manufacturing process in analyses which seek to improve efficiency and product quality. Examples of this include the object-oriented **GAME** system [BrG91]. Visualization also serves as an interface for computer-aided productions systems which completely and automatically control production lines [JeK90].

The field of multimedia utilizes many visual interfaces in manipulating various textual, graphical, and video images. Multimedia systems are constructed using a visual interface in [HoS89], and the **SaTELLite** system allows manipulation of *hypermedia* via visual interfaces [PiT90].

Law professionals use computer graphics to assist in constructing arguments for legal cases. The **Legalese** system is an example of such a computer-aided legal system [Hai91].

The latest hi-tech office environment often uses graphics and visualization to manipulate and control office information services. An example where office procedures are visualized may be seen in [MaC90].

Visualization use has also spread into the education field, where visual techniques for organizing and providing information are being successfully applied [VoS88]. Computer graphics has been used for teaching about the structure and analysis of matrices in mathematics [PaP91]. The home education market has utilized many graphical approaches to enhance learning on the home personal computer [PlP89]. Education in developing computer software has of course been assisted, by bringing visualization into software engineering lessons [ShD88].

Beyond more serious applications of graphics and visualization, there is a very large market in visual interfaces for computer games, both in the home and in arcades [PlP89], and there are surely many other yet unexplored uses which would easily fill the miscellaneous category.

IV. B. Visual Programming & Program Visualization Taxonomy

Given the context of parallel software visualization in the overall general visualization taxonomy presented above, a more detailed examination of the classification of program visualization and visual programming systems will now be provided. This classification is based on the collection of systems discussed in the previous section, and draws from the grouping there to produce a taxonomy for each area. Minimal references will be listed in this portion of the section for the mainstream visual programming and program visualization tools. Rather, a table will be provided at the end of the section which lists all relevant tools with their taxonomical and quantitative classifications.

It should be noted that these taxonomies are not the most detailed possible, in general, and do not include much subdivision of classes from the highest levels. Given the current state of the visualization field with respect to these two areas in software engineering, any further distinction on a more detailed level would be necessarily incomplete. The only well-defined aspects of such tools beyond the provided taxonomy would be based on the individual tools' capabilities and perspectives, which are at best currently not formally understood nor encompassing enough to produce a full partitioning. It is the opinion of the author in completing this examination that the provided taxonomy and quantitative metrics are sufficient to classify the field, even with little more than the single level of classification. It is hoped that as research in this area proceeds, the current taxonomy may be expanded in more detail as the important metrics and characteristics become better understood.

For visual programming, the taxonomy is very general and will not be explored in great depth. Visual programming systems may be broken into several categories including **Program Creation**, **Organization** and **Manipulation**, as seen in Figure 7.

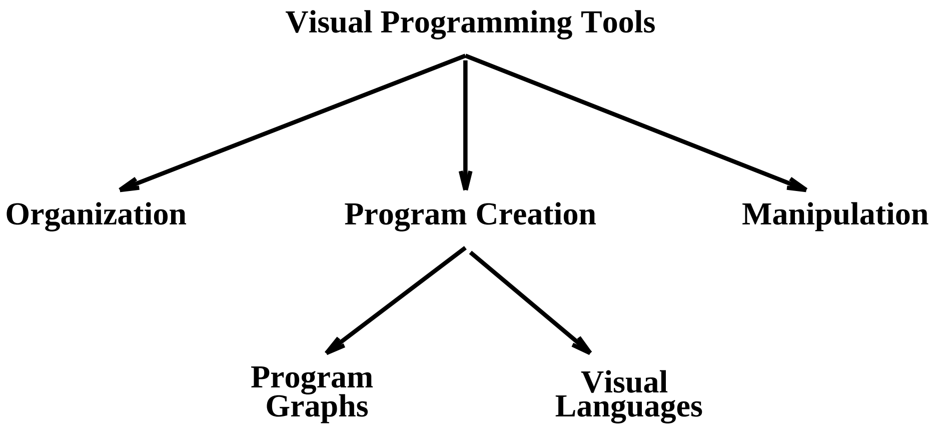

Figure 7: Visual Programming Taxonomy

Program creation refers to the general task of assembling program source code and partitioning and mapping it into executable programs, for which a majority of the visual programming tools have been constructed. In this category, tools may be separated into two groups based on the approach to utilizing graphics in the programming language. One group emphasizes visual call-graph or data-dependency graph specification with textual program modules or code segments, or **Program Graphs**. The other group goes further to actually build programs from completely or mostly visual languages, such that the visual representation implicitly includes the semantics of the programs, or **Visual Languages**.

Organization with visual programming refers to the graphical assembling of systems or information for various purposes. Examples of this include tool integration [Rei90], file system security specification [HeM90], and other software organizational functions, such as process management and interconnection [BhC88]. Other tools which fall into this category include system design tools such as **Microtool**, which visually assists in creating microcode for microprocessors [ElM91].

Manipulation refers to the visual control or manipulation of operations, functions, programs, etc. There have been visual programming tools for specifying and manipulating software scheduling [ChM89b,DeG90] as well as many other aspects of program execution. In fact, the visual interfaces for all program visualization tools can in many ways be considered examples of visual programming.

Program visualization is classified in a more detailed manner, involving several categories and methods of distinction among systems. On a qualitative level, program visualization tools may be separated based on their applicability to the software engineering design cycle. As previously discussed in section III, the various groups include **Problem Analysis** and **Algorithm Analysis**, implemented **Executable Program** analysis, from **Application-Oriented** or **System-Oriented** perspectives, or via visual display of various **Statistics**, and finally a separate **General** group which applies equally well over all stages of the design cycle. This program visualization taxonomy is shown in Figure 8.

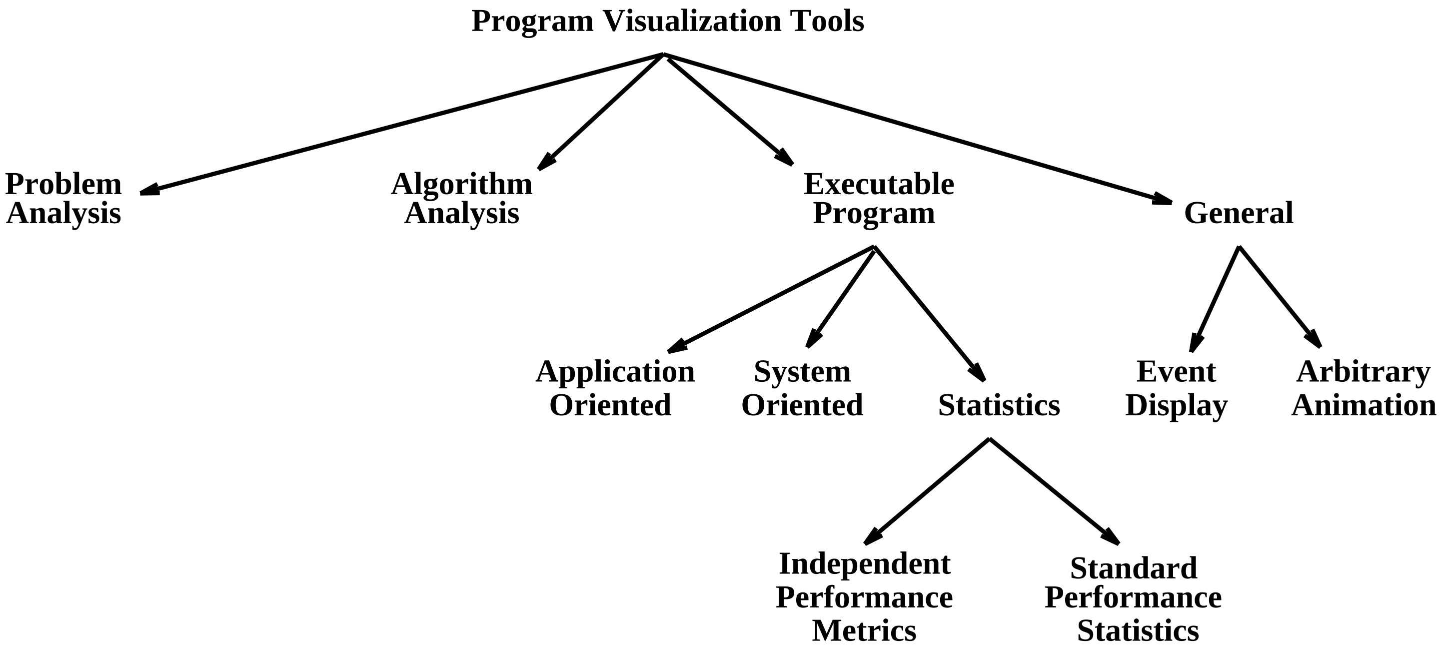

Figure 8: Program Visualization Taxonomy

Program visualization for problem analyses refers to the display of problem specifications or properties prior to the development of an algorithm or the implementation of an executable program. These analyses strive to understand properties or characteristics of a problem to allow better algorithm design and implementation. An example of this type of analysis would be visualizing simple images of missionaries and cannibals riding a boat back and forth across a river to understand the missionaries & cannibals problem, or perhaps manipulating a collection of labeled visual images in preparation for developing a sorting algorithm. Currently, no work has been specifically done in this area aside from generally applicable visualization tools, (note the unbalanced nature of the taxonomy which implies that most work and understanding in this area deals with executable programs), but the classification is clearly worth distinguishing.

Visual algorithm analysis takes a small but significant step away from problem analyses by including an automated approach which actually manipulates elements of the problem. Aside from just giving the user a bunch of problem element "building blocks" to manipulate interactively, the algorithm analysis would actually propose a method for "stacking" them and visually display it. For example, where problem analysis would present a visual representation for manipulating images for a sorting problem, algorithm analysis would actually act out a particular sorting algorithm using abstract operations and visual animation to move the images around and sort them. These analyses are strictly generated only from algorithm simulations, and no actual program execution traces are used.

The next set of the remaining categories for program visualization apply specifically to implemented software systems which are in the process of debugging, performance tuning, or revision. These three categories distinguish among the different perspectives taken towards implemented software analysis, and are application-oriented, system-oriented, and independent performance metrics and statistics.

Application-oriented visual analyses refers to those analyses which emphasize the higher-level characteristics of the application's software components. Specifically, the analysis is from the perspective of the typical programmer, who creates a program using various software constructs and then desires to visualize the behavior and interaction of those constructs. Visual displays of this type usually depict communicating processes or tasks, data structures, and other high-level or abstract program characteristics and entities such as those viewed in algorithm analyses. These analyses all are strongly related to actual program executions, and the visual displays are typically driven by a sequence of traced events from an instrumented program.

A lower-level perspective of software behavior is provided by system-oriented visual tools, which are those tools that focus on the actions or reactions of the target system architecture in response to or in conjunction with the software application. These tools provide information from the systems designer's perspective and visualize the hardware and software components which together constitute the physical computer system and the operating system. Visual displays of this type typically animate communication networks, processor/memory architectures and topologies, and abstract renderings of memory accesses, processor status, or operating system functions such as synchronization with semaphores and message queues.

Another perspective of software behavior is seen by the visualization of various numerical statistics. These statistics fall into two distinct sub-classes, **Independent Performance Metrics** and **Standard Performance Statistics**.

The independent performance metrics are those metrics which are constructed from the relation or mapping of user application to system architecture. These metrics combine separate characterizations of an application and an architecture into a single independent rating which represents the quality of the mapping for the given pair. Especially for parallel and distributed systems, there is no straightforward means for measuring and comparing the behavior of applications on different architectures except by separating the analysis in this way to produce an independent metric. Examples of such metrics include [JaG87a,Mar87,PeG91].

The standard performance statistics are much simpler in nature and refer to any and all numerical measures which have been applied to any aspect of an executing parallel or sequential program. These statistics include everything from processor load, utilization and idle time to message queue length and message traffic, network contention, concurrency profiles, plus MIPS, MFLOPS, Whetstones [CuW76], and Dhrystones [Wei84]. There are several tools which produce displays of this type, as mentioned in Section III.

Both the independent metrics and performance statistics are displayed using a large variety of static and dynamic visual forms and diagrams. Examples include animated Kiviat diagrams, bar charts, histograms, graphs, numeric plots, etc.

The last general category of the program visualization taxonomy encompasses those tools with global applicability to all phases of software engineering and all perspectives on analyzing problems, algorithms, and executing programs. These tools apply equally well in all these areas, but in two significantly different ways which divide the category into two groups, **Event Display** and **Arbitrary Animation**.

The event display tools are extremely simple and provide direct visualization of trace events from program execution histories. The displays are all automatic, aside from various user options, and strive only to provide a more organized, literal display of raw events using graphics. These tools display multiple rows of lines which represent collections of processes or processors in a parallel software application. Events are visualized as colored or textured bars along these lines, resulting in a static display of each process or processor history along the time axis. By lining up the whole set of lines on a common time axis, the synchronization and relative status of the processes or processors are intuitively compared. This straightforward method proves powerful across all types of analyses by providing an organized display of any event sequence, whether user-defined or built-in, and whether from software instrumentation, hardware monitoring, or synthetic simulations.

The arbitrary animation tools are at the other end of the complexity spectrum, and provide the most sophisticated, flexible, and abstract visual animations. These tools often start with user-defined visual models and then construct arbitrarily complex or intricate animations around them. The results can be used to examine the behavior of software project entities along the entire software engineering cycle due to the flexibility in visual display.

The animations typically center around graphical objects and images which can be constructed modularly and hierarchically to individually represent and encapsulate the functionality of each portion of the desired entity's behavior. The objects and images need not correspond to physical entities or characteristics, and can hence emulate conceptual or fabricated behavior as well. The objects or images are combined together to form a complete model, and can move around, change their appearance, or interact with each other to animate and visualize the desired activity. These arbitrary animations are driven by event sequences which indicate the particular behavior of the target entity.

IV. C. Quantitative Metrics

Aside from the qualitative classification system for program visualization tools given above, there is also a need for more concrete evaluation of such visualization tools to allow more detailed comparisons within categories and across categories. These metrics transcend the tree structure of the taxonomical system to introduce another dimension of categorization. (In fact, given an appropriate visualization package, these metrics could be added to the taxonomy above to produce a three-dimensional classification system, with the quantitative metrics branching out of the page in a perpendicular plane.)

Four such metrics have been constructed for quantitatively distinguishing visualization tools for parallel software development. These metrics are **Scalability**, **Visualization Degree**, **Graphical Sophistication**, and **Focusing Capability**. Each of these metrics takes a different form in quantifying visualization tools, but can combine together to produce a compound metric for comparing tools.

Scalability refers to the extent to which the visual displays of a tool apply in displaying detailed information for the number of processors or processes in a parallel or distributed computer architecture. In other words, the Scalability of a tool is the maximum number of processors, processes, or other concurrent entities which can be simultaneously displayed in a single analysis. For example, a system which only visualizes sequential software has a Scalability proportional to one, where a tool which visualizes a parallel system with one million processors has a Scalability proportional to one million.

To provide a more reasonable categorization, the Scalability metric has been narrowed down to a small collection of ranges rather than using an infinite scale. The resulting Scalability metric values are summarized in Table 1 below versus number of processors. Note that the same values may be applied to numbers of processes or other visualization entities.

Scalability	
I	Uniprocessor
II	Up to 16 processors
III	Up to 32 processors
IV	Up to 128 processors
V	Up to 1024 processors
VI	Up to 16K processors
VII	Up to 64K processors
VIII	Up to 256K processors
IX	Up to 1M processors
X	Over 1M processors

Table 1: Scalability Metric

The Visualization Degree of a tool represents the number of dimensions of data and information which can effectively be visualized using the tool's approach and capabilities. This metric is quite literal and evaluates use of the available visualization dimensions in a workstation environment. Given existing technology, there are a total of 10 possible dimensions.

There are three physical dimensions obtained in 3-D monotone graphics, where only solid lines are used to draw three-dimensional shapes and images. Time may be considered a fourth dimension, as some tools produce displays without reference to time by utilizing only aggregate statistics or summaries.

Color and intensity of color count as the fifth and sixth dimensions, as the entire spectrum of color may clearly represent a single dimension of data, and the level of intensity of color or gray-scale may be used in some cases to reveal another dimension. Audio pitch and intensity are the seventh and eighth dimensions, for similar reasons to color and color intensity.

The ninth dimension is a special case of the three physical dimensions and is chosen to be image dynamics, meaning the capability of an individual graphic or visual object to change its shape or form during animation. This ninth dimension is counted separately from the three physical dimensions because it refers to local visual activity as opposed to the overall capability to draw images. The last dimension is obtained from textual labeling or notation in the animation. The use of text as a dimension is also in some ways a special case of general drawing capability, but has special significance due to its obvious semantic features.

As an example to illustrate the identification of this metric, consider a tools which produces two-dimensional animated black-and-white views of static icon images with no audio but with text. Such a tool would have two dimensions for the display area, one for time, and one for text resulting in a Visualization Degree of 4.

The Graphical Sophistication metric attempts to evaluate the extent to which graphics has been effectively utilized to visualize information. When combined with the degree of visualization, this metric will reveal precisely how well the graphical dimensions have been exploited.

The varying levels of Graphical Sophistication have been quantized into ten specific values of increasing sophistication, flexibility and abstraction, as shown in Table 2.

The lowest level of sophistication is clearly text-based, as this has extremely limited ability to display graphical information. Above that level at level II, numeric plots provide a somewhat more pleasing rendering of data, but with little creativity or sophistication.

Graphical Sophistication	
I	Text-Based
II	Numeric Plots
III	Charts, Bar Graphs, and Histograms
IV	Gantt, Kiviat, and Space-Time
V	Program Graphs
VI	Animated Diagrams - Fixed Images, Immobile
VII	Animated Diagrams - User-Defined Images, Immobile
VIII	Animated Diagrams - Fixed Images, Mobile
IX	Animated Diagrams - User-Defined Images, Mobile
X	Full, Continuous, Pure Animation

Table 2: Graphical Sophistication Metric

At level III, the standard types of pie charts, bar graphs, and other simple data displays allow some flexibility in animation possibilities, but still use rather simple forms. The level IV diagrams introduce more complex visual forms for displaying data which can utilize colors and intensity more effectively than level III diagrams.

The level V program graphs include the more abstract call-graph and data-dependency graph representations used most often by visual programming tools. These displays use a dynamic and more abstract technique for visualizing information on program executions.

Starting at level VI, simple animations are used to show interactions among objects in the display, but at this level the icons images used are still pre-selected in the tool for specific visual models and are not capable of moving around the display. Level VII empowers the user to define custom images to allow representation of more than a single model perspective, but the icon images remain immobile.

At levels VIII and IX, the animated icon image displays of levels VI and VII are enhanced to allow mobile icon images which may move around the display to model behavior. Finally, at level X, the graphics are the most sophisticated and become complete, smooth animations with dynamic images and movement around the display area. At this highest level, the animations are more realistic in nature and proceed as elaborate cartoon productions.

Finally, the last quantitative metric, Focusing Capability, refers to the ability of the tool to control or manipulate the level of detail in its visualization displays. This manipulation or control can apply to visualizations in several different ways of varying complexity and benefit, and a given tool may possess none, one, or more than one of the different capabilities. The list of focusing operations are listed in Table 3 below, weighted according to their value in focusing and their complexity in implementation.

Focusing Capability	
0	None
1	Display / Animation Speed
2	Reverse Tracing / Animation
3	Image Size / Data Compression
4	Variable Abstraction Level

Table 3: Focusing Capability Metric

Clearly, having no capability for focusing whatsoever is assigned a weight of zero. The first and most fundamental focusing operation involves the adjust and control the speed of the visual display or animation. This capability can provide a temporal focusing on the detail of a display by slowing down the graphics to allow close examination. In contrast, speeding up the display allows skimming over unimportant details or defocusing, either for overall examinations or to move through a trace to a section of interest. Often, defocusing is necessary because it is not always possible to completely skip over uninteresting trace events, as the display state of interest may depend on the entire sequence of events which precedes it.

The next operation, of reverse tracing or playing animations backwards, is a potentially useful feature in conjunction with the variable speed capability. When tracing forward at high speed, it is certainly likely that a display of interest might appear quite suddenly and then disappear before the user can stop or slow down the display to view it. Being able to stop the tracing and rewind back to the display of interest could save much time by avoiding a complete replay to return to the missed incident. In addition, reverse playback could be generally useful by allowing flexible tracing control for the repeated viewing of interesting displays, perhaps for a multiple-perspective analysis using different displays for a single portion of the trace.

The third capability, focusing through increasing image size, allows close, detailed examination of individual graphic images. By enlarging a particular object, or "zooming in", the details of interest captivate the user’s perspective, and other details are pushed out of view. The opposite capability to compress images and data becomes useful in a different application of focusing which emphasizes the extraction of anomalies or atypical behavior. By compressing image size, or "zooming out", an overall, general view results in which any differences among the graphical objects will draw the user’s attention. Data compression provides the same features as image size compression, but also introduces some added organization. Data compression involves combining or merging multiple graphical objects with similar characteristics or behavior into a single graphical object. The result is that the behavior of the overall visual model is organized into "equivalence classes" of behavior, in which any differences will explicitly appear.

The last capability, manipulation of the level of abstraction, is the most sophisticated form of focusing. In a system where visual objects are constructed hierarchically, the display may be adjusted to view all or part of the visual model from any of the abstraction levels of the hierarchy. This "zooming" capability transcends the mechanical, graphical interpretation and provides manipulation at the conceptual level.

These four Focusing Capabilities, with their weighted values, combine together for a given tool to create a single numerical value for the Focusing Capability Metric. This value is constructed by adding the appropriate weights of the capabilities which the tool supports. The resulting metric falls in a range from zero to ten, and represents the overall quality of the tool’s ability to focus its visualization. For example, if a tool supports variable speed in animation and data compression, its overall Focusing Capability is one plus three which equals 4.

Now, to complete the quantitative analysis, note that there are a total of four quantitative metrics which each produce numerical values in the range from one to ten (with the exception of a possible zero in Focusing Capability). To produce an overall quantitative evaluation of a particular tool, these four metrics can be combined. There are two separate methods for combining these metrics with two slightly different results - **Overall Average** and **Overall Deviation**.

The Overall Average for a tool is simply the numerical average of the four metrics, computed by adding the values together and dividing by 4. This quantity reveals the combined effect of the various quantitative capabilities or features, and eludes to how "good" the typical benefit might be in using the tool.

The Overall Deviation for a tool is computed as the standard deviation of the four quantitative metrics, precisely the sum of the square roots of the differences between each metric value and the Overall Average.

$$Overall\ \ Deviation\ =\ OD\ =\ \sqrt{(S-OA)^2+(VD-OA)^2+(GS-OA)^2+(FC-OA)^2}$$

This value reveals to the user the extent to which a given tool is balanced in its capabilities. A large value for Overall Deviation implies that a tool is severely lacking in certain capabilities, but stronger in others. A small deviation value implies a consistent, well-balanced tool, and in this case the Overall Average may be used to evaluate precisely at what level the tool provides assistance.

To further describe the above quantitative metrics and the taxonomy for visual programming and program visualization, refer to the tool annotation in Tables 4 and 5, respectively, which describe how each tool fits into its taxonomy, and what its values are for the quantitative metrics.

Visual Programming	
Taxonomy Classification	Tool
Program Graphs	CODE/ROPE
	SCHEDULE
	Poker
	GRAPE
	E/SP
	Start/Pat
	PRETS
	Navier-Stokes
Visual Languages	XPAT
	VERDI
	Graspin
	VPW
	PPSE
	PECAN
	PegaSys
	Omega
	PiP

Table 4: Visual Programming Tool Annotation

Program Visualization							
Tool	Taxonomy	S	VD	GS	FC	OA	OD
BALSA / BALSA-II	Algorithm Analysis	5	6	6	8	6.25	2.18
Smalltalk	Algorithm Analysis	4	5	9	1	4.75	5.72
"Program Visualization"	Algorithm Analysis	2	5	9	4	5.00	5.10
Incense	Executable / Application	2	5	7	0	3.50	5.39
VIPS	Executable / Application	5	4	6	4	4.75	1.66
IPS-2	Executable / Application	2	5	5	0	3.00	4.24
Faust	Executable / Application	3	5	5	6	4.75	2.18
PIE	Executable / Application	3	5	5	5	4.50	1.73
Gauge	Executable / Application	4	3	3	3	3.25	0.87
Ravel	Executable / Application	3	5	5	1	3.50	3.32
TupleScope	Executable / Application	2	4	6	0	3.00	4.47
Axe	Executable / System	3	5	3	0	2.75	3.57
Simple / Care	Executable / System	4	4	3	0	2.75	3.28
PARET	Executable / System	4	6	5	4	4.75	1.66
DV-DRAW	Executable / System	2	5	7	0	3.50	5.39
Q+	Executable / System	4	5	9	1	4.75	5.72
Gecko	Executable / System	3	4	6	6	4.75	2.60
TOPSYS	Executable / System	2	5	8	1	4.00	5.48

Table 5: Program Visualization Tool Annotation (Page 1 of 2)

Program Visualization							
Tool	Taxonomy	S	VD	GS	FC	OA	OD
Multiple Views	Executable / System	3	5	6	1	3.75	3.84
SHMAP	Executable / System	2	5	6	1	3.50	4.12
Pictures of Performance	Executable / System	7	6	6	3	5.5	3.00
TMP	Executable / System	2	4	6	5	4.25	2.96
PAWS	Statistics / Independent	1	3	6	0	2.50	4.58
ParaGraph	Statistics / Standard	4	5	4	3	4.00	1.41
HyperView	Statistics / Standard	4	6	4	0	3.50	4.36
Preface	Statistics / Standard	4	4	3	0	2.75	3.28
Upshot	General / Event Display	4	3	4	6	4.25	2.18
PM	General / Event Display	2	3	4	6	3.75	2.96
Jed	General / Event Display	3	3	7	6	4.75	3.57
Voyeur	General / Arbitrary Animation	4	6	9	0	4.75	6.54
Tango	General / Arbitrary Animation	2	5	9	3	4.75	5.36
Triplex	General / Arbitrary Animation	4	7	6	7	6.00	2.45
PARADISE	General / Arbitrary Animation	4	4	7	5	5.00	2.45
Macaw	General / Arbitrary Animation	3	4	7	1	3.75	4.33

Table 5: Program Visualization Tool Annotation (Page 2 of 2)

V. Conclusions & Future Work

It is clear that visualization has proliferated into many fields for many purposes, and is growing in its usage and popularity as a means for simplifying the potentially complex interactions of humans and computer systems. Specifically in the area of research in parallel computing, visualization has already proven useful, and it is gaining a larger following as better and better tools are introduced.

Still, the field of visualization for parallel software development is in many ways a new and, as yet, much unexplored field. Though several dozen visualization tools have been created and applied to visual programming and program visualization over the past decade, there is still little understanding of the fundamental characteristics or properties of visualization, and no general agreement has been reached as to what the best visual representations and approaches are for parallel software development.

In many ways, these issues will remain unanswered until other areas are better understood, such as the functioning of the human brain and its interface to the outside world, and the nature and behavior of parallel software. But these areas are strongly integrated with and connected to visualization, and it appears wise to examine all of these issues together in striving to truly understand each of them.

The important questions at this point are then precisely, "What needs to be done?" and "Where can efforts best be applied?". Some of the more crucial issues faced in visualizing parallel software can be identified most clearly by examining the very means by which visualization tools are critiqued. Specifically, the quantitative characteristics described above concerning the capabilities of tools may be used to look back at the current state of the field and suggest areas for improvement.

Along these lines, scalability is of major concern. There is a growing need to deal with larger and larger multiprocessor systems, and there are some severe limits posed by current graphics technology, not to mention human perception. There is a limit to how much information can be displayed and absorbed from a single screen.

Clearly, in some cases it may suffice to use visualization only on small scale versions of parallel software, to develop and tune the code on that level and then abandon visualization for the fully-scaled parallel versions. However, there are likely parallel algorithms which will not support such scaled-down development.

Currently, few visualization tools handle very large numbers of processors, and there is no accepted method for representing such large systems. There are some current techniques which include compression or "binning" of information [CoK90,Nic90], and the application of scientific visualization techniques [Rov89,RoW90,RoC91,Rov91], but there is a need for more experimentation and other new creative approaches to dealing with this problem.

Beyond scalability, the next most lacking area involves use of the various visualization dimensions or the Visualization Degree. Visualization tools for parallel software have just begun using three-dimensional representations, and have barely explored the uses of the other dimensions, including color, color intensity, and local image dynamics. Use of audio has been discussed [GaS91,WeF90] but not experimented with at all in this context. For all of these dimensions, new and continued explorations are needed to identify their value and properties. In addition, a standard and general approach for specifying the use of each dimension is needed to support continued and widespread use. This issue will be discussed in more detail below.

Another general area which needs improvement relates to the use of more sophisticated graphics techniques and capabilities. Many tools do not take advantage of the available techniques. The solution to this problem is clearly within reach at the present time, and requires only time and effort on the part of tool designers to apply the available techniques. Especially with some of the focusing capabilities, the only obstruction is the amount and complexity of the source code necessary for implementation. With the more sophisticated graphics techniques, it is primarily a matter of graphics programming and the available graphics technology. Though visualization tool creation may become a more difficult task by including extra capabilities and graphics sophistication, this task need only be completed on a one-time basis, and the benefits are enjoyed every time the tool is utilized.

However, if the development of any single tool is involved and complex, it becomes even more important to make each tool extremely flexible and useful, to avoid wasted effort. More important than simply adding the extra "bells and whistles" to visualization tools, it is necessary to provide an easy and efficient interface or method for specifying and constructing arbitrary or custom visual displays. To date, the tools which provide arbitrarily flexible display techniques also require an arbitrarily complex development to use those displays. A standard, powerful specification method is needed for these displays. The resulting approach must efficiently control and utilize all of the visualization dimensions, and the user must be able to quickly and easily create powerful, custom, visual representations for all kinds of data.

The typical engineering approach to this problem is to provide a layered or hierarchical specification system which supports well-defined interfaces between its layers. Each layer would be responsible for handling a particular portion of the visualization process, from monitoring and data collection to data processing, filtering and display. Using this approach, layers of the system could easily be modified and upgraded with changes in graphics technology and parallel software approaches.

The top layer of this layered specification system should be a graphically oriented interface to the user, and should provide fast and efficient construction of arbitrary visual displays. In addition, this interface should be capable of initiating high-level control of the lower layers which control data collection and processing. To allow integration of various kinds of information, raw events must be accessible from all sources including execution histories, simulators, and synthetic generators. There must be a highly flexible system of filters and statistical analyzers for channeling and manipulating the raw data to produce meaningful information for display.

The key focus of the system must be to expedite use of visualization systems to make visualization a more feasible alternative to everyday program development. Unless the process of utilizing visual analysis techniques is simple and straightforward, users will only consider visualization a last resort.

Work has been done towards achieving more efficient and friendly user-interfaces, but the work done specifically for parallel software development has been mostly special-purpose. Other work done in computer-human interaction and interface research is often too general to completely satisfy the specific needs of the parallel programmer. Even the task of integrating the different views and visual systems is still not easily accomplished [Mey91]. A substantial effort must be applied to bring visualization to a point where it can be used more effectively to analyze and understand parallel software.

Finally, work in the visualization field must expand to assist in more phases of the software engineering cycle. There is little to no effort currently being applied to problem analysis, and the other areas are still in their early stages as far as providing the necessary assistance for algorithm design, program creation, and performance tuning and debugging.

Aside from simply producing more graphically sophisticated tools with more efficient user-interfaces, effort needs to be applied towards exploring new perspectives and approaches to using visualization in parallel software development. There are surely new visual representations to be created for viewing parallel systems in different ways. The designers of visualization tools will need to venture through these unknown territories and strive to utilize the full benefits of intuitive, visual representation.

References

[AnC90] Michele Angelaccio, Tiziana Catarci, and Guiseppe Santucci, "QBD*: A Graphical Query Language with Recursion," IEEE Transactions on Software Engineering, Volume 16, Number 10, October 1990, pp. 1150-1163.

[AnP89] Per Andersson, and Lars Philipson, "Movie - An Interactive Environment for Silicon Compilation Tools," IEEE Transactions on Computer-Aided Design, Volume 8, Number 6, June 1989, pp. 693-701.

[ApS89] Bill Applebe and Kevin Smith, "Start/Pat: A Parallel-Programming Toolkit," IEEE Software, Volume 6, Number 7, July 1989, pp. 29-38.

[BaB90] R. Barth, R. Böse, and I. Heilemann, "UGRAF3: A Graphic System for Process and Modeling," Computers & Graphics, Volume 14, Numbers 3/4, 1990, pp. 429-433.

[BaS88] M. L. Bailey, D. Socha, D. Notkin, "Debugging Parallel Programs using Graphical Views," *Proceedings of the 1988 International Conference on Parallel Processing*, Volume II, St. Charles, Illinois, August 1988, pp. 46-49.

[Bar90] Robert Barr, "Using Graphs to Explore Databases and Create Reports," ACM SIGCHI Bulletin, Volume 22, Number 1, July 1990, pp. 24-27.

[BeA90] H. Berger, and J. Altenbach, "Graphical Representation of Models and Results in the Finite Element System COSAR," Computers & Graphics, Volume 14, Numbers 3/4, 1990, pp. 395-404.

[BeB89] David Bernstein, Anthony Bolmarcich, and Kimming So, "Performance Visualization of Parallel Programs on a Shared Memory Multiprocessor System," *Proceedings International Conference on Parallel Processing*, St. Charles, Illinois, August 1989, Volume II, pp. 1-10.

[BeB90] Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Peter Luksch, and Roland Wismüller, "TOPSYS - Tools for Parallel Systems," Users's Overview and User's Manuals, Technische Universität München, Institut Für Informatik, TUM-19047, December 1990.

[BeB91] Thomas Bemmerl, Peter Braun, Olav Hansen, Thomas Treml, and Roland Wismüller, "The Design and Implementation of TOPSYS," Version 1.0 Release Notes, Technische Universität München, Institut Für Informatik, May 1991.

[BeC90] Paul Bercovitz and Nicholas Carriero, "TupleScope: A Graphical Monitor and Debugger for Linda-Based Parallel Programs," Research Report YALEU/DCS/RR-782, Department of Computer Science, Yale University, April 1990.

[BeE88] Thomas Bemmerl, Norbert Erl, Olav Hansen, "Menu and Graphic Driven Human Interfaces for High Level Debuggers," Microprocessing and Microprogramming, The Euromicro Journal, Volume 24, 1988, pp. 153-160.

[Bem88] Thomas Bemmerl, "An Integrated and Portable Tool Environment for Parallel Computers," *Proceedings of the 1988 International Conference on Parallel Processing*, Volume II, St. Charles, Illinois, August 1988, pp. 50-53.

[Ben89] Frank Bennion, "Geographic Information Systems," The Computer Bulletin, Volume 1, Part 5, June 1989, pp. 12-16.

[Ber89] Pierre van Berkel, "SIAS, Strokes Interpreted Animated Sequences," Computer Graphics Forum, Volume 8, 1989, pp. 35-47.

[BhC88] Mitali Bhattacharyya, David Cohrs, and Barton Miller, "A Visual Process Connector for Unix," IEEE Software, July 1988, pp. 43-50.

[BiL88] Roberto Bisiani, Francois Lecouat, and Vincenzo Ambriola, "A Tool to Coordinate Tools," IEEE Software, November 1988, pp. 17-25.

[BlB90] Chuck Blanchard, Scott Burgess, Young Harvill, Jaron Lanier, Ann Lasko, Mark Oberman, and Micheal Teitel, "Reality Built For Two: A Virtual Reality Tool," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 35-36.

[Bla90] Vanessa Blake, "Ken Delooze: Computer Aided Design," The Computer Bulletin, Volume 2, Part 10, December 1990, pp. 12-14.

[BrA89] J. C. Browne, Muhammad Azam, and Stephen Sobek, "CODE: A Unified Approach to Parallel Programming," IEEE Software, Volume 6, Number 7, July 1989, pp. 10-18.

[BrC85] Gretchen P. Brown, Richard T. Carling, Christopher F. Herot, David A. Kramlich, Paul Souza, "Program Visualization: Graphical Support for Software Development," IEEE Computer, Volume 18, Number 5, August 1985, pp. 27-35.

[BrD88] O. Brewer, J. Dongarra, D. Sorenson, "Tools to Aid in the Analysis of Memory Access Patterns for Fortran Programs," Mathematics and Computer Science Division, Argonne National Laboratory, Technical Memorandum Number 120.

[BrG91] Daniel Breugnot, Michel Gourgand, David Hill, and Patrick Kellert, "GAME: An Object-Oriented Approach to Computer Animation in Flexible Manufacturing System Modeling," Simulation Digest, Volume 21, Number 3, April 1991, pp. 217-227.

[BrL90] James C. Browne, Taejae Lee, and John Werth, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment," IEEE Transactions on Software Engineering, Volume 16, Number 2, February 1990, pp. 111-120.

[BrO90] Frederick P. Brooks, Jr., Ming Ouh-Young, James J. Batter, and P. Jerome, "Project GROPE - Haptic Displays for Scientific Visualization," *Proceedings of SIGGRAPH ’90*, Dallas, August 6-10, 1990, (Computer Graphics, Volume 24, Numbers 4, pp. 177-185).

[BrS84] Marc H. Brown, "A System for Algorithm Animation," Computer Graphics, Volume 18, Number 3, July 1984, pp. 177-186.

[BrS85] Marc H. Brown and Robert Sedgewick, "Techniques for Algorithm Animation," IEEE Software, Volume 2, Number 1, January 1985, pp. 28-39.

[BrS89] J. C. Browne, K. Sridharan, J. Kiall, and C. Denton, "Parallel Structuring of Real-Time Simulation Programs," Technical Paper, Scientific and Engineering Software, Inc. and Concurrent Computer Corporation, 1989.

[Bra90] Roger Brackin, "Tools for Graphical Application Building with Relational Databases," The Computer Bulletin, Volume 2, Part 9, November 1990, pp. 9-11.

[Bro88] Marc H. Brown, "Exploring Algorithms Using BALSA-II," IEEE Computer, Volume 21, Number 5, May 1988, pp. 14-36.

[BuJ89] Donald L. Burkhard and Per V. Jenster, "Applications of Computer-Aided Software Engineering Tools: Survey of Current and Prospective Users," Data Base, Volume 20, Number 3, Fall 1989, pp. 28-37.

[CNN91] News story on CNN News channel on cable television, Tuesday, July 23, 1991.

[CaR89] J. E. Caviedes, and M. K. Reed, "ViewPoint: A Troubleshooting-Specific Knowledge Acquisition Tool," SIGART Newsletter, Knowledge Acquisition Special Issue, April 1989, Number 108, pp. 155.

[CeC90] Stefano Ceri, Stefano Crespi-Reghizzi, Robert Zicari, Gianfranco Lamperti, and Luigi A. Lavazza, "Algres: An Advanced Database System for Complex Applications," IEEE Software, Volume 7, Number 7, July 1990, pp. 68-78.

[Cen90] Zoltan J. Cendes, "EM Simulators = CAE Tools," IEEE Spectrum, Volume 27, Number 11, November 1990, pp. 73-93.

[ChF89] M. Alessandra Cherubini, Laura Fanti, Piero Torrigliani, and Massimo Zallocco, "An Integrated Expert-System Builder," IEEE Software, Volume 6, Number 11, November 1989, pp. 44-52.

[ChM89a] Jue-Hsien Chern, John T. Maeda, Lawrence A. Arledge, Jr., and Ping Yang, "SIERRA: A 3-D Device Simulator for Reliability Modeling," IEEE Transactions on Computer Aided-Design, Volume 8, Number 5, May 1989, pp. 516-527.

[ChM89b] ByoungJu Choi, Aditya P. Mathur, and Brian Pattison, "PMothra: Scheduling Mutants for Execution on a Hypercube," ACM Software Engineering Notes, Volume 14, Number 8, December 1989, Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification (TAV3), pp. 58-65.

[ChN89] Minder Chen, Jay F. Nunamaker Jr., and E. Sue Weber, "Computer-Aided Software Engineering: Present Status and Future Directions," Data Base, Volume 20, Number 1, Spring 1989, pp. 7-13.

[ChP90] J. C. H. Chung, D. R. Patel, and R. L. Cook, "Feature-Based Modeling for Mechanical Design," Computers & Graphics, Volume 14, Number 2, 1990, pp. 189-199.

[Cha90] Shi-Kuo Chang, "A Visual Language Compiler for Information Retrieval by Visual Reasoning," IEEE Transactions on Software Engineering, Volume 16, Number 10, October 1990, pp. 1136-1149.

[CoK90] Alva L. Couch and David W. Krumme, "Monitoring Parallel Executions in Real Time," To appear in *Proceedings Fifth Distributed Memory Computing Conference, IEEE Computer Society Press, 1990.*

[CoO91] Ahmet Cosar and Bulent Ozguc, "A Text, Image, and Graphics Editor," Computers & Graphics, Volume 15, Number 1, 1991, pp. 57-66.

[CrS90] Robin Huw Crompton and William Sellers, "3-D Computer Graphics in the Study of Adaptation," The Computer Bulletin, Volume 2, Part 5, June 1990, pp. 10-13.

[CuW76] H. J. Curnow, B. A. Wichman, "A Synthetic Benchmark," *Computer Journal*, Volume 19, Number 1, February, 1976.

[DeG90] Wolfgang Deiters and Volker Gruhn, "Managing Software Processes in the Environment MELMAC," ACM Software Engineering Notes, Volume 15, Number 6, December 1990, Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments, pp. 193-205.

[DiM90] Chen Ding and Prabhaker Mateti, "A Framework for the Automated Drawing of Data Structure Diagrams," IEEE Transactions on Software Engineering, Volume 16, Number 5, May 1990, pp. 543-557.

[Dip90] Stephen M. Dipietro, "A Graphical User Interface to Support a Production Team," ACM SIGCHI Bulletin, Volume 22, Number 1, July 1990, pp. 36-39.

[DoB90] J. Dongarra, O. Brewer, J. A. Kohl, S. Fineberg, "A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors," *The Journal of Parallel and Distributed Computing*, Volume 9, Number 2, June 1990, pp. 185-202.

[DoC90] Debra Dooley and Michael F. Cohen, "Automatic Illustration of 3D Geometric Models: Lines," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 77-82.

[DoN90] David P. Dobkin, Stephen C. North, and Nathaniel J. Thurston, "A Viewer for Mathematical Structures and Surfaces in 3D," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 141-142.

[DoS87] Jack J. Dongarra, Danny C. Sorenson, "A Portable Environment for Developing Parallel Fortran Programs," Parallel Computing 5, North-Holland, 1987, pp. 175-186.

[DuS90] A. Dupuy, J. Schwartz, Y. Yemini and D. Bacon, "NEST: A Network Simulation and Prototyping Testbed," Communications of the ACM, Volume 33, Number 10, October 1990, pp. 63-74.

[Dur90] Tony Durham, "Data-Gathering - From Observation to Interpretation," The Computer Bulletin, Volume 2, Part 4,, May 1990, pp. 12-15.

[EbH90] Rainer Ebner, Günter Haring, Franz Penz, and Gerhard Weichselberger, "A Comparative Evaluation of Graphical User-Interfaces," ACM SIGCHI Bulletin, Volume 22, Number 1, July 1990, pp. 12-15.

[EiD90] Marc Eisenstadt, John Domingue, Tim Rajan, and Enrico Motta, "Visual Knowledge Engineering," IEEE Transactions on Software Engineering, Volume 16, Number 10, October 1990, pp. 1164-1177.

[ElM91] James L. Elshoff and Michael Marcotty, "Microtool: An Environment for Programming Microprocessors," IEEE Software, Volume 8, Number 1, January 1991, pp. 42-48.

[EnL91] Marc Engels, Rudy Lauwereins, and Jean Peperstraete, "A Rapid Prototyping Environment for Digital Signal Processing Systems based on Multiprocessors", To appear in IEEE Design & Test, May 1991.

[FeB90] Steven Feiner and Clifford Beshers, "Visualizing n-Dimensional Virtual Worlds with n-Vision," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 37-38.

[Fis89] Gerhard Fischer, "Human-Computer Interaction Software: Lessons Learned, Challenges Ahead," IEEE Software, Volume 6, Number 1, January 1989, pp. 44-52.

[Fit91] Karen Fitzgerald, "Medical Electronics," IEEE Spectrum, Volume 28, Number 1, January 1991, pp. 76-78.

[FoK89] James Foley, Won Chul Kim, Srdjan Kovacevic, and Kevin Murray, "Defining Interfaces at a High Level of Abstraction," IEEE Software, Volume 6, Number 1, January 1989, pp. 25-32.

[Fru91] Martin Frühauf, "Volume Visualization on Workstations: Image Quality and Efficiency of Different Techniques," Computers & Graphics, Volume 15, Number 1, 1991, pp. 101-107.

[GaS91] William W. Gaver and Randall B. Smith, "Auditory Icons in Large-Scale Collaborative Environments," ACM SIGCHI Bulletin, Volume 23, Number 1, January 1991, p. 96.

[Gai90] Blaine Gaither, "Scientific Visualization of Performance Data: Evaluation of DV-DRAW," Performance Evaluation Review, Volume 18, Number 2, August 1990, pp. 48-53.

[Gan19] H. L. Gantt, "Organizing for Work," Industrial Management, Volume 58, August 1919, pp. 89-93.

[Gan90] Joel W. Gannett, "SHORTFINDER: A Graphical CAD Tool for Locating Net-to-Net Shorts in VLSI Chip Layouts," IEEE Transactions on Computer-Aided Design, Volume 9, Number 6, June 1990, pp. 669-674.

[GuG89] Vincent A. Guarna, Jr., Dennis Gannon, David Jablonowski, Allen D. Malony, and Yogesh Gaur, "Faust: An Integrated Environment for Parallel Programming," IEEE Software, Volume 6, Number 7, July 1989, pp. 20-27.

[HaF90] E. Hart and S. Flavell, "Prototyping Transputer Applications," *Real-Time Systems with Transputers*, ed. H. Zedan, OCCAM User Group, IOS Press, Amsterdam, 1990, pp. 242-247.

[HaL90a] Andrew Harbert, William Lively, and Sallie Sheppard, "A Graphical Specification System for User-Interface Design," IEEE Software, Volume 7, Number 7, July 1990, pp. 12-20.

[HaL90b] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, Mark Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," IEEE Transactions on Software Engineering, Volume 16, Number 4, April 1990, pp. 403-414.

[HaU91] Martin Hardwick, Wayne Uejio, David Spooner, Joe Czechowski, Phil Lohr, and Brion Sarachan, "Implementing a User Interface Management System for Existing Applications Using an Object-Oriented Database Tool," Software - Practice and Engineering, Volume 21, Number 2, February 1991, pp. 167-185.

[HaW90] Dieter Haban, Dieter Wybranietz, "A Hybrid Monitor for behavior and Performance Analysis of Distributed Systems," IEEE Transactions on Software Engineering, Volume 16, Number 2, February 1990, pp. 197-211.

[Hai91] D. Charles Hair, "Legalese: A Legal Argumentation Tool," ACM SIGCHI Bulletin, Volume 23, Number 1, January 1991, pp. 71-74.

[Har89] Rex Hartson, "User-Interface Management Control and Communication," IEEE Software, Volume 6, Number 1, January 1989, pp. 62-70.

[Har90a] Robert M. Haralick, "Investigating the Effects of Color," Communications of the ACM, Volume 33, Number 2, February 1990, pp. 120-125.

[Har90b] James Hargreaves, "3D Seismics," The Computer Bulletin, Volume 2, Part 1, January/February 1990, pp. 14-16.

[Har90c] Warren Harrison, "Tools for Multiple-CPU Environments," IEEE Software, Volume 7, Number 5, May 1990, pp. 45-51.

Survey.

[HeE91] M. T. Heath and J. A. Etheridge, "Visualizing Performance of Parallel Programs," Technical Memo ORNL/TM-11813, Oak Ridge National Laboratory, Engineering Physics and Mathematics Division, Mathematical Sciences Section, May 1991.

[HeL91] Virginia Herrarte and Ewing Lusk, "Studying Parallel Program Behavior with Upshot," Technical Paper, Argonne National Laboratory, Mathematics and Computer Science Division, July 1991.

[HeM90] Allan Heydon, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski, "Miro': Visual Specification of Security," IEEE Transactions on Software Engineering, Volume 16, Number 10, October 1990, pp. 1185-1197.

[HeP89] Gerard Hegron, Patrizia Palamidese, and Daniel Thalmann, "Motion Control in Animation, Simulation and Visualization," Computer Graphics Forum, Volume 8, 1989, pp. 347-352.

[Hea90] Michael T. Heath, "Visual Animation of Parallel Algorithms for Matrix Computations," Mathematical Sciences Section, Oak Ridge National Laboratory.

[HiL90] Rumi Hiraga and Yeong-Chang Lien, "Managing Objects in a User Interface System With ROBA," ACM SIGCHI Bulletin, Volume 22, Number 1, July 1990, pp. 49-51.

[HiS91] Deborah Hix and Robert S. Schulman, "Human-Computer Interface Development Tools: A Methodology for their Evaluation," Communications of the ACM, Volume 34, Number 3, March 1991, pp. 75-87.

[HiT90] Masahito Hirakawa, Minoru Tanaka, and Tadoa Ichikawa, "An Iconic Programming System, HI-VISUAL," IEEE Transactions on Software Engineering, Volume 16, Number 10, October 1990, pp. 1178-1184.

[Hin90] John R. Hines, "Affordable Analog Design," IEEE Spectrum, Volume 27, Number 11, November 1990, pp. 60-93.

[Hix89] Deborah Hix, "User Interfaces: Opening a Window on the Computer," IEEE Software, Volume 6, Number 1, January 1989, pp. 8-11.

[HoS89] Matthew E. Hodges, Russell M Sasnett, and Mark S. Ackerman, "A Construction Set for Multimedia Applications," IEEE Software, Volume 6, Number 1, January 1989, pp. 37-43.

[Hoa90] Ellen D. Hoadley, "Investigating the Effects of Color," Communications of the ACM, Volume 33, Number 2, February 1990, pp. 120-125.

[HuL89] Wolfgang Hubner and Manuel Lancastre, "Towards an Object-Oriented Interaction Model for Graphics User Interfaces," Computer Graphics Forum, Volume 8, 1989, pp. 207-217.

[HuS89] William D. Hurley and John L. Sibert, "Modeling User Interface-Application Interactions," IEEE Software, Volume 6, Number 1, January 1989, pp. 71-77.

[HwW90] Mark I. H. Hwang and Bruce J. P. Wu, "The Effectiveness of Computer Graphics for Decision Support: A Meta-Analytical Integration of Research Findings," Data Base, Volume 21, Number 2/3, Fall 1990, pp. 11-20.

[IcH90] Tadao Ichikawa and Masahito Hirakawa, "Iconic Programming: Where to Go ?" IEEE Software, Volume 7, Number 11, November 1990, pp. 63-68.

[IwK90] A. Iwainsky, D. Kaiser, and M. May, "Computer Graphics and Layout Design in Documentation Processes," Computers & Graphics, Volume 14, Numbers 3/4, 1990, pp. 377-388.

[Iwa90] Hiroo Iwata, "Artificial Reality with Force-Feedback: Development of Desktop Virtual Space with Compact Master Manipulator," *Proceedings of SIGGRAPH ’90*, Dallas, August 6-10, 1990, (Computer Graphics, Volume 24, Numbers 4, pp. 165-170).

[JaG87a] L. H. Jamieson, D. B. Gannon, R. J. Douglass, "Characterizing Parallel Algorithms," *The Characteristics of Parallel Algorithms*, The MIT Press, Cambridge, Mass, 1987, pp. 65-100.

[JaG87b] L. H. Jamieson, D. B. Gannon, R. J. Douglass, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs," *The Characteristics of Parallel Algorithms*, The MIT Press, Cambridge, Mass, 1987, pp. 363-394.

[JaL87] Philippe Jauffret, Claude Laurenco, and Ge’rard Kaufmann, "PSYCHO: A Program for Computer-Aided Synthesis in Organic Chemistry," Technology and Science of Informatics, Volume 6, Number 2, 1987, pp. 129-143.

[JeK90] W. Jetschny, and W. Krug, "Graphic Presentation and Animation in the MSOKS Kernel System for Modeling, Simulation, and Optimization in the Computer-Aided Production," Computers & Graphics, Volume 14, Numbers 3/4, 1990, pp. 419-427.

[JoM90] D. Joensson, J. M*ü*glitz, and M. Vogel, "Computer Graphics for Design Processes in Mechanical Engineering," Computers & Graphics, Volume 14, Numbers 3/4, 1990, pp. 413-417.

[JoS90] A. Russell Jones and Esther R. Steinberg, "Interface Design: An Educational Perspective," Journal of Systems and Software, Volume 11, Number 3, 1990, pp. 199-204.

[JoW89] Nigel W. John and Philip J. Willis, "The Controller Animation System," Computer Graphics Forum, Volume 8, 1989, pp. 133-138.

[Jon90] Beverly J. Jones, "Computer Graphics: Effects of Origins," Digital Image - Digital Cinema, *Leonardo* Journal of the International Society for the Arts, Sciences and Technology, Supplemental Issue 1990, (on SIGGRAPH 1990), pp. 21-30.

[KaF88] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich, "Intelligent Assistance for Software Development and Maintenance," IEEE Software, May 1988, pp. 40-49.

[KaK90] Toshikazu Kato, Takio Kurita, and Hiroyuki Shimogaki, "Multimedia Interaction With Image Database Systems," ACM SIGCHI Bulletin, Volume 22, Number 1, July 1990, pp. 52-54.

[KaK91] Tomihisa Kamada and Satoru Kawai, "A General Framework for Visualizing Abstract Objects and Relations," ACM Transactions on Graphics, Volume 10, Number 1, January 1991, pp. 1-39.

[KaS89] Eliezer Kantrorowitz and Oded Sudarsky, "The Adaptable User Interface," Communications of the ACM, Volume 32, Number 11, November 1989, pp. 1352-1358.

[KaY90] Arie Kaufman, Roni Yagel, and Reuven Bakalash, "Direct Interaction with a 3D Volumetric Environment," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 33-34.

[Kar88] Gerald M. Karam, "An Icon-Based Design Method for Prolog," IEEE Software, Volume 5, Number 7, July 1988, pp. 51-65.

[KoC91] James Arthur Kohl and Thomas L. Casavant, "Use of PARADISE: A Meta-Tool for Visualizing Parallel Systems," *Proceedings Fifth International Parallel Processing Symposium*, Anaheim, California, May 1991, pp. 561-567.

[Kob90] Hans-Ulrich Kobialka, "Configuration Editing, Generation and Test within Working Contexts," ACM Software Engineering Notes, Volume 15, Number 6, December 1990, Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments, pp. 173-182.

[KrB83] D. Kramlich, G. P. Brown, R. T. Carling, and C. F. Herot, "Program Visualization: Graphics Support for Software Development," *Proceedings of the ACM/IEEE 20th Design Automation Conference*, June 1983, IEEE-CS Press, Los Alamitos, California, pp. 143-149.

Just ref.

[KrU89] Murali Krishnamurthi, and Alvin J. Underbrink, Jr., "Knowledge Acquisition in a Machine Fault Diagnosis Shell," SIGART Newsletter, Knowledge Acquisition Special Issue, April 1989, Number 108, pp. 84-91.

[KuB91a] Jayant A. Kulkarni, Joseph S. Byrd, and Robert O. Pettus, "X-Window Based Graphical of Mobile Robot Simulator for Path Planning and Sensor Fusion Experiments," Simulation Digest, Volume 21, Number 3, April 1991, pp. 185-192.

[KuB91b] Gordon Kurtenbach and Bill Buxton, "GEDIT: A Test Bed for Editing by Contiguous Gestures," ACM SIGCHI Bulletin, Volume 23, Number 2, April 1991, pp. 22-26.

[KuO90] Ernest S. Kuh and Tatsuo Ohtsuki, "Recent Advances in VLSI Layout," Proceedings of the IEEE, Volume 78, Number 2, February 1990, pp. 237-263.

[LaE90] Rudy Lauwereins, Marc Engels, Jean Peperstraete, Eric Steegmans, and Johan Van Ginderdeuren, "GRAPE: A CASE Tool for Digital Signal Parallel Processing", IEEE ASSP Magazine, April 1990, pp. 32-43.

[LaM90] G. Landgraf, K.-H. Modler, and M. Ziegenhorn, "System to Represent Doubly-Curved Surfaces," Computers & Graphics, Volume 14, Number 3/4, 1990, pp. 435-443.

[LeB90] Ted Lehr, David Black, Zary Segall, Dalibor Vrsalovic, "Visualizing Context-Switches Using PIE and the Mach Kernel Monitor," *Proceedings of the 1990 International Conference on Parallel Processing*, Volume II, St. Charles, Illinois, August 1990, pp. 298-299.

[LeL90] H. Lempert, M. Lutz, L. A. Messina, W. S. Ting, and C. Sänger, "A Prototype for Configuring Hardware and Software," Computers & Graphics, Volume 14, Number 2, 1990, pp. 303-310.

[LeM87] Thomas J. Leblanc, John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," IEEE Transactions on Computers, Volume C-36, Number 4, April 1987, pp. 471-482.

[LeM90] Thomas J. Leblanc, John M. Mellor-Crummey, Robert J. Fowler, "Analyzing Parallel Program Executions Using Multiple Views," *The Journal of Parallel and Distributed Computing*, Volume 9, Number 2, June 1990, pp. 203-217.

[Lee90] Ed Lee, Hewlett-Packard Interface Technology Operation, "User-Interface Development Tools," IEEE Software, May 1990, pp. 31-36.

[LiB89] Thomas D. Linton, Jr., and Peter A. Blakey, "A Fast, General Three-Dimensional Device Simulator and Its Application in a Submicron EPROM Design Study," IEEE Transactions on Computer-Aided Design, Volume 8, Number 5, May 1989, pp. 508-515.

[LoD85] Ralph L. London and Robert A. Duisberg, "Animating Programs Using Smalltalk," IEEE Computer, Volume 18, Number 8, August 1985, pp. 61-71.

[LuR90] M. Ludwig, and Ch. Richter, "The 3D Geometry Modeling System (GEMO)," Computers & Graphics, Volume 14, Numbers 3/4, 1990, pp. 389-394.

[MAT91] MATHEMATICA, Wolfram Research, Inc., Champaign, IL.

Just Reference.

[MaC90] Dirk E. Mahling and W. Bruce Croft, "An Interface for the Acquisition and Display of Office Procedures," ACM SIGOIS Bulletin, Volume 11, Number 2/3, April 1990, Conference on Office Information Systems, Cambridge, Massachusetts, pp. 123-130.

[MaD89] Nadia Magnenat-Thalmann and Daniel Thalmann, "Synthetic Actors: the Simulation of Human Motion," The Computer Bulletin, Volume 1, Part 1, January/February 1989, pp. 12-14.

[MaK90] Robert Marshall, Jill Kempf, and Scott Dyer, "Visualization Methods and Simulation Steering for a 3D Turbulence Model of Lake Erie," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 89-97.

[MaM89] Stefano Mannucci, Bruno Mojana, Maria Cristina Navazio, Valerio Romano, Maria Caria Terzi, and Piero Torrigliani, "Graspin: A Structured Development Environment for Analysis and Design," IEEE Software, Volume 6, Number 11, November 1989, pp. 35-43.

[MaR89] A. D. Malony, D. A. Reed, "An Integrated Performance Data Collection, Analysis, and Visualization System," *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers and Applications*, 1989.

[Mar87] J. L. Martin, "Performance Evaluation: Applications and Architectures," *Proceedings of the 2nd International Conference on Supercomputing*, Volume III, 1987, pp.

369-373.

[Mar90] Shaun Marsh, "Human-Computer Interaction: An Operational Definition," ACM SIGCHI Bulletin, Volume 22, Number 1, July 1990, pp. 16-22.

[Mat90] Bruce C. Mather, "Needed DSP Software Emerges," IEEE Spectrum, Volume 27, Number 11, November 1990, pp. 52-59.

[McP90] Micheal McKenna, Steve Pieper, and David Zeltzer, "Control of a Virtual Actor: The Roach," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 165-174.

[McW91] Harold J. McWhinnie, "Some Aesthetic Questions on Computer-Based Art and Design," Computers & Graphics, Volume 15, Number 1, 1991, pp. 139-142.

[MeM85] B. Melamed and R. J. T. Morris, "Visual Simulation: The Performance Analysis Workstation," IEEE Computer, Volume 18, Number 8, August 1985, pp. 87-94.

[Med82] R. Medina-Mora, "Syntax-Directed Editing: Towards Integrated Programming Environments," PhD Dissertation, Carnegie-Mellon University, 1982.

[Mey91] Scott Meyers, "Difficulties in Integrating Multiview Development Systems," IEEE Software, Volume 8, Number 1, January 1991, pp. 49-57.

[MiC90] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," IEEE Transactions on Parallel and Distributed Systems, Volume 1, Number 2, April 1990, pp. 206-217.

[MiO90] Margaret Minsky, Ming Ouh-young, Oliver Steele, Frederick P. Brooks, Jr., and Max Behensky, "Feeling and Seeing: Issues in Force Display," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 235-243.

[MiY87] Barton P. Miller, Cui-Qing Yang, "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs," 7th International Conference on Distributed Computing Systems, September 1987, pp. 482-489.

[MiY89] Barton P. Miller, Cui-Qing Yang, "Performance Measurement for Parallel and Distributed Programs: A Structured and Automatic Approach," IEEE Transactions on Software Engineering, Volume 15, Number 12, December 1989, pp. 1615-1629.

[Mic90] John G. Michalakes, "Analysis of Workload and Load Balancing Issues in NCAR Community Climate Model," Mathematics and Computer Science Division, Argonne National Laboratory, ANL/MCS-TM-144, 1990.

[Mil88] Barton P. Miller, "DPM: A Measurement System for Distributed Programs," IEEE Transactions on Computers, Volume 37, Number 2, February 1988, pp. 243-248.

[MoH85] Mark Moriconi and Dwight F. Hare, "Visualizing Program Designs Through PegaSys," IEEE Computer, Volume 18, Number 8, August 1985, pp. 72-85.

[Mor74] M. F. Morris, "Kiviat Graphs - Conventions and Figures of Merit," Performance Evaluation Review, Volume 3, Number 3, October 1974, pp. 2 - 8.

[MuS91] Michael J. Muller, John G. Smith, J. Zachary Shoher, and Harry Goldberg, "Privacy, Anonymity and Interpersonal Competition Issues Identified During Participatory Design of Project Management Groupware," ACM SIGCHI Bulletin, Volume 23, Number 1, January 1991, pp. 82-87.

[Mus89] Mark A. Musen, "Knowledge Acquisition at the Metalevel: Creation of Custom-Tailored Knowledge-Acquisition Tools," SIGART Newsletter, Knowledge Acquisition Special Issue, April 1989, Number 108, pp. 45-55.

[MyG90] Brad A. Myers, Dario A. Giuse, Roger B. Dannenburg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal, "Garnet - Comprehensive Support for Graphical, Highly Interactive User Interfaces," IEEE Computer, Volume 23, Number 11, November 1990, pp. 71-85.

[Mye83] Brad A. Myers, "Incense: A System for Displaying Data Structures," Computer Graphics, Volume 17, Number 3, July 1983, pp. 115-125.

[Mye88] B. A. Myers, "The State of the Art in Visual Programming and Program Visualization," Report Number CMU-CS-88-114, Computer Science Department, Carnegie Mellon, 1988.

[Mye89] Brad A. Myers, "User-Interface Tools: Introduction and Survey," IEEE Software, Volume 6, Number 1, January 1989, pp. 15-23.

[NeM90] F. Neelamkavil and O. Mullarney, "Separating Graphics from Application in the Design of User Interfaces," The Computer Journal, Volume 33, Number 5, October 1990, pp. 437-443.

[Nee90] Henry Neeman, "A Decomposition Algorithm For Visualizing Irregular Grids," Computer Graphics, Volume 24, Number 5, November 1990, pp. 49-56.

[NiE88] Kathleen M. Nichols and John T. Edmark, "Modeling Multicomputer Systems with PARET," IEEE Computer, Volume 21, Number 5, May 1988, pp. 39-49.

[NiM89] Celso Niskier, Tom Maibaum, and Daniel Schwabe, "A Look Through PRISMA: Towards Pluralistic Knowledge-Based Environments for Software Specification Acquisition," ACM SIGSOFT Software Engineering Notes, Volume 14, Number 3, May 1989, pp. 128-136.

[Nic90] Kathleen M. Nichols, Apple Computer, "Performance Tools," IEEE Software, May 1990, pp. 21-30.

[Nic91] Kathleen M. Nichols, "A Customizable User Interface for Simulation," Simulation Digest, Volume 21, Number 2, Winter 1990/91, pp. 37-50.

[Oma90a] Paul W. Oman, University of Idaho, "CASE Analysis and Design Tools," IEEE Software, Volme 7, Number 5, May 1990, pp. 37-43.

[Oma90b] Paul Oman, "Maintenance Tools," IEEE Software, Volume 7, Number 5, May 1990, pp. 59-65.

Survey of various computing uses.

[PaC90] Y. E. Papelis and T. L. Casavant, "Mathematical Transformation of Petri-Nets for Development of Parallel/Distributed Programs," *Proceedings 14th Annual Int'l Computer Software and Applications Conference (COMPSAC)*, Chicago, Illinois, November 1990, pp. 92-98.

[PaC91] Y. E. Papelis and T. L. Casavant, "XPAT: An Interactive Graphical Tool for Synthesis of Concurrent Software using Petri-Nets," To appear *Proceedings 15th Annual Int'l Computer Software and Applications Conference (COMPSAC)*, Tokyo, Japan, September 1991.

[PaN89] L. F. Pau, and S. Skafte Nielson, "Conceptual Graphs as a Visual Language for Knowledge Acquisition in Architectural Expert Systems," SIGART Newsletter, Knowledge Acquisition Special Issue, April 1989, Number 108, pp. 151.

[PaP91] Patrick Paultre and Jean Proulx, "CAL/CGI - An Application of Graphics for Matrix Structural Analysis Education," Computers & Graphics, Volume 15, Number 1, 1991, pp. 131-135.

[PeE90] A. Pentland, I. Essa, M. Friedmann, B. Horowitz, S. Sclaroff, "The ThingWorld Modeling System: Virtual Sculpting By Modal Forces," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 143-144.

[PeG91] Daniel Pease, Arif Ghafoor, Ishfaq Ahmad, David L. Andrews, Kamal Foudil-Bey, Thomas E. Karpinski, Mohammad A. Mikki, and Mohamed Zerrouki, "PAWS: A Performance Evaluation Tool for Parallel Computing Systems," IEEE Computer, Volume 24, Number 1, January 1991, pp. 18-29.

[PeM88] Mikel D. Petty, J. Michael Moshell, and Charles E. Hughes, "Tactical Simulation in an Object-Oriented Animated Graphics Environment," Simuletter, Volume 19, Number 2, June 1988, pp. 31-46.

[PeR89] J.-R. Perez-Cazares and R. Rannou, "Developing an Animated Local Network Simulator," Computer Graphics Forum, Volume 8, 1989, pp. 139-142.

[Per90] Tekla S. Perry, "Biomechanically Engineered Athletes," IEEE Spectrum, Volume 27, Number 4, April 1990, pp. 43-44.

[PhV90] Kiem-Phong Vo, "IFS: A Tool to Build Application Systems," IEEE Software, Volume 7, Number 7, July 1990, pp. 29-36.

Note: Not visually oriented - creates visual interfaces.

[PiM90] Clifford A. Pickover, and Kevin McCarty, "Visualizing Cantor Cheese Construction," Computers & Graphics, Volume 14, Number 2, 1990, pp. 337-341.

[PiT90] Xavier Pintado and Dennis Tsichritzis, "SaTELLite - A Visualization and Navigation Tool for Hypermedia," ACM SIGOIS Bulletin, Volume 11, Number 2/3, April 1990, Conference on Office Information Systems, Cambridge, Massachusetts, pp. 271-280.

[PlP89] Michael Plummer and David Penna, "Mass Market Applications for Real Time 3D Graphics," Computer Graphics Forum, Volume 8, 1989, pp. 143-150.

[PoL83] M. L. Powell and M. A. Linton, "Visual Abstraction in an Interactive Programming Environment," *Proceedings Sigplan Symposium on Programming Language Issues in Software Systems*, Printed as Sigplan Notices, Volume 18, Number 6, June 1983, pp. 14-21.

[PuJ91] James M. Purtilo and Pankaj Jalote, "An Environment for Prototyping Distributed Applications," Computer Languages, Volume 16, Number 3/4, 1991, pp. 197-207.

[Rac90] John Race, "Using Computer Graphics to Find Interesting Properties in Data," The Computer Bulletin, Volume 2, Part 7, September 1990, pp. 15-16.

[Rae84] Georg Raeder, "Programming in Pictures," PhD Dissertation, University of Southern California, Los Angelos, California, November 1984, Technical Report TR-84-318, USC.

[Rae85] Georg Raeder, "A Survey of Current Graphical Programming Techniques," IEEE Computer, Volume 18, Number 8, August 1985, pp. 11-25.

[Rei84] Steven P. Reiss, "Graphical Program Development with PECAN Program Development Systems," *Proceedings ACM Sigsoft-Sigplan Software Engineering Symposium on Practical Software Development Environments*, April 1984, Printed as Sigplan Notices, Volume 19, Number 5, May 1984, pp. 30-41.

[Rei85] Steven P. Reiss, "PECAN: Program Development Systems that Support Multiple Views," IEEE Transaction on Software Engineering, Volume SE-11, Number 3, March 1985, pp. 276-285.

[Rei90] Steven P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," IEEE Software, Volume 7, Number 7, July 1990, pp. 57-66.

[RhT90] Penny Rheingans and Brice Tebbs, "A Tool for Dynamic Explorations of Color Mappings," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 145-146.

[RoC91] Diane T. Rover, Michael B. Carter, and John L. Gustafson, "Performance Visualization of SLALOM," To appear DMCC6 proceedings, 1991.

[RoP89] Linda L. Rodi, Jonathan A. Pierce, and Robert E. Dalton, "Putting the Expert in Charge: Graphical Knowledge Acquisition for Fault Diagnosis and Repair," SIGART Newsletter, Knowledge Acquisition Special Issue, April 1989, Number 108, pp. 56-62.

[RoW90] D. T. Rover, C. T. Wright, "Pictures of Performance: Highlighting Program Activity in Time and Space," *Proceedings of the 5th Distributed Memory Computer Conference*, IEEE Computer Society Press, 1990.

[Rov89] Diane T. Rover. "Visualization of Program Performance on Concurrent Computers," PhD Thesis, Iowa State University, Ames, Iowa, 1989.

[Rov91] Diane T. Rover, "A Performance Visualization Paradigm for Data Parallel Computing," submitted to HICSS-25 Workshop on Parallel Programming Tools, 1991.

[RuW90] Robert V. Rubin, James Walker II, and Eric J. Golin, "Early Experience with the Visual Programmer's WorkBench," IEEE Transactions on Software Engineering, Volume 16, Number 10, October 1990, pp. 1107-1121.

[ScA89] R. W. Schwanke, R. Z. Altucher, and M. A. Platoff, "Discovering, Visualizing, and Controlling Software Structure," ACM SIGSOFT Software Engineering Notes, Volume 14, Number 3, May 1989, pp. 147-150.

[Sch88] Edward N. Schwartz, "A Pictorial Aid for Programming Expert Systems," SIGART Newsletter, October 1988, Number 106, pp. 22-24.

[SeR85] Z. Segall, L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, Volume 2, No. 6, November 1985, pp. 22-37.

[SeR90] Kudang B. Seminar and Robert N. Robson, "An Iconic Description Language: Programming Support for Data Structure Visualization," ACM SIGCHI Bulletin, Volume 22, Number 1, July 1990, pp. 70-72.

[ShD88] Mark Sherman and Robert L. Drysdale III, "Teaching Software Engineering in a Workstation Environment," IEEE Software, Volume 5, Number 5, May 1988, pp. 68-75.

[ShI91] Takao Shimomura and Sadahiro Isoda, "Linked-List Visualization for Debugging," IEEE Software, Volume 8, Number 5, May 1991, pp. 44-51.

[ShR90] Vincent Y. Shen, Charles Richter, Michael L. Graf, Jeffrey A. Brumfield, "VERDI: A Visual Environment for Designing Distributed Systems," *The Journal of Parallel and Distributed Computing*, Volume 9, Number 2, June 1990, pp. 128-137.

[Sha89] Mildred L. G. Shaw, "A Grid-Based Tool for Knowledge Acquisition: Validation with Multiple Experts," SIGART Newsletter, Knowledge Acquisition Special Issue, April 1989, Number 108, pp. 168-169.

[Sny84] Lawrence Snyder, "Parallel Programming and the Poker Programming Environment," IEEE Computer, Volume 17, Number 7, July 1984, pp. 27-37.

[SpK90] Don Speray and Steve Kennon, "Volume Probes: Interactive Data Exploration on Arbitrary Grids," Computer Graphics, Volume 24, Number 5, November 1990, pp. 5-12.

[Spe90] Spectrum Staff, "Displaying Processed Data," IEEE Spectrum, Volume 27, Number 11, November 1990, pp. 86-89.

[SrM89] K. Sridharan, M. McShea, J. C. Browne, and P. Newton, "An Environment for Parallel Structuring of Fortran Programs," *Proceedings International Conference on Parallel Processing*, St. Charles, Illinois, August 1989, Volume II, pp. 98-106.

[StD90] Richard St-Denis, "LGV: A Domain Knowledge Validation Environment," Computers & Graphics, Volume 14, Number 2, 1990, pp. 311-320.

[Sta90] John T. Stasko, "Tango: A Framework and System for Algorithm Animation," IEEE Computer, Volume 23, Number 9, September 1990, pp. 27-39.

[ToC88] Sherryl Tomboulian, Thomas W. Crockett, and David Middleton, "A Visual Programming Environment for the Navier-Stokes Computer," *Proceedings International Conference on Parallel Processing*, St. Charles, Illinois, August 1988, Volume II, pp. 67-71.

[TsR88] Jeffrey J.-P. Tsai and Joel C. Ridge, "Intelligent Support for Specifications Transformation," IEEE Software, Volume 5, Number 9, November 1988, pp. 28-35.

[Tyu91] Enn Tyugu, "Three New-Generation Software Environments," Communications of the ACM, Volume 34, Number 6, June 1991, pp. 46-59.

[Ven90] George D. Vendelin, "Evaluating Nonlinear Models for Microwave GaAsFETs," IEEE Spectrum, Volume 27, Number 9, September 1990, pp. 48-50.

[VoS88] Edward L. Vockell and Eileen Schwartz "Computer Software That Helps Manage and Deliver Instruction," Interface - The Computer Education Quarterly, Volume 10, Number 4, 1988, pp. 44-50.

[WaK90] Chu P. Wang, Lawrence Koved, and Semyon Dukach, "Design for Interactive Performance in a Virtual Laboratory," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 39-40.

[WeF90] Elizabeth M. Wenzel and Scott H. Foster, "Realtime Digital Synthesis of Virtual Acoustic Environments," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 139-140.

[Wei84] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communication of the ACM*, Volume 27, Number 10, 1984.

[Wil90a] Lance Williams, "3D Paint," Computer Graphics, Volume 24, Number 2, March 1990, (Special Issue on 1990 Symposium on Interactive 3D Graphics), pp. 225-233.

[Wil90b] Andrew Wilkes, "Mapping Technology," The Computer Bulletin, Volume 2, Part 10, December 1990, pp. 17.

[Wri90] Richard Wright, "Computer Graphics as Allegorical Knowledge: Electronic Imagery in the Sciences," Digital Image - Digital Cinema, *Leonardo* Journal of the International Society for the Arts, Sciences and Technology, Supplemental Issue 1990, (on SIGGRAPH 1990), pp. 65-73.

[ZyM90] Michael J. Zyda, Robert B. McGhee, Corrine M. McConkle, Andrew H. Nelson, and Ron S. Ross, "A Real-Time, Three-Dimensional Moving Platform Visualization Tool," Computers & Graphics, Volume 14, Number 2, 1990, pp. 321-333.