SANDIA REPORT

SAND98–0359 • UC–900
Unlimited Release
Printed March 1998

Software Attribute Visualization for High Integrity Software

Guylaine M. Pollock

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A06
Microfiche copy: A01

SAND98-0359
Unlimited Release
Printed March1998

Distribution
Category UC-900

Software Attribute Visualization for High Integrity Software

Guylaine M. Pollock
Computer Sciences Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1109

Abstract

This report documents a prototype tool developed to investigate the use of visualization and virtual reality technologies for improving software surety confidence. The tool is utilized within the execution phase of the software life cycle. It provides a capability to monitor an executing program against prespecified requirements constraints provided in a program written in the requirements specification language SAGE. The resulting Software Attribute Visual Analysis Tool (SAVAnT) also provides a technique to assess the completeness of a software specification.

Intentionally Left Blank

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

List of Figures

List of Figures(Cont)

Intentionally Left Blank

Executive Summary

This report documents a prototype tool developed to investigate the use of visualization and virtual reality technologies for improving software surety confidence. The tool is utilized within the execution phase of the software life cycle. It provides a capability to monitor an executing program against prespecified requirements constraints provided in a program written in the requirements specification language SAGE. The resulting Software Attribute Visual Analysis Tool (SAVAnT) also provides a technique to assess the completeness of a software specification.

The prototype tool is described along with the requirements constraint language after a brief literature review is presented. Examples of how the tool can be used are also presented, before specific information is given on how to access the tool and provide extensions for future development. In conclusion, the most significant advantage of this tool is to provide a first step in evaluating specification completeness, and provides a more productive method for program comprehension and debugging. The expected payoff is increased software surety confidence, increased program comprehension, reduced development and debugging time.

Acronyms and Abbreviations

Eigen/VR	A software platform developed at Sandia National Laboratories to utilize virtual reality technologies. Initiated as MUSE.
HIS	High Integrity Software; Software that is ultra-reliable (a failure probability on the order of less than 10 to the 6 power).
MUSE	Multi-User Synthetic Environment: a software platform developed at Sandia National Laboratories and transferred to the public sector as a product
RCL	Requirements Constraint Language: a specification language used to specify software requirement constraints for use with the SAVAnT system
SAGE	Software Attributes Generic Evaluation; a requirements constraint language for specifying software attribute constraints
SAVAnT	Software Attribute Visual Analysis Tool

Definitions:

High Consequence Applications — Applications where process failures are likely to result in human injury/death, damage to the environment and/or damage to valuable resources or equipment

Surety — Confidence that a system will satisfy its reliability, safety, and security expectations in its intended environment without initiating undesirable actions

Software Surety — Those processes and technology methods/techniques that provide assurance that a system's software component will not cause catastrophic failures that keep a system from achieving its required level of surety

1. Introduction

The development of software for use in high-consequence systems mandates rigorous (formal) processes, methods, and techniques to improve the safety characteristics of those systems. To address this need, research efforts must progress in several areas over the next few decades to allow us to reach, with greater certainty, the higher levels of reliability required by software used in high-consequence systems [7, 15]. This paper describes a strategic surety program developed for high-consequence software under a new initiative at Sandia National Laboratories (SNL)—to identify how we will develop ultra-reliable software in the 2010 time frame.

HIS Initiative

This initiative, the High Integrity Software Program (HIS), is tasked with guiding strategic investments in the development of new capabilities and technologies in the domain of high consequence software at SNL. The program sponsors research within the strategic surety backbone of the defense sector to establish predictive confidence that a system is safe, secure, and under control through the exploration, extension and application of the science of software systems [16]. The program emphasizes high-risk, high payoff research through a correctness research track focussed on a "correctness by design," and more immediate lower-risk, medium payoff applications research through a systems immunology™ track. This track, visualization of abstract objects, produces methods and techniques to render today's systems safer, more secure and more reliable. (Other tracks have been defined but are not currently staffed.)

Once a software system has been developed, a problem still remains of assessing software surety status--rigorous processes and methods applied to early phases of the software life cycle alone cannot assure software integrity, safety, security, and reliability in the final end product. The implementation itself must be verified, with particular focus on surety aspects for high-consequence systems. In that regard, several key issues include whether or not the executing software properly incorporates specified constraints, and whether or not all necessary constraints and their interactions have been considered, understood, and correctly implemented to avoid loss of life or other undesirable effects. How do we verify the surety attributes of a system implementation?

Traditional Research Approaches

Traditionally, there have been three areas of research for verification of system implementations: logical verification, mathematical verification, and statistical verification. However, Berztiss [3] has advocated that every possible technique and method should be utilized to address safety concerns, as current methods to address this problem are inadequate. While testing the actual system code does provide substantial information regarding the correctness of the system, generally this is an incomplete method for assessing surety aspects as economic and scheduling restraints prohibit the level of testing required to achieve the necessary confidence in the surety of real-world systems. Further, tested programs may correctly execute their specifications, but with cur-

rent textual and limited graphical documentation, it is difficult to ascertain whether a code does what is needed.

Mathematical models can be considered for this task. However although rigorous, they can only prove that the implementation meets the specific requirements. They do not allow support for identifying any cases that have not been considered within the requirements and specifications--a drawback of mathematical techniques, they only work if the right cases are proven. Reliability models are also useful, but again, they can only provide statistical confidence at levels that are clearly beneath those required for these high-consequence systems, and they, generally, are making predictions about future failures of the systems without addressing the types of errors or their significance. Finally, none of these existing methods of research address the difficulty of assessing whether all necessary constraints have been specified. This is an area visualization can address.

Therefore, it is time to consider a fourth category, the use of visualization, in addressing the issue of verification. Accordingly, several such efforts are underway in various laboratories and universities [2, 18, 13], including an investigation of software attributes visualization within the High Integrity Software program at Sandia National Laboratories.

Visualization Techniques

Visualization techniques have been used quite successfully within the scientific community for some time; and not surprisingly, many researchers feel the utility of visualization as a means of illustrating the properties of multiple objects, or as a means of demonstrating properties of supersets of discrete items, may be considered a given [4]. Fortunately, this benefit of improved comprehension through visualization can be achieved in other application areas as long as the appropriate visual model is selected. Correspondingly, although system verification is a new context, visualization provides the capability of increased system comprehension, thereby facilitating discoveries that are not otherwise possible. This is a major benefit of using visualization in a formal method to investigate surety aspects of a system implementation. However, little work currently has been undertaken to apply multi-dimensional visualization techniques to software analysis [5], while a number of projects have focussed on algorithm animation, at least in two dimensional formats [23]. (It is only fairly recently that hardware support has been sufficient to allow work on information visualization for analysis of software.)

Projects are just beginning to investigate the use of this methodology for enhancing understanding of system software. Initial successes have resulted in recommendations of investigating the use of virtual reality technology to map multiple-layer software systems onto expansive 3-dimensional terrains and providing more direct means for traversal as a more effective facility for software visualization [13]. We are investigating such a use of visualization and virtual reality techniques, with our efforts going further in utilizing these technologies in assessing surety factors for high-consequence software through the verification of system software [16], as current visualization models do not evaluate or portray surety issues. A multi-dimensional abstract model is used to reduce system complexities associated with the conceptual mapping of a problem domain into a software solution space.

The goal of this track is to improve cognition of software systems behaviour and improve software surety confidence by providing an environment that allows visualization of abstract objects and animation of program behavior incorporating requirement constraints. The project focuses on a multi-dimensional visualization of software abstractions that incorporates a technique for assessing the correct implementation of select requirement constraints during the execution phase of the life-cycle process.

The prototype software attribute visualization tool is developed on Eigen/VR, a multi-dimensional user-oriented synthetic environment developed at Sandia National Laboratories. The tool incorporates the use of requirement constraints, expressed in a requirements constraint language, in the visualization of an executing program. As the program executes, selected requirement constraints are monitored and if violated, the abstract visual model indicates those errors have occurred.

Intentionally Left Blank

2. Background

HIS Program Goals

The High Integrity Software Program (HIS) at Sandia National Laboratories was established to provide a crucial role in guiding internal research efforts to improve technologies that enhance surety aspects of high-consequence systems. This program strives to develop better technologies within the software industry enabling us to increase our confidence in the correctness of high-consequence systems, many of which may become life-threatening if flawed.

State of the Industry

Examining this industry in general, we see software becoming more complex and being relied upon more often for an ever-widening variety of applications. In fact, our dependence on software is exploding quietly—"The amount of code in most consumer products is doubling every two years... televisions may contain up to 500 kilobytes of software; an electric shaver, two kilobytes; while the power trains in new General Motors cars run 30,000 lines of computer code." [11] — and yet software is not reliable in most systems. As a result, software irregularities, in some instances, have taken or degraded people's lives in various system accidents.

Notwithstanding, new types of applications continue to appear on the technological horizon, generating continued cause for concern regarding current abilities to evaluate software surety. For example, Andy White, Director of Los Alamos National Laboratories Advanced Computing Laboratory, has stated that an important goal for new software applications is to solve large problems (such as helping the Forest Service fight fires, helping doctors determine which flu vaccines to use, and making sure that U.S. nuclear bombs do not go off accidentally) that, in short, require us to trust computers to predict the future [1].

While some have encouraged expansion of these types of applications, many others have cited this proliferation as a potential powder-keg for our society: "These days we adopt innovations in large numbers, and put them to extensive use, faster than we can ever hope to know their consequences ... which tragically removes our ability to control the course of events" [14].

Information Warfare

Even more alarming, this increase in numbers and types of software applications has increased our vulnerability as a nation to information warfare. (This is a problem for other nations as well.) In fact, last year the Joint Security Commission stated that "The U.S. vulnerability to infowar may be the major security challenge of this decade and possibly the next century" [8]. Not surprisingly, Pentagon officials have reported an attempt at such warfare was actually suggested to U.S. adversaries during the Gulf war when a group of Dutch hackers offered to disrupt the U.S. military's deployment to the Middle East for $1 Million. If current trends continue, this

type of vulnerability will only increase unless we work to ameliorate our skills in assessing software surety.

Industrial Concerns

Clearly software integrity and surety (safety, security, reliability) issues are a major concern for U.S. industries; as such, they are also a concern for Sandia National Laboratories. Current surety technologies just are not good enough for industries' increasing needs.

Sponsors

Consequently, the HIS program initiative was formulated to address high integrity and surety software issues. Sponsors of the program include the Strategic Surety Backbone of the Defense Programs Sector and the Vice President of Defense Programs. The HIS objective is to establish predictive confidence that a system is safe, secure, and under control.

3. Project Goals

The development of software for use in high-consequence systems mandates rigorous (formal) processes, methods, and techniques to improve the safety characteristics of those systems. Once developed however, the problem still remains of assessing software surety status--rigorous processes and methods alone cannot assure software integrity, safety, security, and reliability. The key issues are whether or not the executing software properly incorporates specified constraints, and whether or not all necessary constraints and their interactions have been considered, understood, and correctly implemented.

Current methods to address this problem are inadequate. While testing the actual code does provide substantial information regarding the correctness of the code, generally this is an incomplete method as economic and scheduling restraints prohibit the level of testing required to achieve the necessary confidence in the surety of real-world systems. Reliability models are useful, but again, they can only provide statistical confidence at levels that are clearly beneath those required for these high-consequence systems. Further, existing methods do not address the difficulty of assessing whether all necessary constraints have been specified.

Visual Models

Visualization techniques have been used quite successfully within the scientific community for some time. It is only recently that hardware support has been sufficient to allow current work on information visualization for analysis of software. Traditional work in this area, however, has focused on two-dimensional flow-chart like structures. This project investigates the use of visualization in this area.

Surety Assessment

This project examines a technique for assessing the correct implementation of select requirement constraints. Further, the project assesses software during the execution phase of the life-cycle process.

Program Comprehension

The primary goal of this project is to improve cognition of software systems behavior and improve software surety confidence by providing an environment that allows visualization of abstract objects and animation of program behavior incorporating requirement constraints. To achieve this goal, a prototype tool, SAVAnT (Software Attribute Visual Analysis Tool), was developed to aid in the visualization of an executing program. The tool is designed to allow the ability to monitor the execution and compare it to prespecified requirements constraints expressed in

what we have termed a requirements constraint language.

Ease of Use

In addition, a goal is to provide a tool that is easy to use. Therefore a preprocessor is provided to generate the required version of the executable program.

Portability

Finally, portability is important. So standard programming languages were utilized.

4. Brief Literature Review

Briefly reviewing related literature, it is clear that work in this area has yet to capitalize on the use of multi-dimensional virtual reality and visualization techniques as applied to the software development process. Figure 1 documents the state of the art for uses of Visualization[1]. It is clear that the use of visualization for the application of software development lags behind the use of visualization for scientific applications. Appendix C contains a more thorough report reviewing the use of visualization.

In reviewing the state of the art in requirements verification approaches, Yau's work is typical. [22] This work checks the completeness between the natural language requirements statements and the object-oriented requirements specification for a given application. However it does not address the completeness of the natural language requirement statements, which the technique described herein can address.

In reviewing the use of visual techniques for software development, a number of systems are described briefly. More detailed information can be found in [17].

Balsa

The Balsa system animated algorithms in Pascal programs for educational purposes. The models were two dimensional in black and white.

Zeus

An upgrade of Balsa, Zeus supports multiple synchronized views of algorithms. It has not been used outside the laboratory and no empirical evaluations have been performed on the tool. Figures 2-9 illustrate the Zeus prototype tool and how it can be used. The first four figures depict a sorting algorithm, and the other four depict additional usage.

Figure 2 shows the beginning of a sorting algorithm. The colored bars represent different data values. The relative values are depicted through color and length with the red bars being the highest values and the blue bars the smallest. The tree structure used to initiate the sort is depicted as well. Figure 3 is a view of the data looking down the Z axis. Figure 4 shows the view down the X axis. Figure 5 shows the sort partially completed. Notice the ordering of the bars.

Figure 6 illustrates a comparison of an implemented algorithm against a correct implementation of the same structure. This comparison clearly identifies a problem with the new implementation as the depth of the generated tree should be balanced and it is not. Figure 7 illustrates a use of

1. Huff, C.C., Klein, M. and Stevens, S., "The State of the Art in Scientific Visualization," appendix C, p. 98.

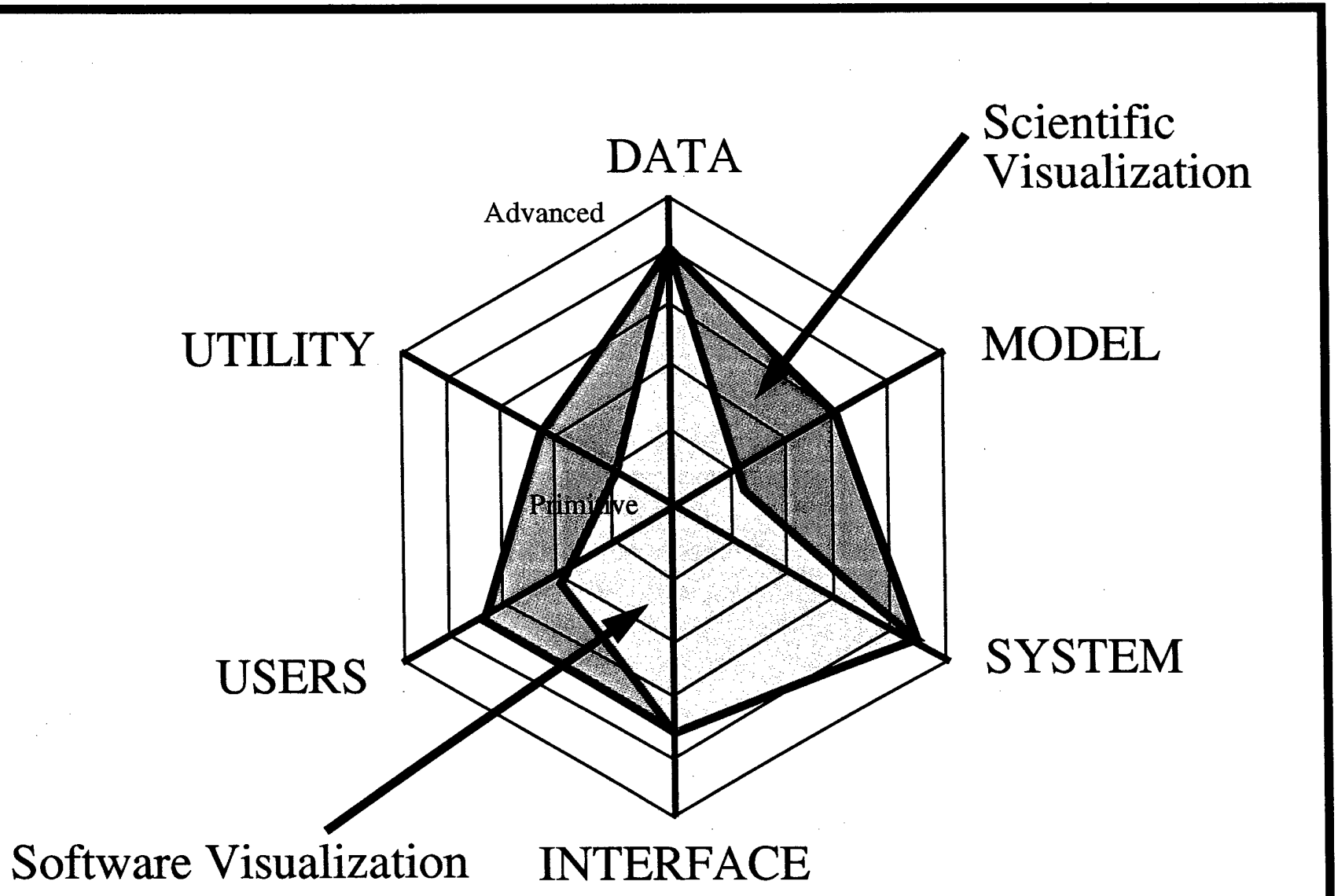


Figure 1: Visualization State of the Art

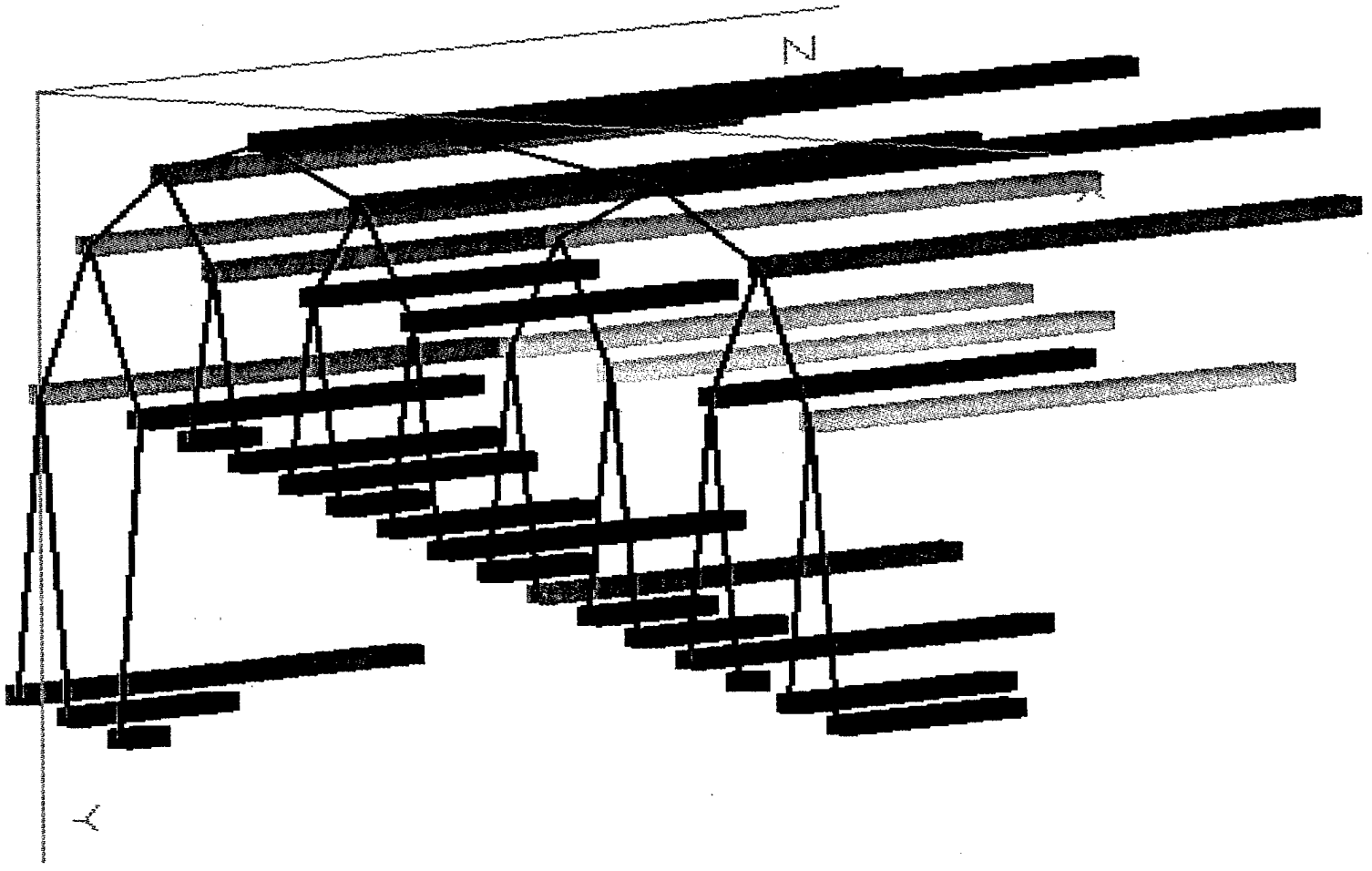


Figure 2: Zeus Sorting Example.

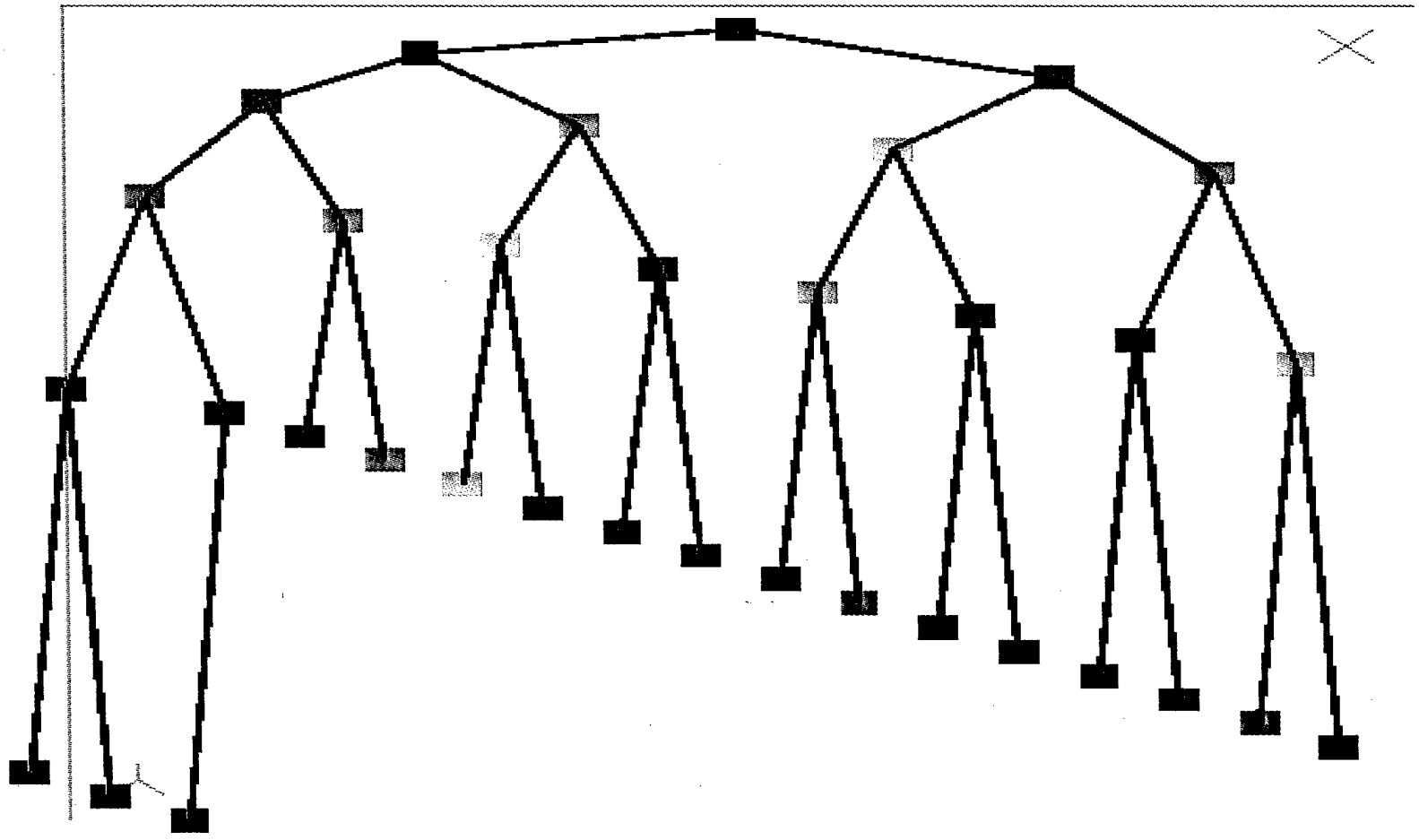


Figure 3: Zeus Sorting: View Down the Z Axis.

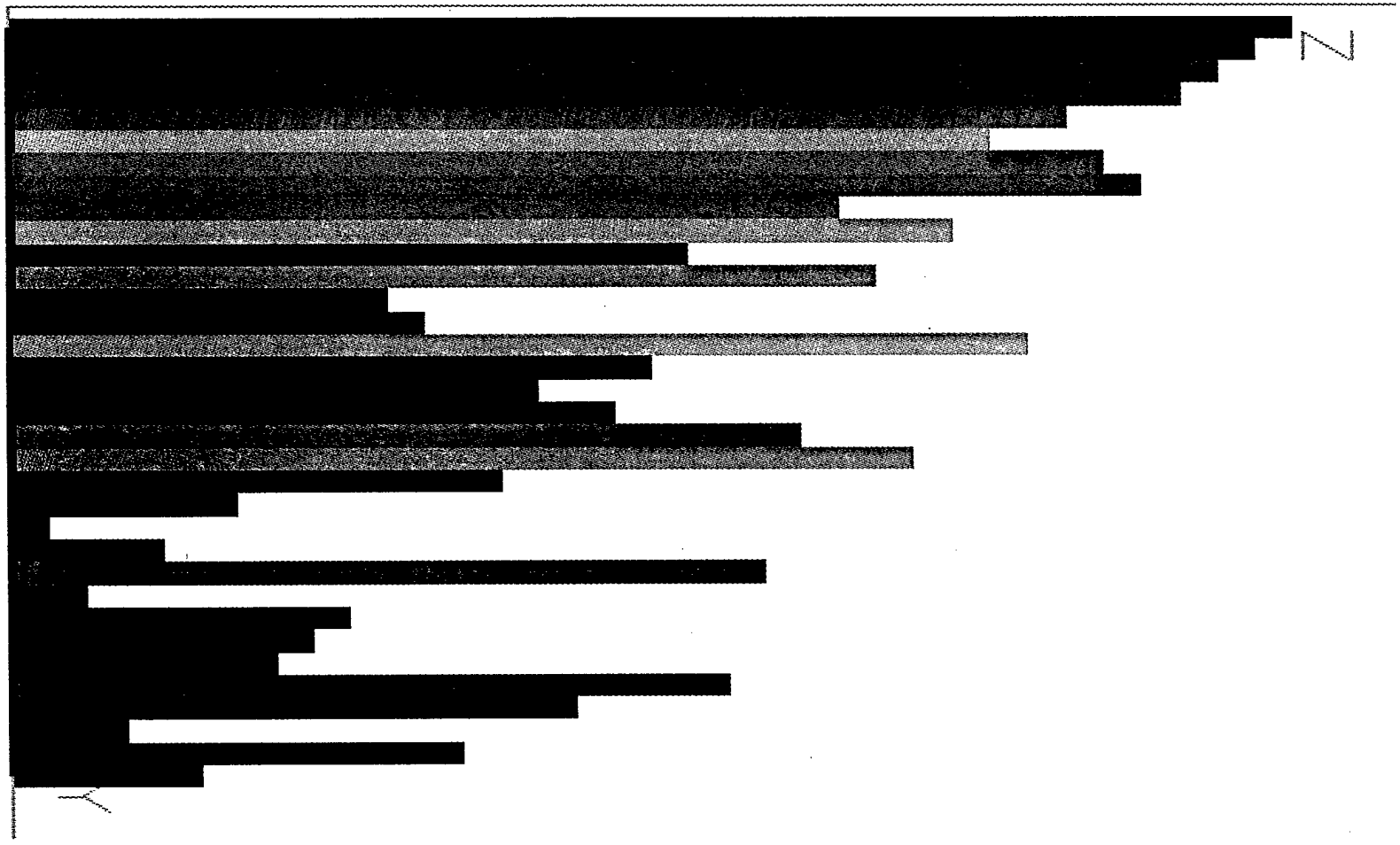


Figure 4: Zeus Sorting: View Down the X Axis.

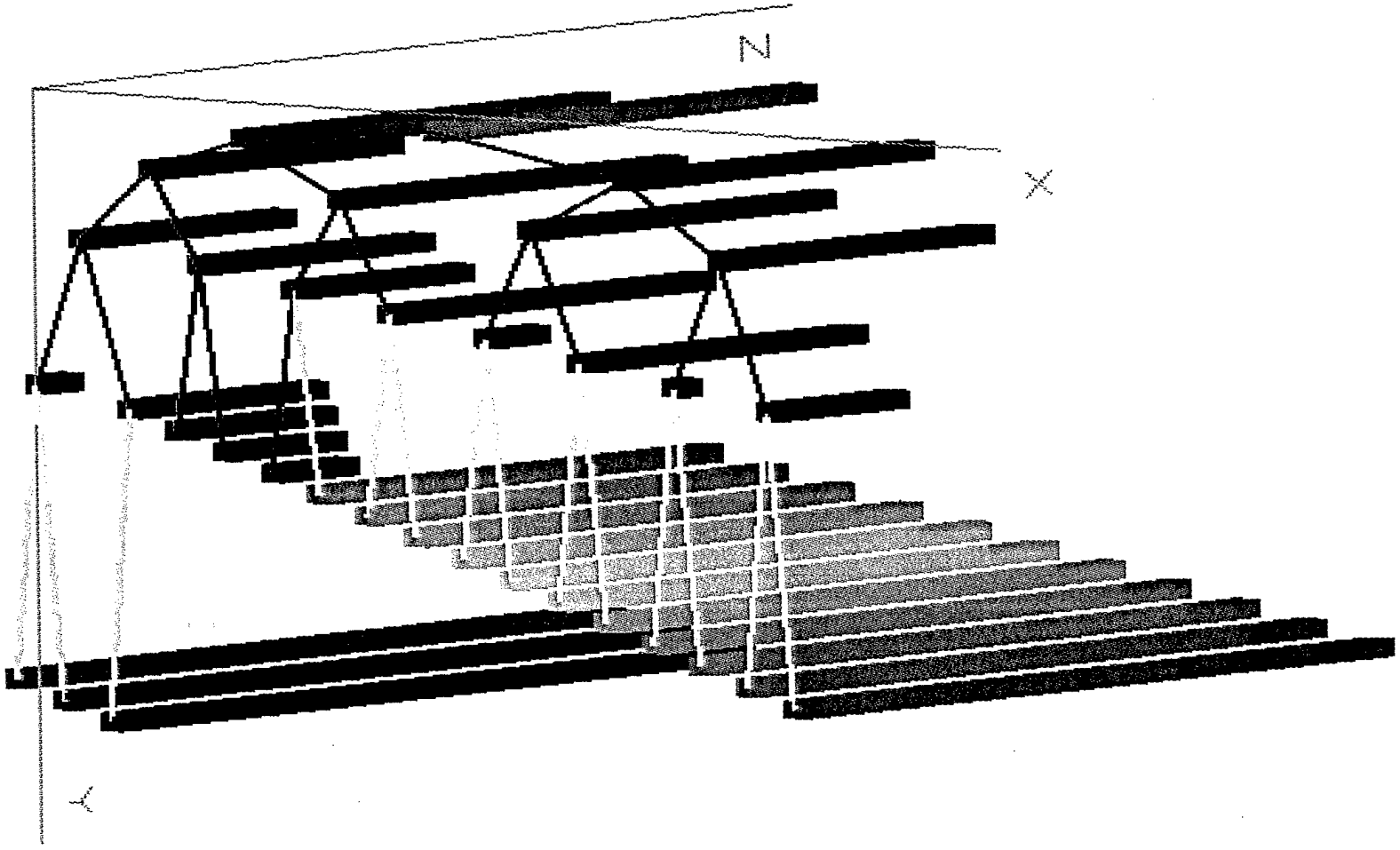


Figure 5: Zeus Sorting: View After Partial Completion.

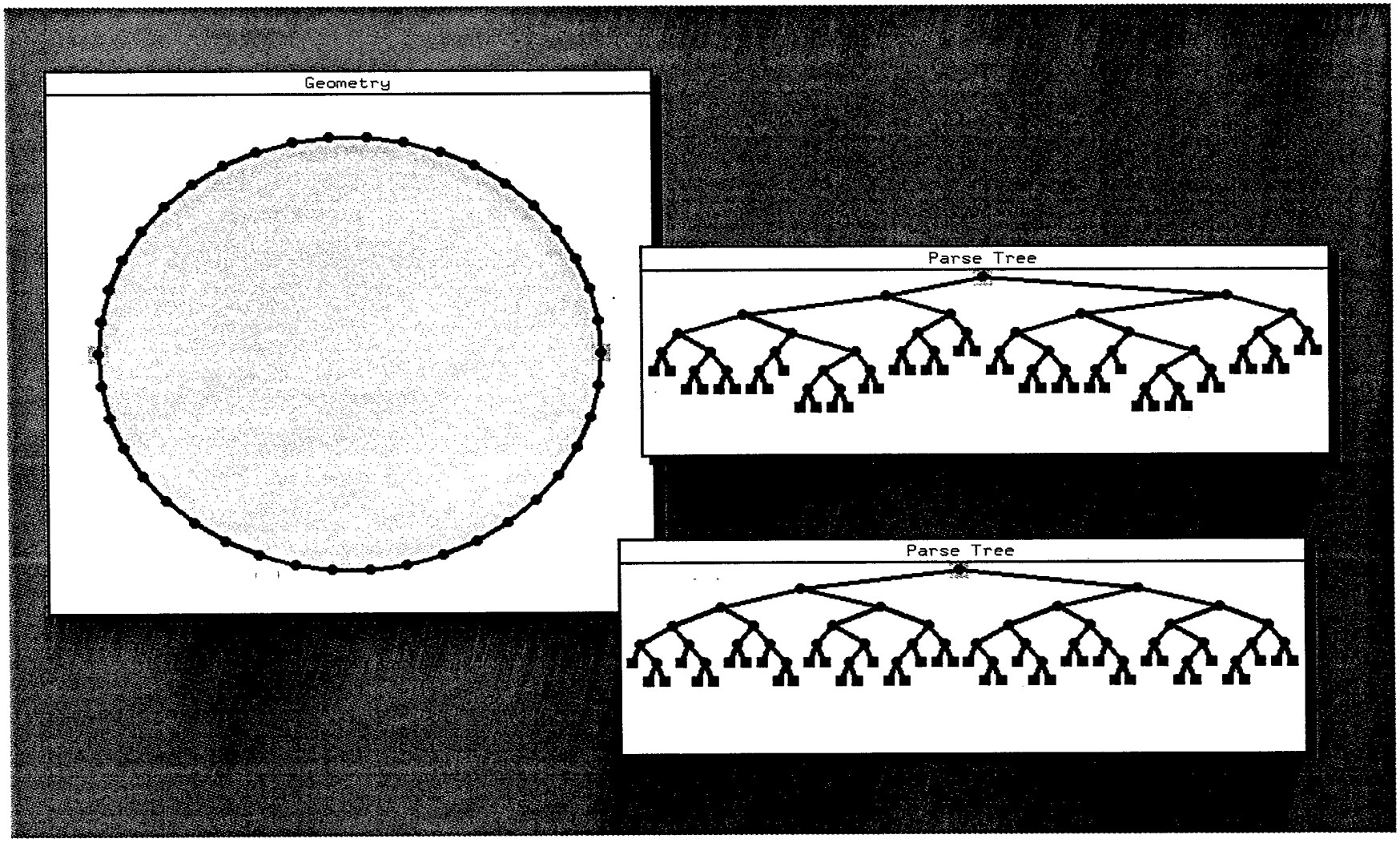


Figure 6: Zeus Algorithm Anomalies.

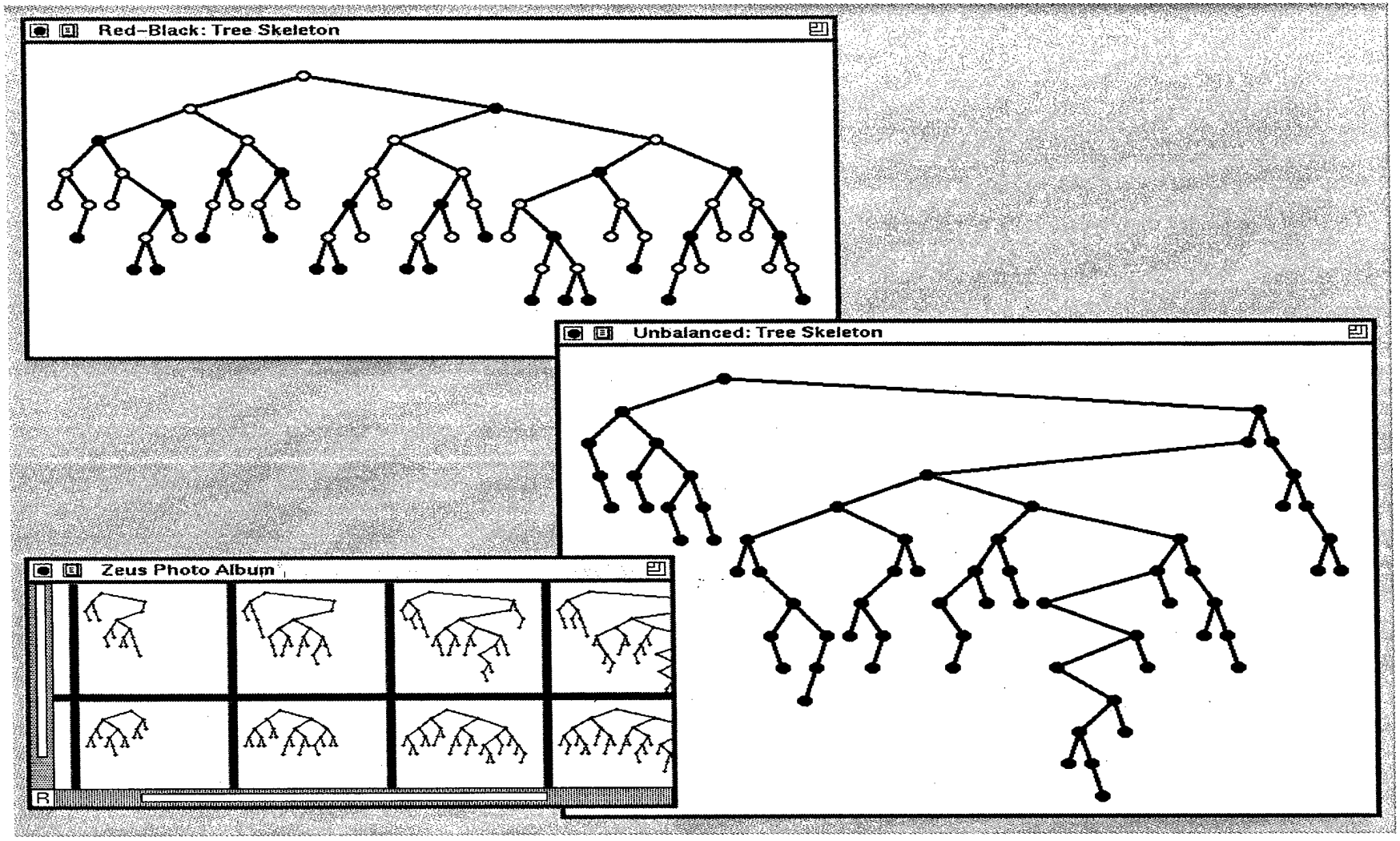


Figure 7: Zeus Algorithm Comparison.

Zeus in comparing implementations of various algorithms and the impact on the trees that develop within the supporting data structures. Figure 8 demonstrates how to use the tool for algorithm feature analysis, and Figure 9 shows how it could be used for execution flow analysis.

TANGO

Allows the animation of four abstract data types: trajectory, size, color, or visibility. It produces silent two-dimensional, black and white animations. The user annotated their code with algorithm operation calls which drove animation scenes invoked through C functions.

ANIM

Provides four operations: view, click, erase, and clear; and four drawing commands: line, text, box, and circle. Does not work on executable code. Allows static snapshots to be made of the animation for inclusion in documents.

Genie

Automatically creates displays of Pascal program data structures. Empirical evaluations suggest that the approach of using visualization is more effective than conventional program editing.

UWPI

The University of Washington Program Illustrator automatically provides visualizations for high-level abstract data structures designed by the programmer. It can animate abstract data structures in programs written in a subset of Pascal. However, it only gathers shallow information.

SEE

Provides a pretty printed version of the software code. It is a static representation.

TPM

Utilizes an annotated tree to depict program execution for the declarative language Prolog.

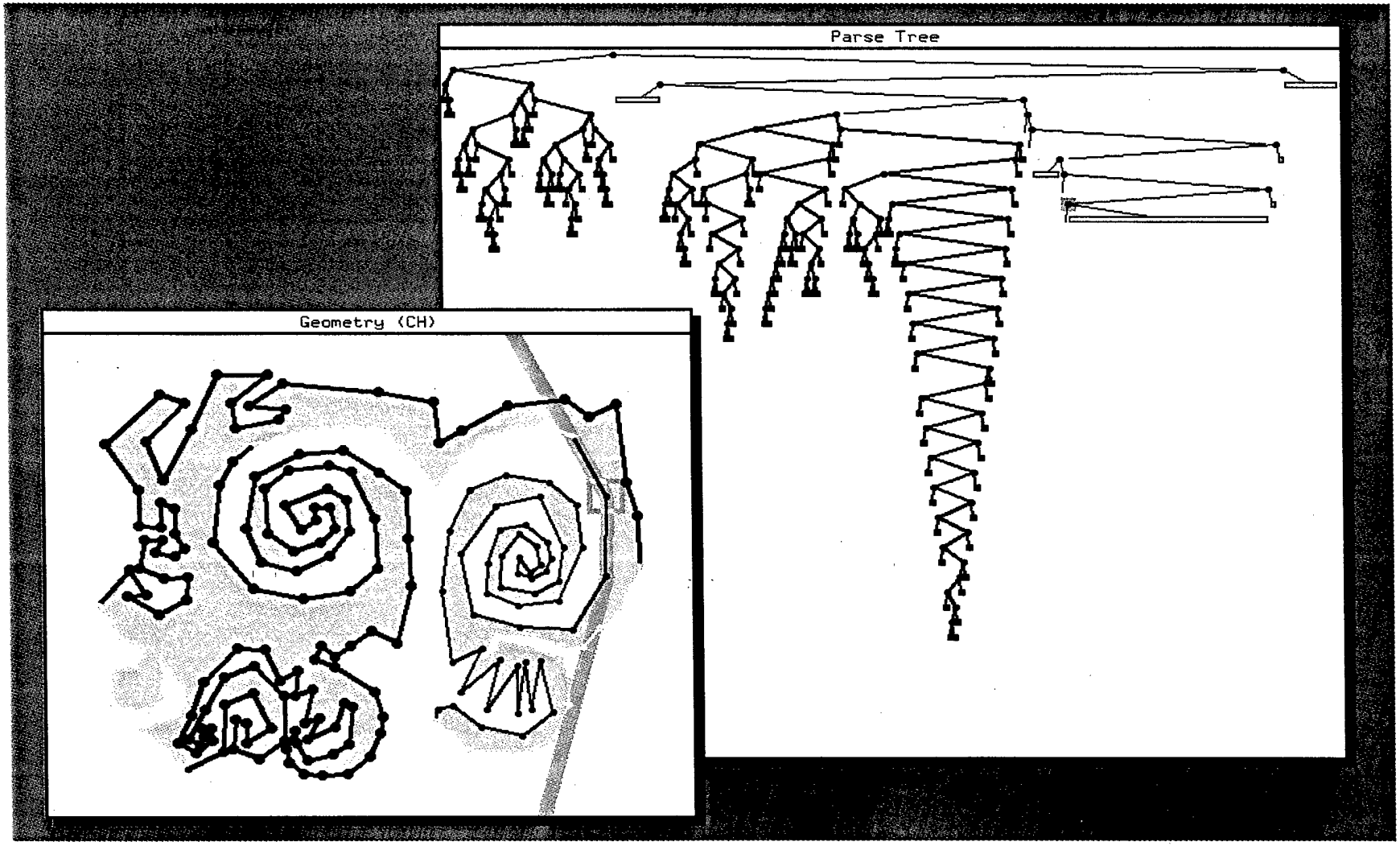


Figure 8: Zeus Algorithm Feature Analysis.

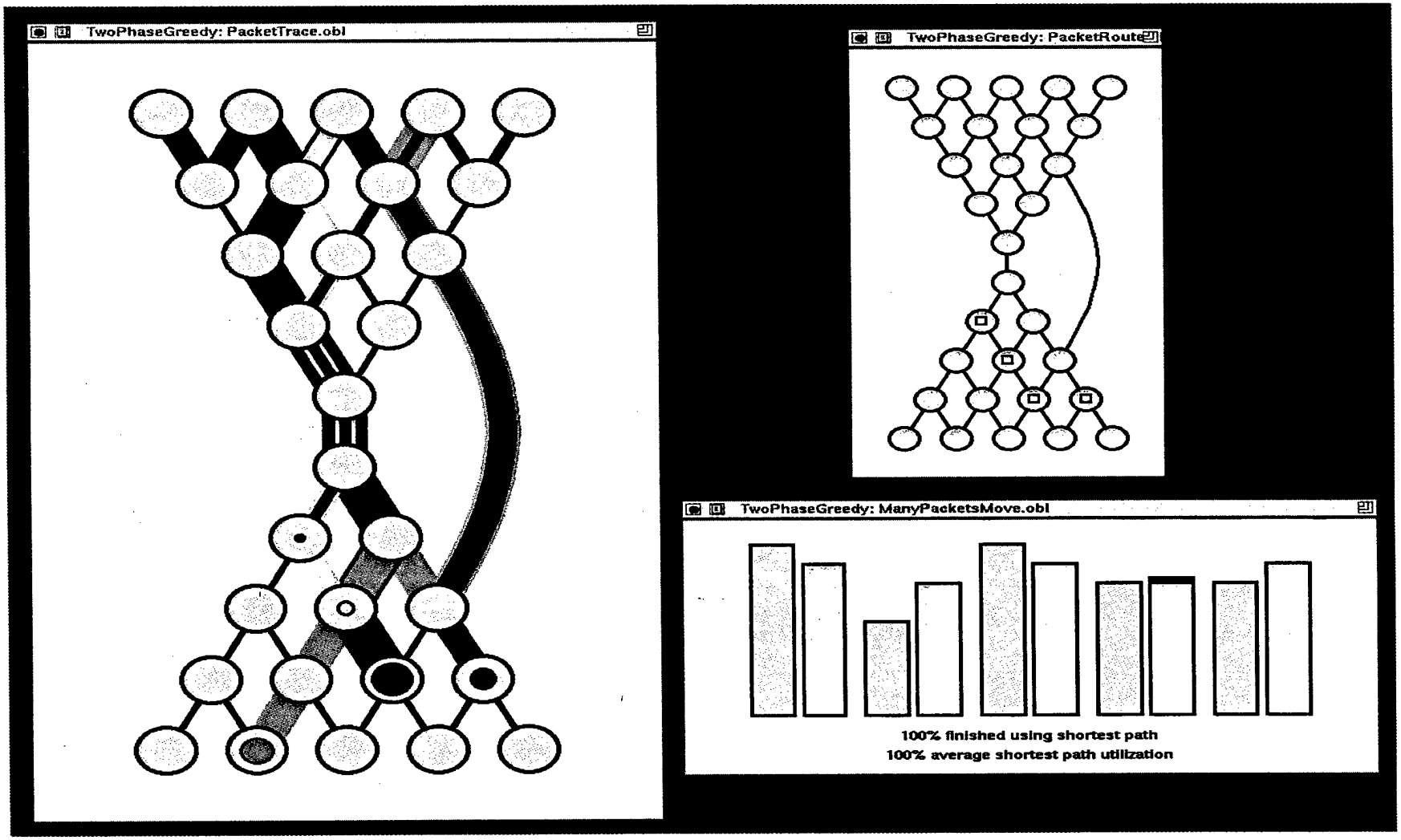


Figure 9: Zeus Execution Analysis.

Pavane

Provides declarative three-dimensional visualizations of concurrent programs written in Swarm. It is currently a research prototype.

LogoMedia

A prototype Logo programming environment to allow programmers to associate non-speech audio with program events.

ObjectCenter

Extends Unix's *dbx* to utilize simple graphics for static compile-time information about the source code. It is a commercial tool unlike the others described herein which are research tools.

5. Project/Tool Overview

Figure 10 illustrates the semantic view of the system. The selected program is executed within the SAVAnT environment. The program has been altered through a preprocessor to feed needed information to the controller for representation of the visual model. This information also is analyzed by the constraint system as specified by the requirements constraint language. The RQL program must be developed by the user for this feature of the environment to be utilized. The constraint monitor projection utilizes input from the executing program and the constraint system to determine the visual representation to depict. A controlling monitor alters execution control between the modules which are essentially functioning as coroutines. The entire system is embedded within Eigen/VR, a spin-off of the Muse system originally developed internally at Sandia National Laboratories. The Eigen/VR system provides a consistent interface to utilize virtual reality technologies. It is utilized by developing an OpenGL visual model which is then "plugged in" to Eigen/VR. Thus, the visual model generated by the SAVAnT system is an OpenGL model.

Functionality

Figure 11 shows what an initial program visualization looks like. This view depicts a program with one subroutine and a number of data structures, all of which are arrays. This model is generated automatically by scanning the original program to be visualized. A preprocessor was developed to automate the scan. The large circular object is the main program, and the smaller circular object is a subroutine. When the subroutine executes, the smaller object rotates and "orbits" the main program. Additional actions could be specified as desired by the user. The visual model may be altered by the development of additional routines. The placement and definition of the data structures are also automated. While the ability to select which data structures are to be represented is not yet implemented, the basic structure is in place to allow that functionality. Advanced development will allow the user to switch among models during the execution.

Figure 12 illustrates the same program at a later time. Note that the subroutine has altered position. Eigen/VR allows the user to "fly" around and into the various structures appearing in the visualization. Figure 13 shows a rotated view that the user sees while reorienting themselves through the "flight" capabilities, while Figure 14 shows an overhead view.

In moving about the system, it is easy to become disoriented. This is especially true in development of the visual model. As the system is developed to automatically place certain structures, the user may have difficulty determining the current orientation of the system in order to add additional features. Therefore, a feature is available to show the orientation of each object in relation to *X*, *Y*, and *Z* coordinates. This is achieved by embedding an axis within each object. Figure 15 shows the orientation when this feature is activated. The red axis is the *Z* axis, blue depicts the *X* axis, and the *Y* axis is denoted by the white arrows. Figure 16 is a different view of the orientation.

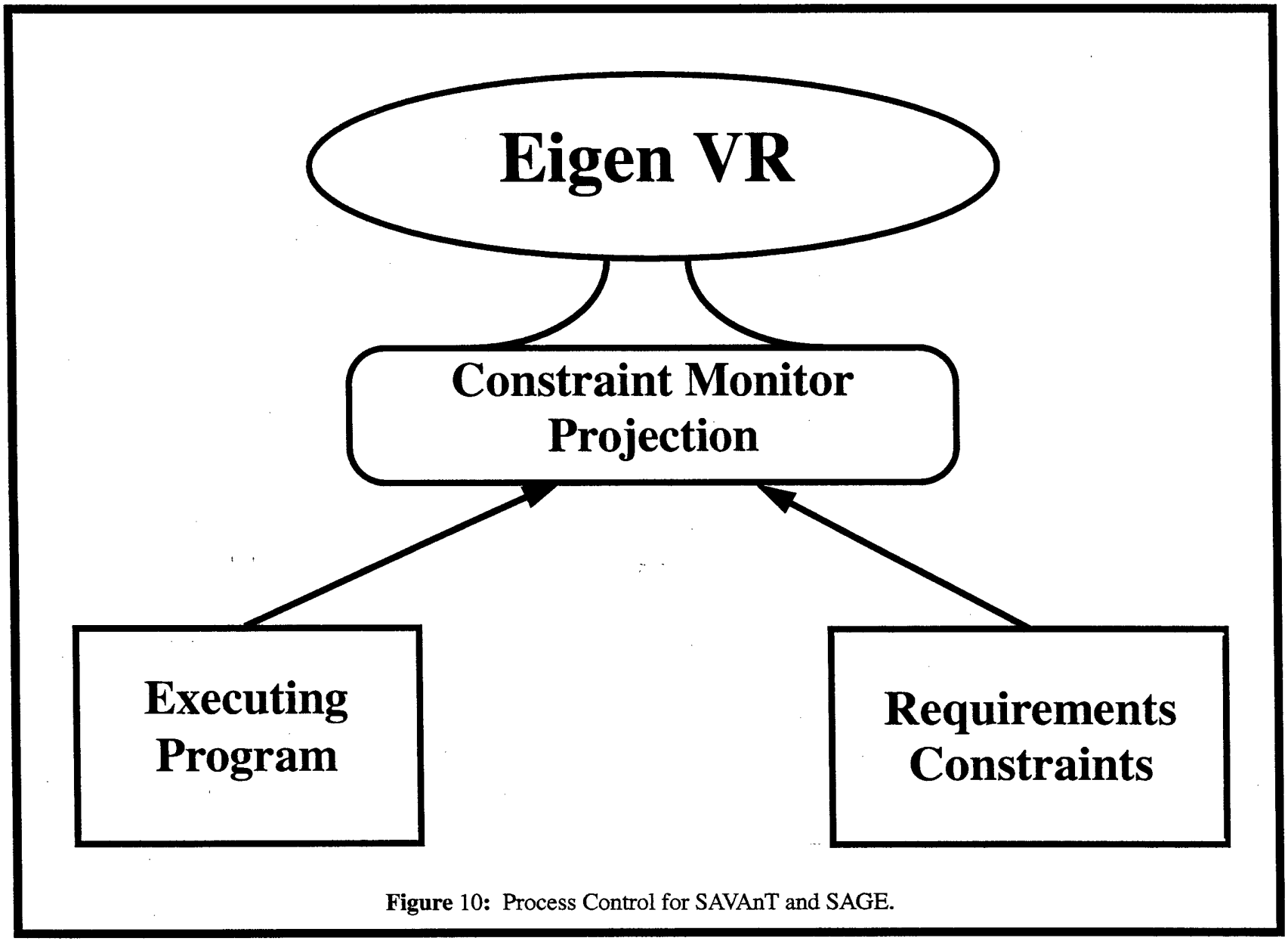


Figure 10: Process Control for SAVAnT and SAGE.

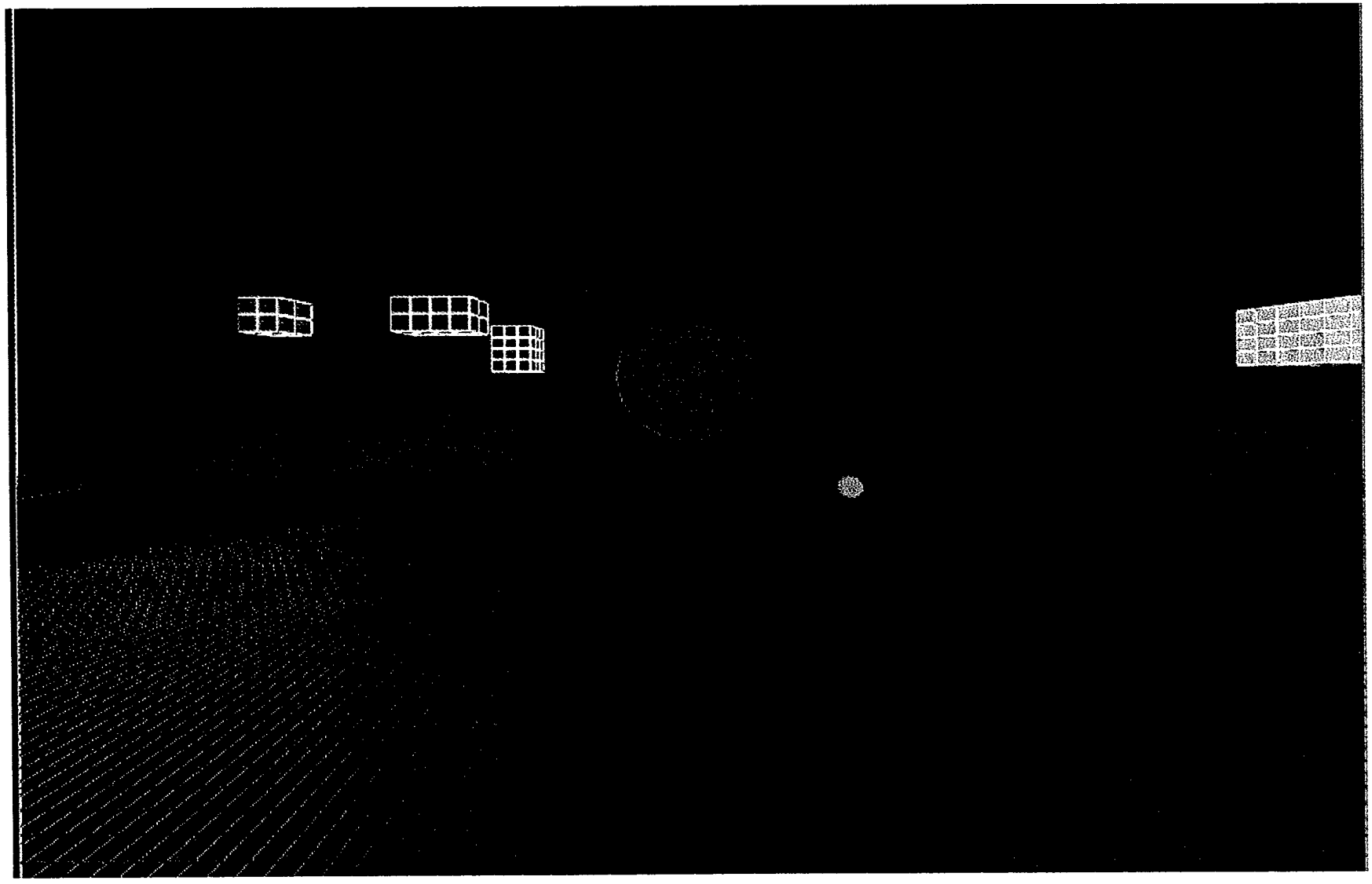


Figure 11: Sample Initial View of Generated Program Visual Model.

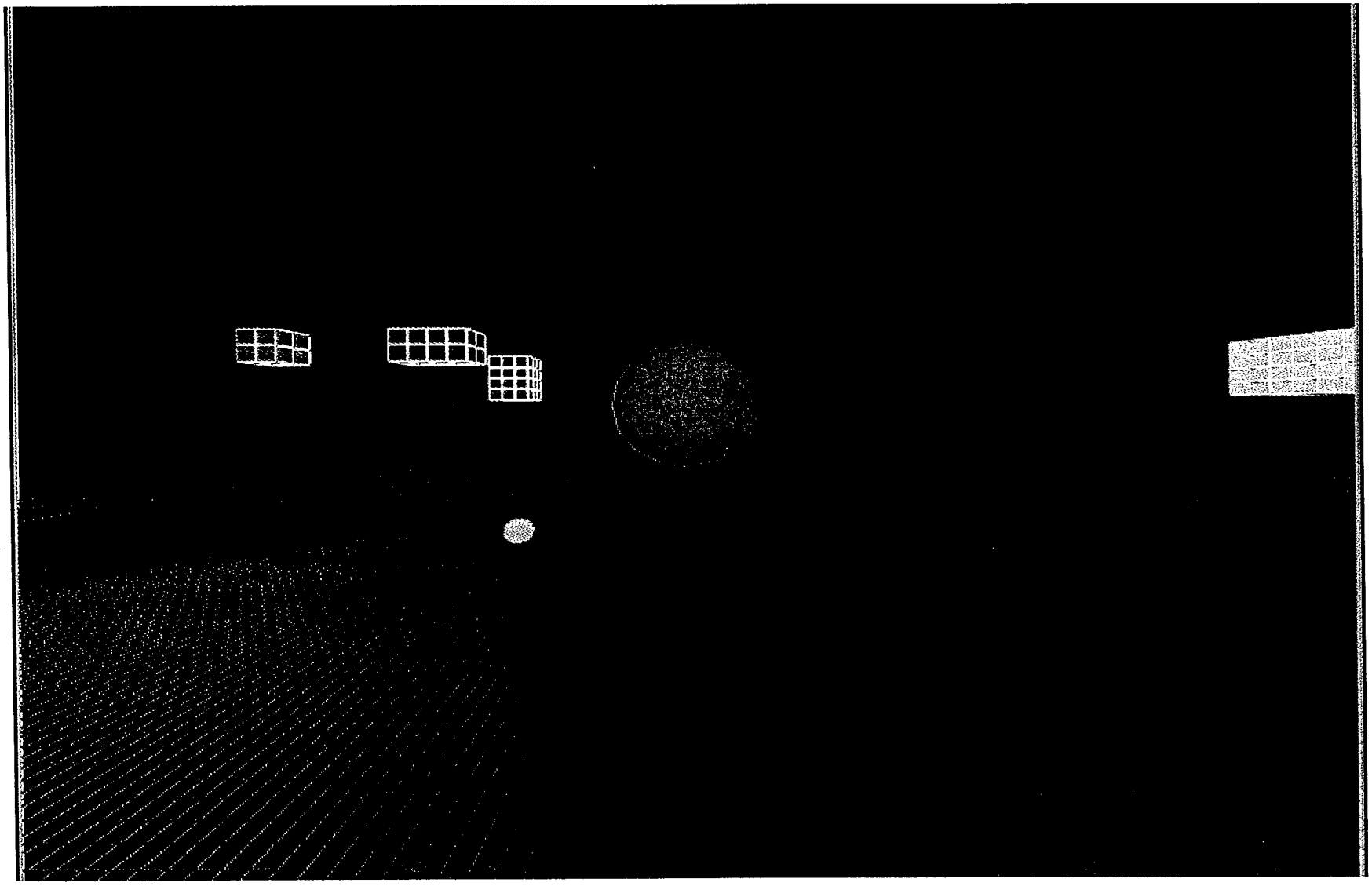


Figure 12: Subroutine Execution has Altered Position.

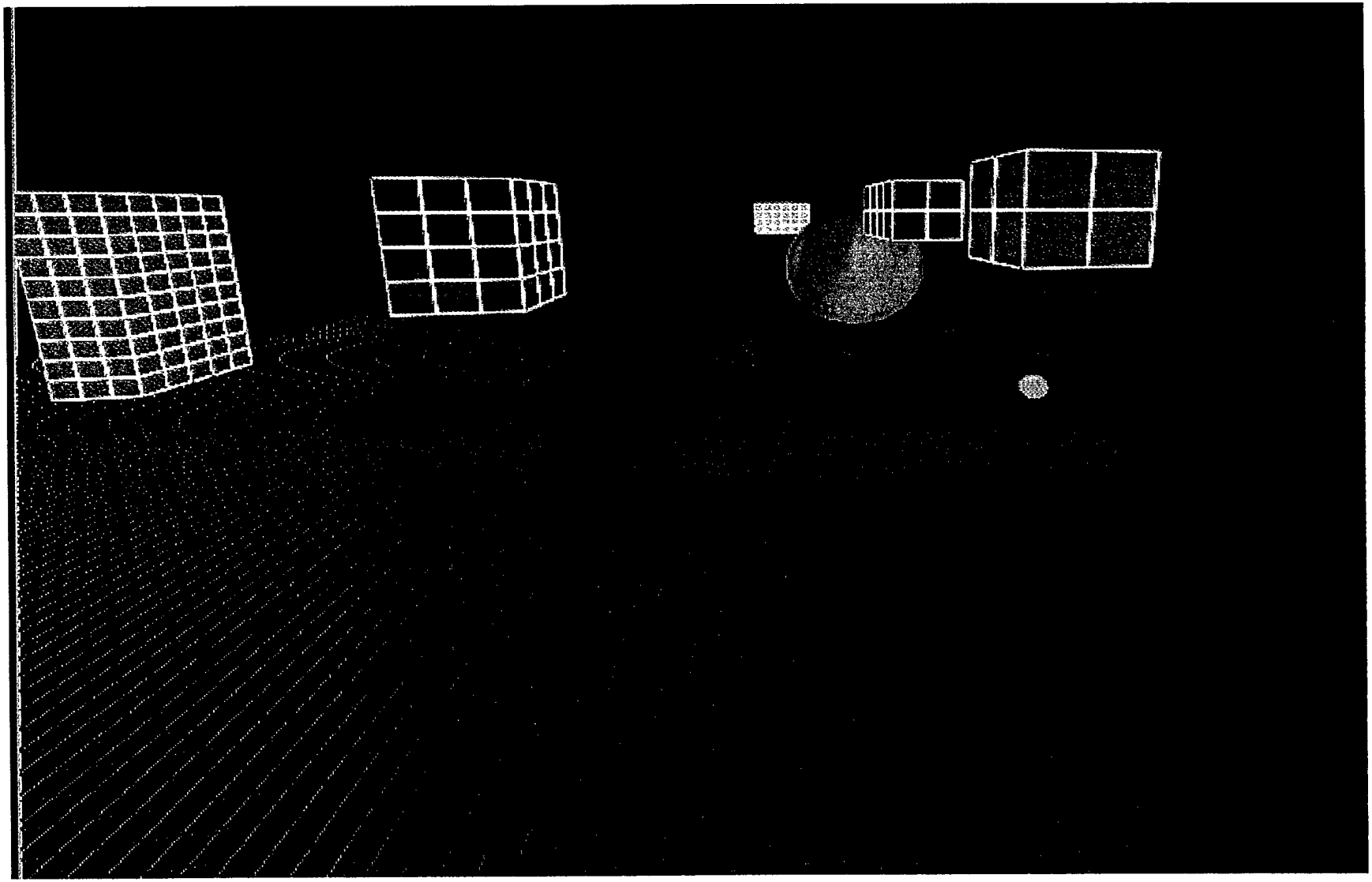


Figure 13: Original Model Rotated for Differennt View

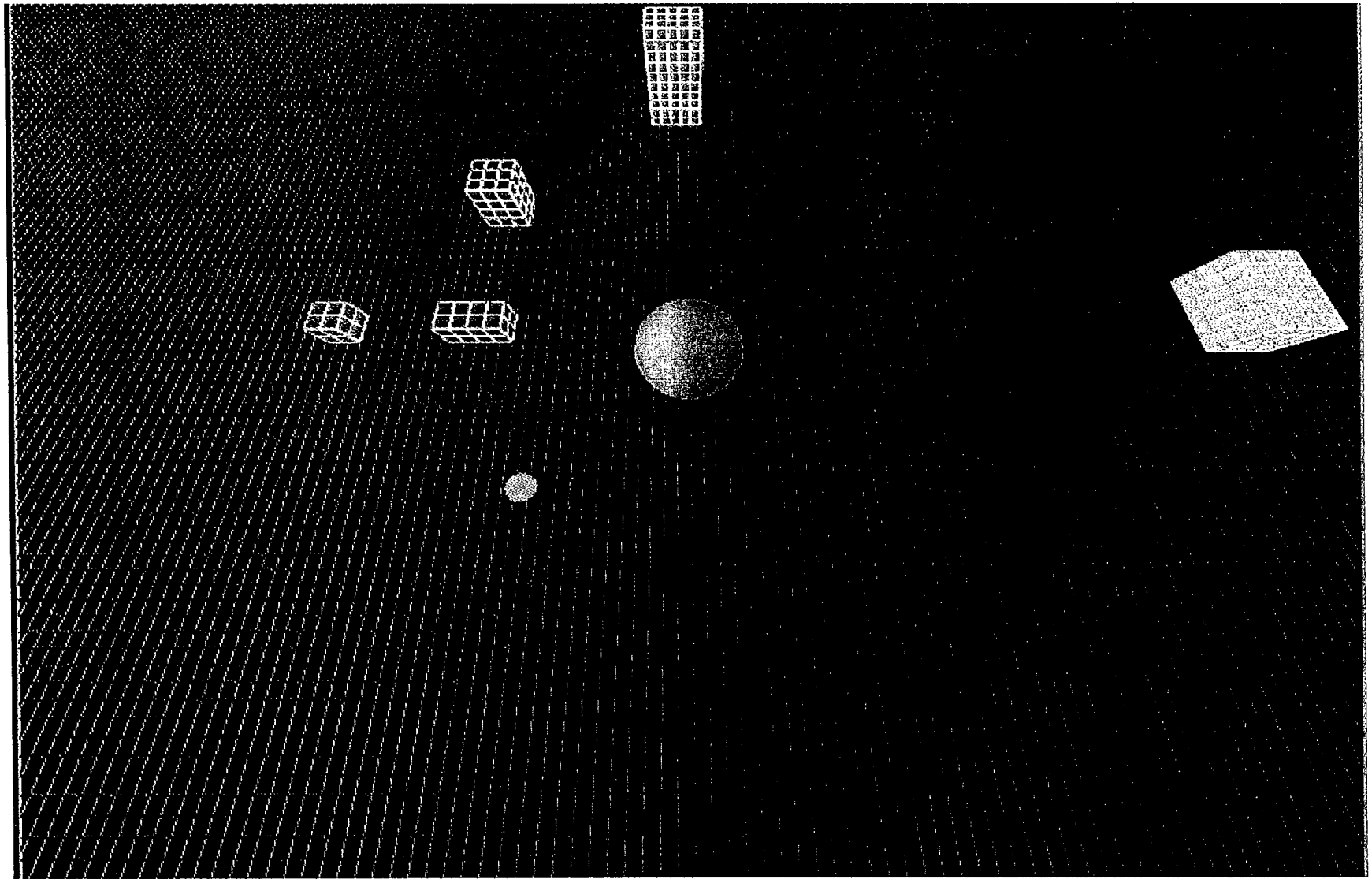


Figure 14: Overhead View of Visual Model.

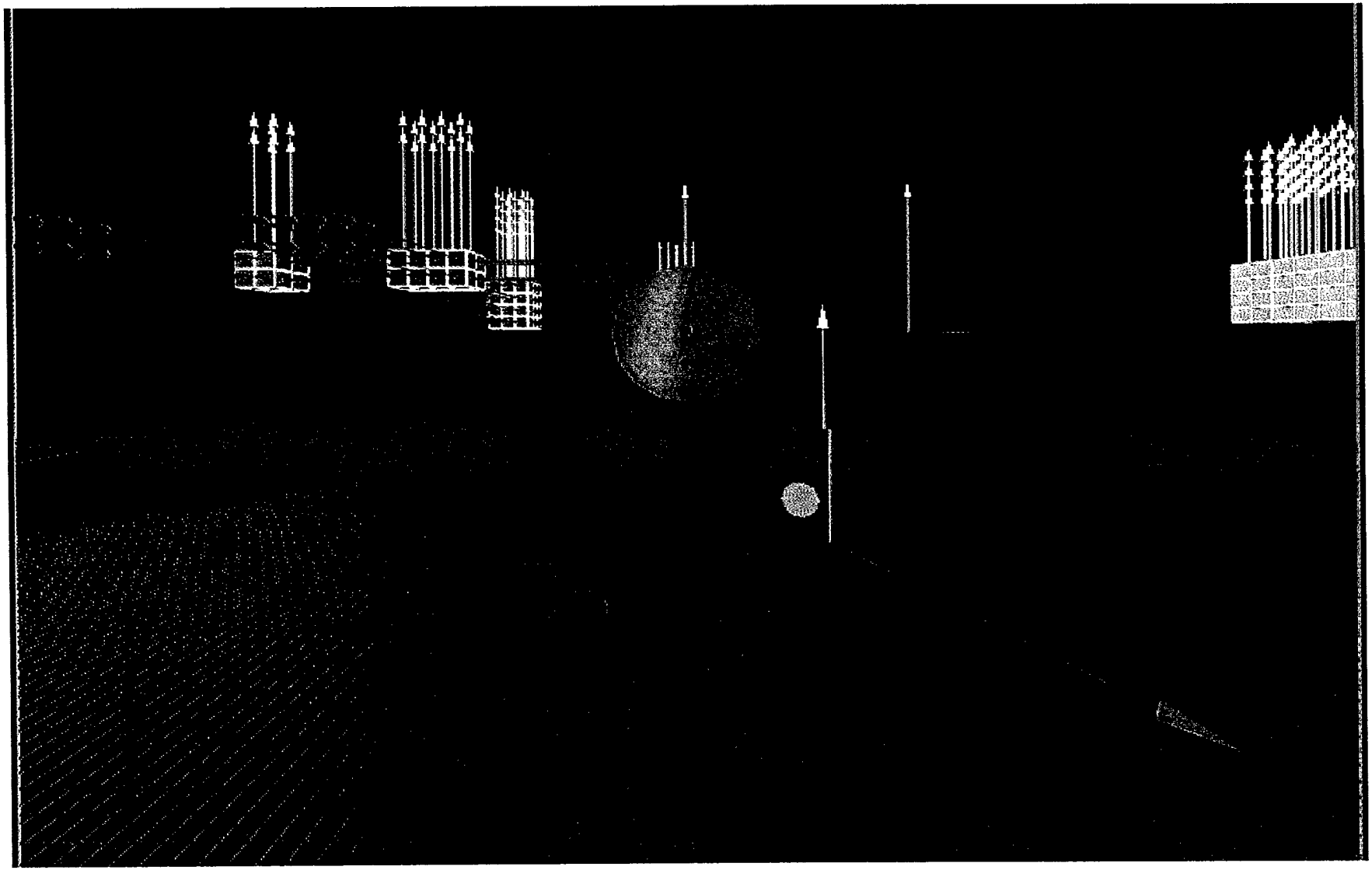


Figure 15: Visual Representation with Orientation Depicted.

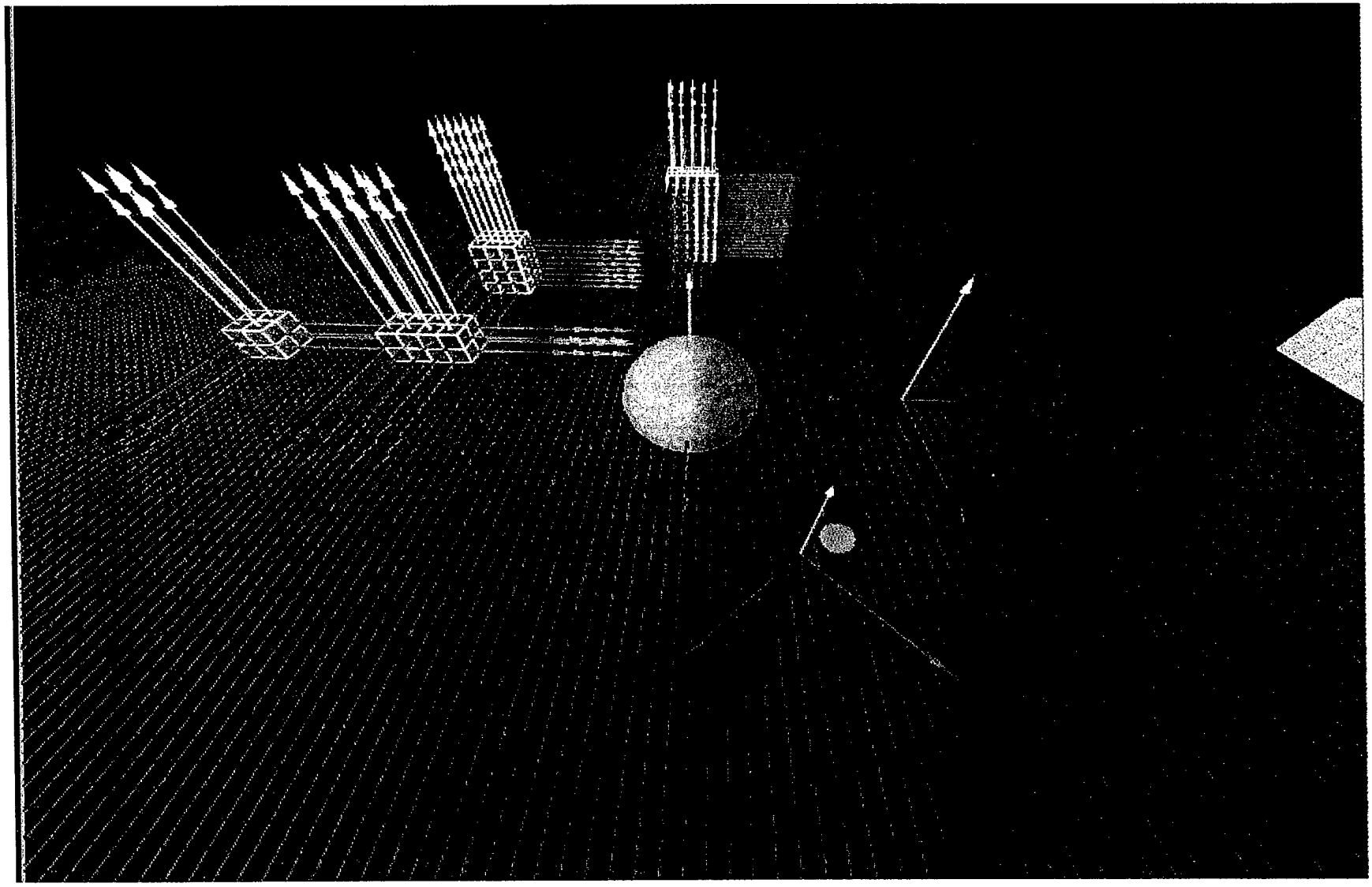


Figure 16: Orientation with Altered Perspective.

Differentiating Aspects

This system allows multiple tools to be utilized in the world view. To achieve this, the model must project rectangular vertical planes. Then texture mapping can be utilized to place the execution of other tools on those planes. This would be similar to viewing various monitors within the virtual environment. This allows the coordination of multiple tools and view. The differentiating aspects of this tool allows viewing of all aspects of the executing program, not just partial algorithm animation. Furthermore, this allows the visualization of various selected attributes within the model. In addition, the model provides the capability to assess the execution against specified requirements constraints. The user does not need to alter his code, as a preprocessor is provided to automatically insert the correct statements for implementation of the program as a coroutine process and feed the necessary information to the controller for generation of the visual model.

In addition, the following aspects are also evident:

Provides a multi-dimensional environment;
Allows user to define own visual model;
Allows user to extend existing visual model;
Allows user to specify actions for constraint violations;
Links the program to the requirements specifications;
Can be used to identify constraint violations before they occur (Future Work);
Can identify situations not outlined in the requirements specifications, thereby identifying incomplete specifications in certain cases;
Does not require the user to alter his code;
Can be used to identify unexpected algorithm actions;
Can be used to monitor link structures;
Provides greater flexibility in types of analysis possible; and,
User can construct a visual model corresponding to his mental model of program state space.

Computing Environment

This system was developed on a Sun Ultra 2 with Creator3D Graphics and a Freedom Series 3300 Graphics Accelerator. The visual model was developed in OpenGL to process programs written in the C programming language. The OpenGL model was developed for subsequent usage within the Eigen/VR system.

Components

The basic components of the system are the preprocessor which consists of a lexical analyzer and parser developed with LEX and YACC. And the visual controlling program. The constraint system has not been implemented as yet.

Intentionally Left Blank

6. SAVAnT Description (Software Attribute Visual Analysis Tool)

SAVAnT is a visual tool that generates a visual model of an executing C program. This model currently depicts the basic structures of the program, including functions and data structures. Additional attributes can be visualized if the desired visual models are prepared. The visual model allows the user an easy way to conceptualize the program in their own mental model. Traditional methods require the user to map the program solution space to a two dimensional model whereas SAVAnT allows a multiple dimensional mapping. In addition, the tool is structured to allow ease of customization. Thus, a user may alter the visual model to represent the action in whatever manner the user conceptualizes the program space. This allows a concrete representation to view and alter in understanding the program execution. Program comprehension is achieved faster with the additional visual information.

The system currently visualizes C programs that can be represented within a single file. While real applications typically consist of several files, due to time limitations, the prototype only processes a single file. An extension to process multiple file programs can easily be done by including the processing of an "include" statement. In addition, the preprocessor will not handle compiler directives. A brief review of the major components/aspects follows.

Preprocessor

The preprocessor consists of a lexical analyzer and parser that are used as input to LEX and YACC to generate the complete preprocessor. As the code is parsed, a symbol table is generated to be used between the executing program, the visual model routines, and the constraint system. information about structural aspects of the program and selected attributes is also collected to establish the initial visual model of the executing program. In addition, the preprocessor generates a new version of the executable program. This new version has appropriate statements inserted to feed execution data to the visual model and constraint system. However, the visual model does not depict the inserted statements.

Any necessary data is queried from the user in driving the preprocessor. This feature can be extended to allow the user to specify which data is to be visualized. However, it would be best to allow all of the data to be collected, and then to selectively invoke and eliminate desired aspects of the model as the execution progresses. This can be achieved through voice commands to the Eigen/VR system.

Additional information can be collected by expanding the parser and lexical analyzer routines. The entire language is implemented for the parser. This allows for complete functionality in future extensions by providing the appropriate "hooks" for expansion. Although the code recognizes all language features, the prototype does not process all features at present. The key consideration is the recursive nature of the algorithm which can create surprises in the parsing process if proper analysis is not done prior to implementation.

Executing Program

The executing program must be supplied by the user. It must be developed in C. The program must not utilize include files or compiler directives. The preprocessor will generate error messages if the program exceeds any limitations due to size. The problem can then be addressed by increasing the associated data structure within the parser or lexical analyzer and recompiling the routines to regenerate the preprocessor.

A new version of the program will be generated. This new version is the one that will actually be executed. Appropriate statements are inserted into the original program to drive the visual model. This provides an advantage of automating the process for the user. A disadvantage of this approach is that some errors might be masked by the process of altering the size of the code. This is a typical problem shared by all debuggers.

Visualization Routines

The visualization routines require structural input regarding the program to be visualized. This information is provided by the preprocessor. Figures 11-16 show the visualization of an actual program. The placement of the figures, their size, color and orientation are all achieved automatically based on the information provided by the preprocessor. An advantage of this approach is that it allows for the user to develop different visual models to be generated by the specified data. This allows the user to define their preferred model to coincide with their unique mental model of the executing code. This is important, because a single model may not provide sufficient information to address individual needs and understandings.

In addition, this approach eliminates the need for the user to alter their original code themselves. Further, the Eigen/VR environment allows for multiple tools to be utilized at once. With future expansion, this can significantly improve software surety capabilities as well as debugging productivity, and program comprehension. In addition, when the constraint monitor is fully implemented, this system will provide a unique capability to monitor correct execution as specified by requirement constraints. This will not identify all errors, but selected conditions can be monitored. If a violation occurs, the visual model will dramatically increase the user's ability for detection.

The model can also identify situations that have not been addressed by the requirements constraint language. This has an important impact. This is the first documentable technique to allow assessment of completeness for the software requirements specification. Current methods focus on proving that an implementation correctly implements a specification, but do not address the issue of whether the specification is correct or complete. While this technique will not fully resolve the completeness problem, it is a first step in identifying errors in completeness occurring during execution.

A disadvantage of this approach is that it focuses on the execution phase, thus the error has already occurred by the time it is visualized. However, this is a limitation only within the current prototype, and can be turned into a definite advantage. The advantage can be achieved by keying the routines to "look ahead" or "tentatively compute" ahead of any changes to be made in the pro-

gram or visual environment. This would allow earlier processing of the constraint monitor and allow the program to be halted or terminated safely. Essentially, this is the same concept utilized in processing software faults, just allowing the faults to be captured at a higher phase before a critical error can be initiated. While undoubtedly there will be code to address this issue within the program, the expanded functionality of the constraint system may allow for more extensive checking at any particular junction.

Constraint Monitor

The constraint monitor is described in greater detail within the next section. Basically, it functions similar to a data flow machine in determining which constraints apply at any given time. It utilizes the common symbol table routines, and basically has no action other than to monitor the execution of the code. So it compares applicable constraints to the changing execution values and program flow. If a violation occurs, the appropriate visual routines are invoked.

Controlling Routine

The controlling routine is very primitive in the current definition of the prototype. It basically directs the coroutines for switching of execution between the executing program, the visual routines, and the constraint monitor. Future extension to this routine will allow the user to selectively alter the visual models during execution, as well as collapse or expand world views.

Advantages

In summary, a major advantage of this work is that it will allow to monitor completeness of the specifications. In addition, this work has the potential to significantly increase software surety confidence, by providing an independent analysis of correct behavior. Further, this approach can significantly aid in the assessment of program behavior for systems using advanced control techniques such as neural net and fuzzy logic based controls. Additional advantages have been mentioned in previous sections.

Disadvantages

The main disadvantage of this work, is that the user must develop a requirements constraint program in order for the constraint monitoring system to function. This requires the user to be familiar with a new language, SAGE. However, this is not an a particularly onerous requirement. In addition, the user must have a similar platform available. In addition, until the extensions are added to process include statements and compiler directives, the tool cannot be used for real world applications. This disadvantage will be resolved once additional development is completed. In addition, the user must utilize current visual models until they develop their own models.

Future Extensions

Future work should focus on incorporating multiple world views, providing more control over the model by the user, and expanding available visual models. Later work should expand the environment to visual the specification phase of the software.

7. REQUIREMENTS CONSTRAINT LANGUAGE DESCRIPTION

What is meant by a constraint language? While a lot of research has been done in the area of constraint programming, the idea of a constraint language is unique to this application, expanding current techniques in software surety. To understand the type and purpose of this type of language, one must first understand the concept of a constraint.

Constraints

A constraint embodies the idea of enforced or defined limitations. This idea is inherent throughout most aspects of human endeavors, and thus is evident in many different types of applications. For example mathematically, constraints are precisely specifiable relations among several unknowns, each taking a value in a given domain. Consequently, many mathematical and geometrical definitions could be considered constraints. (The definition of a right triangle illustrates this point: a right triangle is a three sided polygon with one of the internal angles consisting of 90 degrees.) In the research area of computer programming, constraints are used to limit the values specific variables can be assigned. More specifically, upper and lower limits of an array subscript value are one type of constraint, restricting the subscript value to be within the range of the upper and lower limits. Altogether, the constraint concept is extremely powerful and has been used to address a large variety of application areas through the development of various basic constraint systems.

Constraint Systems

Basic constraint systems are systems of inference on partial information that provide the ability to perform such functions as constraint propagation, entailment, satisfaction, normalization, and optimization. Classic illustrations of constraint systems appear throughout many fields. Typically, the area of operations research investigates many issues specifically related to constraint analysis. For example in operations research, often a set of equations must be solved with specified constraints to either optimize or minimize a particular value or values. However within the last decade, researchers have realized that unifying efforts to exploit ideas for constraint analysis via programming under a common conceptual and practical framework provides a more powerful approach to programming, modeling, and problem solving rather than developing disjunct basic constraint systems.

Consequently, constraint programming ties together the use of basic constraint systems with programming languages; thereby allowing more precise specification of how constraints are generated, combined, and processed. Expanding the utility of these systems by incorporating them with programming languages provides a more expressive unified framework; allowing the user to easily generate, manipulate, and test constraints--clearly, a more powerful computational framework. Examples of such frameworks include constraint logic programming and concurrent constraint programming systems. Examples of specific systems include cc(fd) [20], clp(fd) [6], ECL^{i}PS^e [10], CIAO [12], and Oz [19]. These systems generally consist of two levels, the under-

lying constraint system, and the programming language level.

Current research with constraint programming shows that constraints can be used in a number of different ways. A few typical applications are to represent knowledge, guide searches, prune useless branches, filter queries, describe process communication, and describe synchronization. The goal of constraint programming is to determine whether a solution exists that satisfies all constraints, to identify one or all solutions, to determine whether a partial instantiation can be extended to a full solution, or to find an optimal solution relative to a given cost function.

Accordingly, this type of programming has been used in many different application areas including artificial intelligence, databases, operations research, user interfaces, concurrency, robotics and control theory. A new area for application investigated by the work described by this report is the area of software engineering. The work described within this paper applies and expands the concept of constraint programming to address software surety issues within the area of software engineering research by defining a requirements constraint language (RQL SAGE).

Software Attribute Generic Evaluation

The requirements constraint language SAGE allows the development of programs to perform constraint analysis on executing programs as a monitoring process. A program written in this language is used to provide an independent audit of an executing program to verify that it is executing as planned and expected. This allows unexpected program states to be identified and addressed before critical action occurs that could cause loss of life or some other unexpected devastating, costly, undesired action. This is most helpful in embedded systems.

The idea of a requirements constraint language expands the basic constraint programming paradigm to a higher level. A requirements constraint language is expressed in a very high level language utilizing functions and operations to address higher level ideas and conceptualizations related to a system requirements specification, in addition to more common lower level functions dealing with variables, registers, various arithmetic, character and logical operations, and memory management. The language primarily expresses what should be done, rather than how it is done (although some aspects of how it is done can be specified as a constraint); and provides mapping capabilities to an underlying program representation that implements the required functionality. A requirements constraint program monitors the execution of the lower level program to ascertain that constraints are not violated. It does this through a very high level pattern assessment linked to the executing program.

Thus, a program written in a requirements constraint language functions as a bridge between the requirements and the actual implementation. It also provides a second, independent assessment of the correct functioning of the targeted implementation; and while it does not provide a second calculation for comparison, it does function as an independent monitor similar to established fault tolerant techniques. This provides a new technique for assessing software surety. As future advancements provide improved performance for this approach, it can be incorporated appropriately during run-time to prohibit select, critical errors.

Advantages

The use of the requirements constraint language is important to this application for several reasons. Usage of this language provides a technique to address actual software surety issues during the execution phase of the software life cycle. As performance issues are addressed, this approach can be used to monitor and approve program execution before critical sections of the code can be executed for high assurance systems. Preliminary work focuses on monitoring the correct execution of critical code after it has executed, but with recent advancements in performance issues and in the magnitude of constraints being evaluated, it is reasonable to predict that the code can be structured to allow the monitoring assessment to be conducted just prior to execution, thereby providing a independent auditing function as a software surety technique to ensure that the executing code only executes in acceptable, expected ways.

In addition to providing monitoring capabilities for the correct execution of critical code, the RQL SAGE provides input back to the SAVAnT system to generate visual and other stimulus for identifying unexpected occurrences within the executing code. In addition, SAGE provides a second opinion through the auspices of an independent auditor on the correctness of the code execution--an established fault tolerant technique. Other advantages of this techniques include the ability to assess trade-offs between requirements constraints where conflicts occur, and most importantly, the ability to identify specification errors or omissions. Particularly significant, the ability to identify specification errors addresses an unsolved problem under review for many years by the software engineering community; the problem of incorrect specifications. Formal methods have made great advances in mathematically proving that a particular program precisely implements a given specification; however, those methods do not provide any information as to the correctness of the specification. SAGE in conjunction with SAVAnT provides a mechanism to identify errors and discrepancies within the specification itself. As many people have been working on this problem with no solutions to date, our approach is a major advancement in this research area.

Disadvantages

However, as with any technique, several drawbacks exist with using this approach. The most significant is that the user must learn the requirements constraint language SAGE; and in addition, the user must be familiar with the requirement specifications for the target program in order to encode the appropriate constraints depicting the specified requirements. Yet, as similar requirements are often required for implementation of current technologies; having to learn SAGE and familiarize oneself with the application's requirements specifications should not be considered particularly onerous requirements. Other technical knowledge or skills needed to apply this technique include knowledge of the SAVAnT system and of the target program to be monitored. The user must be familiar with the SAVAnT system in order to specify the appropriate/desired visual effect to occur for each situation of interest; while the user must be familiar with the target program in order to establish the appropriate links between the executing program and the monitoring SAGE code. Appropriate visual tool sets will be developed within SAVAnT to facilitate these efforts and depending upon how the requirements were initially specified for the system, the program links may be easily determined.

Two restrictions limit application of this technique. First, the constraint monitoring cannot be applied to all of the code until performance improvements have been achieved. This is not as great a problem as it might seem, because the most critical portions of the code can be targeted for monitoring initially; and performance advancements in constraint analysis are nearly adequate to handle monitoring of the entire code--so this problem will be resolved in time. Finally, this approach does not allow monitoring of timing constraints as currently planned for implementation. Later developments can address this shortcoming.

Implementation

SAGE utilizes C as the underlying language base. Language extensions are used to expand the ability to define concepts, objects, and semantic patterns of interest for monitoring purposes. Mapping capabilities are also provided to allow mappings between the targeted executable program and the RQL state space. The mappings identify what state space information will be needed, and potentially can be used to drive the preprocessor in preparing the executable code, by identifying which state spaces are of interest for observation--a possible future extension. Mappings are limited to measuring program state spaces. In analyzing semantic issues, the concepts must be translatable into specific program states. The mapping capability allows extensive reusability of function constraints; such reusable definitions will greatly reduce development time as experience with the system occurs and suitable libraries are developed.

Execution patterns can be mapped to program slices through regular expressions. This allows the execution sequence of the target program to be assessed. A common approach for checking prior to execution of critical code is to check the values of flag variables, however, the SAGE RQL allows monitoring of the sequence invoked in setting the variables. This allows identification of an improper execution sequence, a potential error.

The SAGE RQL program runs in conjunction with a constraint analysis system incorporating artificial intelligence technology, data-flow technology, and (with future development) neural network technology to expand pattern analysis for higher semantic reasoning. The constraints are specified along with the state variables monitored by the constraints. When state information is received, it is mapped to corresponding constraints. When the required data is available the appropriate rules for evaluation are fired. The constraints and their relevant variable mappings are maintained in a sparse matrix indexed by standard scoping rules.

As the target program executes, state space information is generated to drive the visual representation and the SAGE RQL monitor. Thus the system is basically event driven. The variables, or rather their specified mappings, are indexed into the constraint matrix to identify related constraints. If adequate information is available to evaluate a constraint, it is selected for analysis; otherwise, the information is either saved for later analysis, or a partial analysis is conducted if possible. The constraint analysis system identifies conflicting constraints and identifies what happens if constraints are violated. This allows the user to verify that appropriate priorities have been established between conflicting requirements.

Operations

The basic functions, capabilities, and operators defined within SAGE as extensions to the C language include support for first order logic: logical quantifiers, implication operators, partially defined expressions, as well as access type collections, type constructors, bounded quantifiers, mapping constructors, and pattern notation. Examples of most of these can be seen in languages such as Anna and Refine.

Additional operations include: hence, precedes, follows, subsumes, distinct, disallow, occurs, and sequenced. *Hence* used in conjunction with a logical expression (*e.g.* if *a* hence *b*), indicates that the condition following *b* must not have been true prior to the occurrence of condition *a*, and after *a* has occurred, *b* must hold true. *Precedes* identifies states (or execution patterns) that must occur prior to other states or patterns. *Follows* is similar except that it identifies states that occur after a known state. It does not address the immediacy of the occurrence, just that the specified state occurs sometime after the state initiating the constraint. These two operators allow greater flexibility in defining and specifying constraint conditions. (Generally, order of appearance can be used to indicate dependencies among variable states in programming languages. However in this constraint system, that approach is insufficient to identify required relationships and does not support constraint orthogonality.)

Subsumes indicates that constraints related to a particular state *i* are applied to another state *j* as a partial definition of the constraint requirements for the new state *j*. This allows reusability of definitions. *Distinct* specifies that a state or event, normally occurring as part of a sequence or grouping, appears temporarily disjunct from that association. *Disallow* designates a guard against the occurrence of a noted state, condition, event, or pattern. *Occurs* defines a grouping or selection of states that must occur in relation to one another without establishing a definitive order. *Sequenced* determines an ordering of event or state occurrences.

The new operations are important in establishing appropriate relationships between the ordering of the specified requirement constraints. The normal ordering of control evident in general purpose languages does not apply to the constraint definitions, requiring additional syntactic support in specifying ordering relations. When a constraint is defined, it does not apply until specified by the defined operations. This allows greater freedom in the application and release of constraints onto the program state space. Thus a particular constraint may only be applicable under particular conditions. Normal sequence of execution flow does apply within the definitions. This approach avoids forcing the constraint program into a two dimensional flow mapping.

Syntax Issues

The syntax for these operations is depicted in Figure 17. A constraint specifies one or more mathematical expressions and or conditions that apply to the executing program being monitored. A condition represents mathematical or logical expressions related to the requirements constraint language monitoring program; while a state is characterized by a collection and/or sequence of constraints and conditions. A bag provides a convenient way to reference a collection of orthogonal or heterogeneous qualifiers such as execution patterns, states, and conditions. Commas should

[constraint(s)|state|condition(s)]: **Hence** {*constraint(s)*}
[bag|constraint(s)]: **Precedes** {*bag|constraint(s)*}
[bag]: **Follows** {*bag*}
[state|condition(s)]: **Subsumes** {*bag*}
Distinct {*constraint(s)*}
Disallow {*state|condition(s)*}
Occurs {*event|bag|constraints(s)*}
Sequenced {*state|condition, state|condition, ...* }

Figure 17: SAGE Sequence Operations Syntax.

separate multiple constraints, states, conditions, or bags.

A simple label naming convention allows constraints to be referenced by name. The constraints' names can be specified when using the operations described above. In addition, a name can be applied to a group of constraints. Alternatively, a constraint may be specified instead of using a named reference. However, a constraint may only be defined once. Definitions of constraints may appear wherever variable definitions are allowed.

Examples

A surjection function is a mathematical function that is an onto mapping. That is, a function from A to B is an onto function if every object of set A maps onto an object in set B, and every object in set B is mapped onto by one or more elements of set A. Thus the function "generates" a mapping to every element in set B by applying the function to set A. Figure 18 illustrates the constraints that might be coded to represent this type of function.

$\forall$ x in A -- x $\Rightarrow$ y of B;
$\forall$ y in B -- $\exists$ x in A occurs{ x $\Rightarrow$ y of B};
Surjection_Count == $\Sigma \forall$ A $\Rightarrow$ B;

Figure 18: SAGE Constraints For Count of Surjection Mappings

We read these constraints as: For every x that is an element in set A, x maps to an element y of set B. For every y that is an element of B, there exists an element x in set A such that x maps to that element y of set B. The value of Surjection_Count is the sum of all possible mappings of A onto B. The representation of these constraints provide a greater detail of semantic knowledge than is generally inherent in simple programming code. This can be seen by looking at the following sample code. This code implements the Surjection_Count function defined by the above constraints, that determines the number of possible onto mappings that can be achieved between two groups of objects--the number of ways of mapping set A onto set B. Compare this to one possible implementation of the function as depicted in Figure 19.

In this particular example, the constraints cannot verify the logical correctness of the algorithm. However, by assessing the data values and structures that are generated by an algorithm, some errors can be identified.

```
int power( a, b )
    int a, b;
{
    if( b == 1 )return( a );
    return( a * power( a, b - 1 ) );

} /* Calculate a raised to b */


int fact( a )
{
    if( a == 1 )return( 1 );
    return( a * fact( a - 1 ) );

} /* Calculate a! */


int comb( a, i )
    int a, i;
{
    int result;
    result = fact( a ) / ( fact( i ) * fact( a - 1 ) );
    return( result );

} /* Calculate the combinatorial of a objects taken i at a time */


int Surjection_Count( a, b )
    int a, b;
{
    int i, sum;
    if( a < b )return( 0 );
    sum = power( b, a );
    for( i = 1; i <= b - 1; i++ ){
        sum = sum + ( power( -1, i ) * comb( b, i ) * ( power( ( b - i ), a ) ) );
    }
    return( sum );

} /* Calculate the number of mappings of a onto b */
```

Figure 19: C Functions Counting Surjection Mappings

8. Examples of Use (identification of errors)

Figures 20 (*a-d*) illustrate example usage of the prototype tool. Figures *a* and *b* show various programs, and how they would appear initially. Of course, the actual visualization would appear similar to Figures 11-16. These examples are illustrated to minimize space. Figures *c* and *d* illustrate error conditions that could occur as desired by the user. Additional examples may include the following:

Example 1: Flag condition is set and a key variable is changed when it should be constant under specified conditions. (perhaps side effect)

Example 2: Specific conditions are met; and statements are executed when they should be barred from execution (e.g. action taking place in an unsafe condition)

Example 3: Timing constraints are not met (will not be able to handle this in present version)

Example 4: Variable is not processed within an array when all other values are altered, (end of list processing error)

Example 5: Wrong array is accessed to retrieve or alter a value (invalid pointer)

Example 6: Process values beyond the storage range of an array or other data structure (algorithm processes two structures or alters values outside array dimensions)

Example 7: Statement alters data structures when it is not expected (side effects)

Example 8: In applying semantic overlays to identify pointers and links, identification of a variable pointing to a different item (variation in consistent pattern as in linked lists or other structures)

Example 9: Program violates stated semantic patterns for execution sequences

Example 10: Program reaches a semantic state not previously specified in requirement constraints relating to specific variables and conditions, thereby entering an unknown condition

Example 11: Conditions not set in proper order (similar to example 9, but concerning variable states)

Example 12: More statements executed than expected

Example 13: Changes in execution pattern

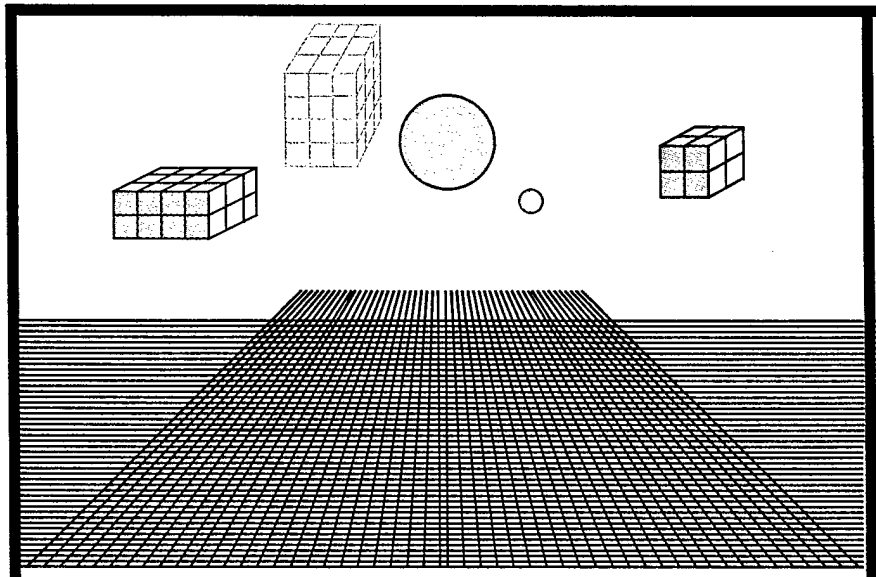


Figure 20 (a): Sample Program Execution, Data Structures.

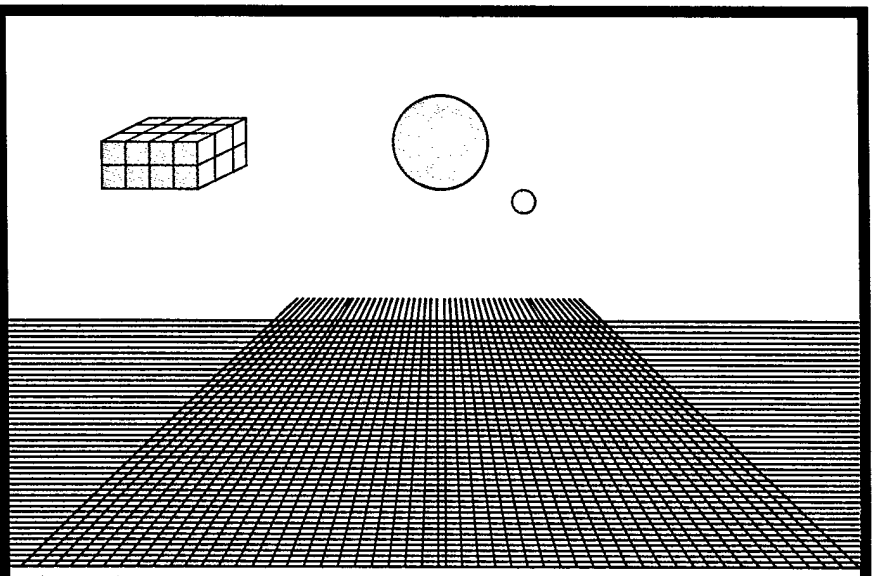


Figure 20 (b): Sample Program Execution, Simple Model.

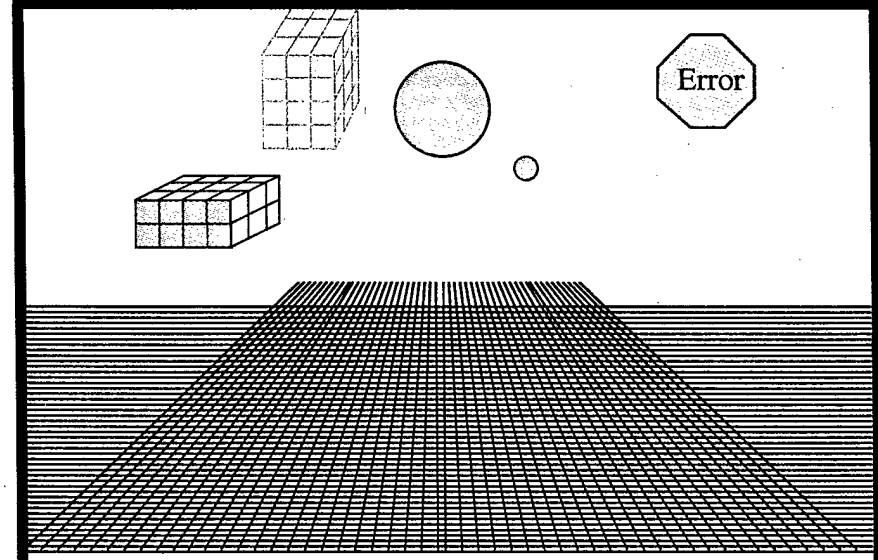


Figure 20 (c): Sample Error Detection, Visual Model 1.

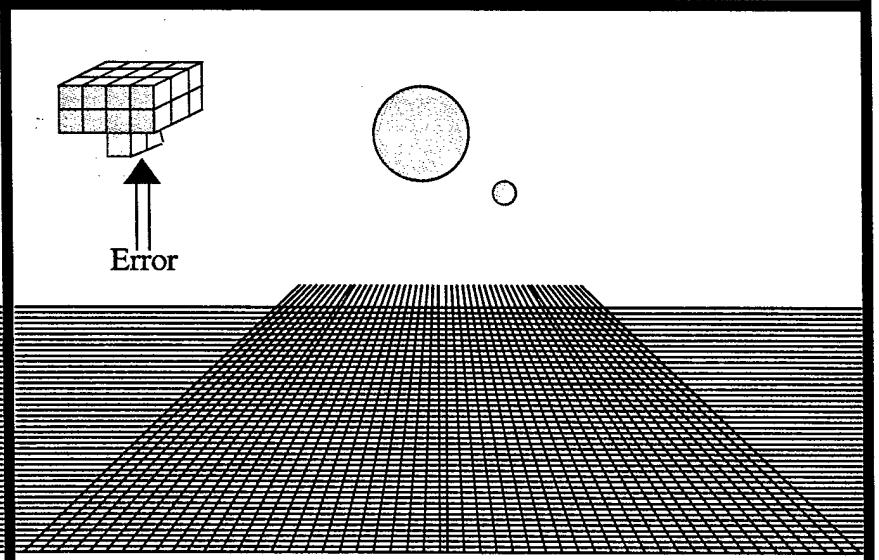


Figure 20 (d): Sample Error Detection, Visual Model 2.

Example 14: Execution of rarely executed code

Example 15: Formation of discrepancies in link patterns

Example 16: Unusual formations of data structures

Intentionally Left Blank

9. User Directives

Tool Location

The tool has been provide on a DAT tape created by a tar command with no compression. Two directories are on the tape within a directory named VisAttProject. The first directory is Graphics-Routines. It contains the code to generate the visual OpenGL models. The second directory is VisualPreprocessorFiles. It contains the lexical analyzer and the parser routines as well as required header files.

Required Software Environment

To run the system, one must have access to a C compiler, and an OpenGL compiler. The model can be run stand-alone. However, to incorporate the full functionality of the multi-dimensional capabilities, Eigen/VR should be used. The OpenGL model is the input to Eigen/VR.

User Requirements

The user must specify the name of the new executable program as well as providing the original code with the previously specified requirements of providing a single file with no include statements or compiler directives. In addition, the tool will not handle continuation lines.

Compiler Directives

Sample makefiles are provided with the source code. Basically, the user must run the preprocessor with the executable program as input, then invoke the visual routines by providing the generated output to the visual routines. The system cannot process compiler directives within the program to be visualized.

Intentionally Left Blank

10. Developer Directives

Functionality Extension

Appendix A contains a description of the language definition that was used to develop the pre-processor. The tool processes the complete language, although the prototype does not use all of the information at present. In addition, many of the language features are simply identified with no further action taken. This provides excellent functionality expansion. As new attributes or language features need to be visualized, the appropriate statements can be inserted at the specified locations.

Internal Structures

Appendix B contains a description of the data structures utilized within the parser. The lexical analyzer builds the appropriate data structure to generate the new executable, and the parser inserts appropriate statements depending on the analysis. In addition, a symbol table routine is generated to run the visual model.

Intentionally Left Blank

11. Conclusions

Advancements

The major advancement of this work is to develop multidimensional visual models of abstract and concrete program features that cooperate with a constraint monitor thereby allowing an approach to identifying completeness errors with the software specifications.

Disadvantages

The major disadvantage of the work is that select requirement constraints must be specified within a Requirements Constraint Language.

Significance

The significance of this work is that it provides a first step in evaluating specification completeness, and provides a more productive method for program comprehension and debugging.

Expected Payoff

The expected payoff is increased software surety confidence. In addition, increased program comprehension and reduced development and debugging time.

Future Work

Future work will focus on expanding the visual models, completing the constraint monitor, and expanding the work to the specification phase of the software life cycle model.

Intentionally Left Blank

References

1. Albuquerque Journal, Sunday, November 12, 1995.

2. Ball, T., and S.G. Eick, "Software Visualization in the Large," *Computer*, April 1996, pp. 33-43.

3. Berztiss, A.T., "Safety-Critical Software: A Research Agenda," *International Journal of Software Engineering and Knowledge Engineering*, Vol. 4 No. 2, 1994, pp. 165-181.

4. Braham, R., "Math & Visualization: New Tools, New Frontiers," *IEEE Spectrum*, November 1995, pp. 19-37.

5. Huff, C. C., M. Klein, and S. Stevens, "The State of the Art in Scientific Visualization," *Technical Report*, CMU/SEI-95-SR-Visualization, Software Engineering Institute Carnegie Mellon University, September 1995.

6. Codognet, P. and D. Diaz, "Compiling Constraints in clp(fd)," *Journal of Logic Programming* 27, 3, 1996.

7. Collins, E., L. Dalton, D. Peercy, G. Pollock, and C. Sicking, "A Review of Research and Methods for Producing High-Consequence Software," *1995 IEEE Aerospace Applications Conference*, Vol 1, January 1995, pp. 197-245.

8. "Cyberware," *Time*, August 21, 1995.

9. Embley, D., B. Kurtz, and S. Woodfield, Object-Oriented Systems Analysis (A Model-Driven Approach), Yourdon Press, 1992.

10. European Computer Research Center, *Eclipse User's Guide*,1993.

11. Gibbs, W., "Software's Chronic Crisis," *Scientific American*, September 1994.

12. Hermenegildo, M. and the CLIP Group, "Some Methodological Issues in the Design of CIA--A Generic, Parallel Concurrent Constraint System," In *Principles and Practice of Constraint Programming*, LNCS 874, May, Springer-Verlag, New York, 123-133, 1994.

13. Kimelman, D., B. Rosenburg, and T. Roth, "Strata-Various: Multi-Layer Visualization of Dynamics in Software System Behavior," IBM Thomas J. Watson Research Center, June 1994.

14. Lagedec, P., "Major Technological Risk", Quoted in *Safeware, System Safety and Computers*, Nancy Leveson, University of Washington, Addison-Wesley, 1995.

15. Musa, J. D., A. Iannino, and K. Okumoto, Software Reliability: Measurement, Prediction, Application, McGraw-Hill, Inc., 1987

16. Pollock, G. M., and L. J. Dalton, "A Strategic Surety Roadmap for High Consequence Software," *1996 Aerospace Applications Conference*, Snowmass CO, Vol. 4, February 1996, pp. 351-370.

17. Price, B.A., Baecker, and I. A. Small, "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages and Computing* 4(3):211-266.

18. Reiss, S. P., "An Engine for the 3D Visualization of Program Information," Dept. of Computer Science, Brown University, May 1995.

19. Smolka, G., "The Oz Programming Model," In *Computer Science Today*, Jan van Leeuwen, Ed., LNCS, No. 1000, Springer-Verlag, Berlin, 324-343, 1995.

20. Van Hentenryck, P., V. A. Saraswat, and Y. Deville, "Constraint Processing in cc(fd)," In *Constraint Programming: Basics and Trends*, A. Podelski, Ed., LNCS 910, Springer-Verlag, 1995.

21. Walden, K., and J. Nerson, Seamless Object-Oriented Software Architecture, Prentice Hall, 1995.

22. Yau, S. S., D. Bai, and K. Yeom, "An Approach to Object-Oriented Requirements Verification in Software Development for Distributed Computing Systems," *Proceedings of the Eighteenth Annual International Computer Software & Applications Conference*, 1994, pp. 96-102.

23. Zeus, DEC Systems Research Center, http://www.research.digital.com/SRC/zeus, all Zeus images copyrighted 1997 DIGITAL Equipment Corporation, All rights reserved. Provided couresy DIGITAL Systems Research Center, Palo Alto, California.

Appendixes

A. Language Grammar:

This appendix contains the language definition utilized by the parser for creation of the preprocessor. Future extensions will need to be incorporated within this structure.

```
%}

%token  AMPEQ           ARROW           AUTOTK
%token  BAREQ           BREAKTK
%token  CAROTEQ         CASETK          CHARCONSTTK   CHARTK
%token  CONSTTK         CONTINUETK
%token  DBAMP           DBBAR           DBEQ          DBGRTEQ
%token  DBGRTR          DBLESS          DBLESSEQ      DBMINUS
%token  DBPLUS          DEFAULTTK       DOTK          DOTSTK
%token  DOUBLETK
%token  ELSETK          ENUMCONSTTK     ENUMTK        EXCLAEQ
%token  EXTERNTK
%token  FIX_INDEX_EXPR  FLOATTK         FLTCONSTTK     FORTK
%token  GOTOTK          GRTEQ
%token  IDENTIFIERTK    IFTK            INTTK
%token  LESSEQ          LONGTK          LOWER_THAN_ELSE
%token  MINUSEQ         MYINTCONTK
%token  PERCTEQ         PLUSEQ
%token  QUESTION
%token  REGISTERTK      RETURNTK
%token  SHORTTK         SIGNEDTK        SIZEOFTK      SLASHEQ
%token  STAREQ          STATICTK        STRINGTK      STRINGTKIO
%token  STRUCTTK        SWITCHTK
%token  TYPEDEFNAMETK   TYPEDEFTK
%token  UNIONTK         UNSIGNEDTK
%token  VOIDTK          VOLATILETK
%token  WHILETK


%left  ","
%right "=" PLUSEQ MINUSEQ STAREQ SLASHEQ PERCTEQ AMPEQ CAROTEQ
BAREQ DBLESSEQ DBGRTEQ
%right SIZEOFTK
%right QUESTION   ":"
%left  DBBAR
```

```
%left  DBAMP
%left  "|"
%left  "^"
%left  "&"
%left  DBEQ  EXCLAEQ
%left  "<" LESSEQ ">" GRTEQ
%left  DBLESS DBGRTR
%left  "+" "-"
%left  "*" "/" "%"
%right "!" "~" DBPLUS DBMINUS
%nonassoc FIX_INDEX_EXPR
%left  "(" ")" "[" "]" ARROW "."
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSETK


%%

translation_unit:
     external_declaration
   | translation_unit external_declaration
   ;


external_declaration:
     function_definition
   | declaration
   ;


function_definition:
     declarator compound_statement
   | declarator declaration_list compound_statement
   | declaration_specifiers declarator compound_statement
   | declaration_specifiers declarator declaration_list
        compound_statement
   ;


declaration:
     declaration_specifiers ";"
   | declaration_specifiers init_declarator_list ";"
   ;
```

```
declaration_list:
        declaration
      | declaration_list declaration
      ;


declaration_specifiers:
        storage_class_specifier
      | storage_class_specifier declaration_specifiers
      | type_specifier
      | type_specifier declaration_specifiers
      | type_qualifier
      | type_qualifier declaration_specifiers
      ;


storage_class_specifier:
        AUTOTK
      | REGISTERTK
      | STATICTK
      | EXTERNTK
      | TYPEDEFTK
      ;


type_specifier:
        VOIDTK
      | CHARTK
      | SHORTTK
      | INTTK
      | LONGTK
      | FLOATTK
      | DOUBLETK
      | SIGNEDTK
      | UNSIGNEDTK
      | struct_or_union_specifier
      | enum_specifier
      | typedef_name
      ;


type_qualifier:
        CONSTTK
      | VOLATILETK
      ;
```

```
struct_or_union_specifier:
     struct_or_union identifier "{" struct_declaration_list "}"
   | struct_or_union "{" struct_declaration_list "}"
   | struct_or_union identifier
   ;


struct_or_union:
     STRUCTTK
   | UNIONTK
   ;


struct_declaration_list:
     struct_declaration
   | struct_declaration_list struct_declaration
   ;


init_declarator_list:
     init_declarator
   | init_declarator_list "," init_declarator
   ;


init_declarator:
     declarator
   | declarator "=" initializer
   ;


struct_declaration:
     specifier_qualifier_list struct_declarator_list ";"
   ;


specifier_qualifier_list:
     type_specifier
   | type_specifier specifier_qualifier_list
   | type_qualifier
   | type_qualifier specifier_qualifier_list
   ;
```

```
struct_declarator_list:
        struct_declarator
    |   struct_declarator_list "," struct_declarator
    ;


struct_declarator:
        declarator
    |   declarator ":" constant_expression
    |   ":" constant_expression
    ;


enum_specifier:
        ENUMTK identifier "{" enumerator_list "}"
    |   ENUMTK "{" enumerator_list "}"
    |   ENUMTK identifier
    ;


enumerator_list:
        enumerator
    |   enumerator_list "," enumerator
    ;


enumerator:
        identifier
    |   identifier "=" constant_expression
    ;


declarator:
        direct_declarator
    |   pointer direct_declarator
    ;

direct_declarator:
        identifier
    |   "(" declarator ")"
    |   direct_declarator "["  "]"
    |   direct_declarator "[" constant_expression "]"
    |   direct_declarator "("  ")"
    |   direct_declarator "(" parameter_type_list ")"
    |   direct_declarator "(" identifier_list ")"
    ;
```

```
pointer:
      "*"
    | "*" type_qualifier_list
    | "*" type_qualifier_list pointer
    | "*" pointer
    ;


type_qualifier_list:
      type_qualifier
    | type_qualifier_list type_qualifier
    ;


parameter_type_list:
      parameter_list
    | parameter_list "," DOTSTK
    ;


parameter_list:
      parameter_declaration
    | parameter_list "," parameter_declaration
    ;


parameter_declaration:
      declaration_specifiers declarator
    | declaration_specifiers
    | declaration_specifiers abstract_declarator
    ;


identifier_list:
      identifier
    | identifier_list "," identifier
    ;


initializer:
      assignment_expression
    | "{" initializer_list "}"
    | "{" initializer_list "," "}"
    ;
```

```
initializer_list:
        initializer
      | initializer_list "," initializer
      ;


type_name:
        specifier_qualifier_list
      | specifier_qualifier_list abstract_declarator
      ;


abstract_declarator:
        pointer
      | pointer direct_abstract_declarator
      | direct_abstract_declarator
      ;


direct_abstract_declarator:
        "(" abstract_declarator ")"
      | direct_abstract_declarator "[" constant_expression "]"
      | direct_abstract_declarator "["  "]"
      | "[" constant_expression "]"
      | direct_abstract_declarator "(" parameter_type_list ")"
      | direct_abstract_declarator "("  ")"
      | "(" parameter_type_list ")"
      ;


typedef_name:
        TYPEDEFNAMETK
      ;


statement:
        labeled_statement
      | expression_statement
      | compound_statement
      | selection_statement
      | iteration_statement
      | jump_statement
      ;
```

```
labeled_statement:
      identifier ":" statement
    | CASETK constant_expression ":" statement
    | DEFAULTTK ":" statement
    ;


expression_statement:
      expression ";"
    | ";"
    ;


compound_statement:
      "{" declaration_list statement_list "}"
    | "{" statement_list "}"
    | "{" declaration_list  "}"
    | "{" "}"
    ;


statement_list:
      statement
    | statement_list statement
    ;


selection_statement:
      IFTK "(" expression ")" statement    %prec LOWER_THAN_ELSE
    | IFTK "(" expression ")" statement ELSETK statement
    | SWITCHTK "(" expression ")" statement
    ;

iteration_statement:
      WHILETK "(" expression ")" statement
    | DOTK statement WHILETK "(" expression ")" ";"
    | FORTK "(" ";" ";" ")" statement
    | FORTK "(" expression ";" ";" ")" statement
    | FORTK "(" ";" expression ";" ")" statement
    | FORTK "(" ";" ";" expression ")" statement
    | FORTK "(" ";" expression ";" expression ")" statement
    | FORTK "(" expression ";" expression ";" ")" statement
    | FORTK "(" expression ";" ";" expression ")" statement
    | FORTK "(" expression ";" expression ";" expression ")"
        statement
    ;
```

```
jump_statement:
      GOTOTK identifier ";"
    | CONTINUETK ";"
    | BREAKTK ";"
    | RETURNTK expression ";"
    | RETURNTK ";"
    ;


expression:
      assignment_expression
    | expression "," assignment_expression
    ;


assignment_expression:
      conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;


assignment_operator:
       "="
    | STAREQ
    | SLASHEQ
    | PERCTEQ
    | PLUSEQ
    | MINUSEQ
    | DBLESSEQ
    | DBGRTEQ
    | AMPEQ
    | CAROTEQ
    | BAREQ
    ;


conditional_expression:
      logical_OR_expression
    | logical_OR_expression QUESTION expression ":"
         conditional_expression
    ;


constant_expression:
      conditional_expression
    ;
```

```
logical_OR_expression:
     logical_AND_expression
   | logical_OR_expression DBBAR logical_AND_expression
   ;


logical_AND_expression:
     inclusive_OR_expression
   | logical_AND_expression DBAMP inclusive_OR_expression
   ;


inclusive_OR_expression:
     exclusive_OR_expression
   | inclusive_OR_expression "|" exclusive_OR_expression
   ;


exclusive_OR_expression:
     AND_expression
   | exclusive_OR_expression "^" AND_expression
   ;


AND_expression:
     equality_expression
   | AND_expression "&" equality_expression
   ;


equality_expression:
     relational_expression
   | equality_expression DBEQ relational_expression
   | equality_expression EXCLAEQ relational_expression
   ;


relational_expression:
     shift_expression
   | relational_expression "<"  shift_expression
   | relational_expression ">"  shift_expression
   | relational_expression LESSEQ shift_expression
   | relational_expression GRTEQ shift_expression
   ;
```

```
shift_expression:
      additive_expression
    | shift_expression DBLESS additive_expression
    | shift_expression DBGRTR additive_expression
    ;


additive_expression:
      multiplicative_expression
    | additive_expression "+" multiplicative_expression
    | additive_expression "-" multiplicative_expression
    ;


multiplicative_expression:
      cast_expression
    | multiplicative_expression "*" cast_expression
    | multiplicative_expression "/" cast_expression
    | multiplicative_expression "%" cast_expression
    ;


cast_expression:
      unary_expression
    | "(" type_name ")" cast_expression
    ;


unary_expression:
      postfix_expression
    | DBPLUS unary_expression
    | DBMINUS unary_expression
    | unary_operator cast_expression
    | SIZEOFTK unary_expression
    | SIZEOFTK "(" type_name ")"
    ;


unary_operator:
      "&"
    | "*"
    | "+"
    | "-"
    | "~"
    | "!"
    ;
```

```
postfix_expression:
      primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression "(" argument_expression_list ")"
    | postfix_expression "(" ")"
    | postfix_expression "." identifier
    | postfix_expression ARROW identifier
    | postfix_expression DBPLUS
    | postfix_expression DBMINUS
    ;


primary_expression:
      identifier
    | myconstant
    | STRINGTK
    | STRINGTKIO
    | "(" expression ")"
    ;


argument_expression_list:
      assignment_expression
    | argument_expression_list "," assignment_expression
    ;


identifier:
      IDENTIFIERTK
    ;


myconstant:
      intconst
    | charconst
    | fltconst
    | enumconst
    ;


enumconst:
      ENUMCONSTTK
    ;
```

```
fltconst:
     FLTCONSTTK
  ;


charconst:
     CHARCONSTTK
  ;


intconst:
     MYINTCONTK
  ;
```

Intentionally Left Blank

B. Data Structures:

This appendix defines the major data variables and structures used within the preprocessor (the lexical analyzer and parser) for the common symbol table. Updates to the code may incorporate additional changes. Therefore, the actual code represents the final definitions.

```
/*  Define the data type codes ....     */

   These codes are used to provide
semantic meaning when assigning and
testing data vars for their data type.


    #define char_type          1

    #define double_type        2

    #define float_type         3

    #define int_type           4

    #define long_type          5

    #define short_type         6

    #define signed_type        7

    #define struct_union_spec  8

    #define type_typedefname   9

    #define unsigned_type     10

    #define void_type         11
```

/* Define the symbol table size limitations */

These codes are used to facilitate increasing processing sizes when the tool is ready to be scaled up. The TEST_ vars are used to make sure that the appropriate arrays are initialized properly if their allocation sizes change. They are initialized statically, and thus, additional code must be added if the size increases.

```
#define NUM_OF_ARRAY_SIZES          1000
#define TEST_ARRAY_SIZES            1000

#define NUM_OF_CHAR_VALUES          1000
#define TEST_CHAR_VALUES            1000

#define NUM_OF_DOUBLE_VALUES        1000
#define TEST_DOUBLE_VALUES          1000

#define NUM_OF_FLOAT_VALUES         1000
#define TEST_FLOAT_VALUES           1000

#define NUM_OF_FUNC_VAR_NAMES       1000
#define TEST_FUNC_VAR_NAMES         1000

#define NUM_OF_INT_VALUES           1000
#define TEST_INT_VALUES             1000

#define NUM_OF_VAR_NAME_VALUES      1000
#define TEST_VAR_NAME_VALUES        1000

#define NUM_OF_VAR_PNTRS_VALUES     1000
#define TEST_VAR_PNTRS_VALUES       1000

#define NUM_OF_LONG_INT_VALUES      1000
#define TEST_LONG_INT_VALUES        1000

#define NUM_OF_SHORT_INT_VALUES     1000
#define TEST_SHORT_INT_VALUES       1000

#define NUM_OF_SIGNED_CHAR_VALUES   1000
#define TEST_SIGNED_CHAR_VALUES     1000

#define NUM_OF_SIGNED_INT_VALUES    1000
#define TEST_SIGNED_INT_VALUES      1000
```

```
#define NUM_OF_SUBSCRIPT_VALUES      1000
#define TEST_SUBSCRIPT_VAL_SIZE      1000

#define NUM_OF_UNSIGNED_CHAR_VALUES 1000
#define TEST_UNSIGNED_CHAR_VALUES    1000

#define NUM_OF_UNSIGNED_INT_VALUES   1000
#define TEST_UNSIGNED_INT_VALUES     1000

#define SIZE_DS_SYM_TABLE           10000

#define SIZE_OF_DYNAMIC_INFO        10000

#define SIZE_OF_EXEC_MODULES        10000

#define SIZE_OF_FUNC_TABLE             50
```

```
/* Define the Variable Names Data Structures  */

      This structures contains the basic information about each
    of the program variables identified within the program.

int    Sym_table_Next_Empty = 0;
struct Vis_Sym_Data_Structures{
  char   *Vis_Sym_DS_Name;
              Points to a string of the var name;

  int     Vis_Sym_DS_Data_Type;
              Equal to the code for the data type;

  int     Vis_Sym_DS_Num_Dimen;
              States num of dimen, if 0, not an array
              and related DS var will be 0;

  int     Vis_Sym_DS_Size_Lnk;
              Points to a specific subscript position
              within the Array_Sizes array. This is
              the beginning of a short link that
              consists of VIS_Sym_DS_Num_Dimen elements.
              Each entry contains the size of one of
              the dimensions of this array var. They are
              listed in the order of the original definition.

  int     Vis_Sym_DS_Value_Lnk;
              This is a similar link to a position in a
              data array that contains the value(s) for
              this variable.  The Vis_Sym_DS_Data_Type var
              identifies which array contains the data. For
              array data, it is mapped linearly in row major
              order.

  int     Vis_Sym_DS_Scope_Lvl;
              This is a simple variable denoting the scope
              level and range of the variable.

  int     Vis_Sym_DS_Line_Defined;
              This identifies the line on which the variable
              was defined

  int     Vis_Sym_DS_Data_Structures;
              This is used to link entries within the structure
              as needed.  Exact format currently undecided.
} Vis_Sym_Data_Structures[SIZE_DS_SYM_TABLE];
```

```
/*  Define the Data Type Value Storage Arrays  */
    Each of these arrays is used to store data
    values of the associated types.


  int             Array_Sizes_Next_Empty = 1;
  int             Array_Sizes[NUM_OF_ARRAY_SIZES]

  int             Char_Values_Next_Empty = 1;
  char            *Char_Values[NUM_OF_CHAR_VALUES]

  int             Int_Values_Next_Empty = 1;
  int             Int_Values[NUM_OF_INT_VALUES]

  int             Short_Int_Values_Next_Empty = 1;
  short int       Short_Int_Values[NUM_OF_SHORT_INT_VALUES]

  int             Long_Int_Values_Next_Empty = 1;
  long int        Long_Int_Values[NUM_OF_LONG_INT_VALUES]

  int             Float_Values_Next_Empty = 1;
  float           Float_Values[NUM_OF_FLOAT_VALUES]

  int             Double_Values_Next_Empty = 1;
  double          Double_Values[NUM_OF_DOUBLE_VALUES]

int             Unsigned_Char_Values_Next_Empty = 1;
unsigned char *Unsigned_Char_Values[NUM_OF_UNSIGNED_CHAR_VALUES]

int             Unsigned_Int_Values_Next_Empty = 1;
unsigned int    Unsigned_Int_Values[NUM_OF_UNSIGNED_INT_VALUES]
```

```
    /* Define the Function Names Data Structures  */

int    Sym_Functions_Next_Empty = 0;
struct Vis_Sym_Functions{

  char    *Func_Name;
            Identifies the name of a defined function.

  int      Func_Num_Params;
            Identifies the number of parameters.

  int      Func_Var_Pntrs;
            This is a pointer into the Func_Var_Names
            array. It indicates the start of a short
            consecutive string of integer pointers into
            the Vis_Sym_Data_Structures array. Each
            pointer points to the information for that
            parameter value. They are listed in the order
            they appear on the function declaration.

  int      Func_Num_Loc;
            This is the number of lines of code within
            the function to calculate the size of the
            function "ball" in the visualization.

  int      Func_Sym_Pntr;
            Pointer used to provide order within this
            array structure. Not yet defined.

} Vis_Sym_Functions[SIZE_OF_FUNC_TABLE];

  int     Func_Var_Names_next_empty = 1;
  int     Func_Var_Names[NUM_OF_FUNC_VAR_NAMES]  = {
            This contains short subscript pointers
            into the Vis_Sym_Data_Structures array
    /* Define the Dynamic Execution Information   */

int    Dynamic_Info_Next_Empty = 1;
struct Sym_Dynamic_Info{

  int     Line_Number;
            This identifies the line number to which the
            associated dynamic info is related.

  int     Line_Scope;
```

This identifies the scoping level of the associated line.

int **Line_Num_Var_References**;
Contains the number of variables referenced on this line.

int **Line_Var_Ref_Pntrs**;
Pointer into Line_Var_Name for beginning of a short list containing Line_Num_Var_References entries. Each entry is a pointer into the Vis_Sym_Data Structures array for the definition of this variable. This gives typing info and num of expected subscripts.

int **Line_Var_Subscript_Pntrs**;
Pointer into Line_var_Pntr for beginning of a short list containing Line_Num_Var_References entries. Each entry is a pointer into the Subscript_Values array for each of the variable references. If a variable does not have any subscript references, then it has a value of zero.

int **Line_Info_Pntr**;
Pointer to add structure to this array if needed. Not yet defined.

] **Sym_Dynamic_Info**[SIZE_OF_DYNAMIC_INFO];

```
/* Define Supporting Structs for Dynamic Line Info */

int     Subscript_Values_next_empty = 1;
int     Subscript_Values [SIZE_OF_SUBSCRIPT_VALUES] = {
          Lists of subscript values. Line_Var_Pntr
          indicates the start of each "list" for each
          variable reference.

int     Line_Var_Pntr_next_empty = 1;
int     Line_Var_Pntr[SIZE_OF_LINE_VAR_PNTR] = {
          List of pointers, one for each var referenced
          on a line, that points into the start of a list
          in the Subscript_values array, giving the subscript
          reference values at the time of reference on the
          line.

int     Line_Var_Name_next_empty = 1;
int     Line_Var_Name[SIZE_OF_LINE_VAR_NAME] = {
          List of pointers, one entry for each var referenced
          on a line.  The pointer points to the appropriate
          name in the Vis_Sym_Data Structures array for
          additional info on the var.  The start of the list
          is in Line_Var_Ref_Pntrs for each line.
```

```
/*  Define required flags      */

int   Array_Flag_Cntr = 0;

        Used to identify when array is being processed, and
        to easily handle nesting of array references within
        subscripts.

        It is initialized when a "[" is found after an
        identifier;

        It is also compared to the "Array_Sub_Nest_Flag_Cntr"
        and the Sub_Expr_Flag_Cntr to know when a multi-
        dimensional array is being accessed within the array
        subscript of another array.

int   Type_Specifier_Flag = 0;
        Used to indicate when variable definitions are being
        made so appropriate data is entered into the symbol
        table.

        Var references are handled differently from var
        definitions.

        It is set when the type specifier is found, and
        decremented when the end of statement is found.

int   Array_Sub_Nest_Flag_Cntr = 0;

        Useful/Needed if array references are allowed as
        subscripts, then within the nesting of the array
        references, need to have some way of connecting the
        current specification with the correct previous
        reference.  So when "]" is encountered, it can be
        matched to the proper "[".

int   Sub_Expr_Flag_Cntr = 0;

        Flag/Cntr to indicate whether the current subscript
        evaluation is an expression.  This helps with nesting
        levels, and references to asrrays within the subscript
        definitions.  When matching "]" is found, the counter
        is decremented.

int    Current_Scope = 0;
```

```
  /*  Define a Structure to track Execution Modules */

struct Exec_Modules{

  int  Exec_Beginning = 0;
  int  Exec_End       = 0;

} Exec_Modules[SIZE_OF_EXEC_MODULES];


/************  End of Symbol Table Definitions ********/
```

C. Visualization State of the Art Survey:

This appendix contains a report contracted with the Software Engineering Institute at Carnegie Melon University as a precursor to this work. It is added here as an addendum to our literature review, rather than repeating the information in section 4.

The State of the Art in Scientific Visualization

Clifford C. Huff
Mark Klein
Scott Stevens

Technical Report
CMU/SEI-95-SR-Visualization
ESC-SR-95-Visualization
September 1995

Software Engineering institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

CONTENTS

The State of the Art in Scientific Visualization

Abstract: Scientific Visualization is a rapidly developing technology which has not been significantly exploited or pushed by the needs of the software development community. The state of the practice for scientific visualization is quite advanced for many domains outside of software development. in the realm of visualization of software and visualization as a tool for software developers, the current state of this visualization domain appears to be far behind. Principally this is due to visualizations in these other scientific disciplines being based much more on recognizable physical attributes than the artificial immature abstract attributes found in software development. We have found there is a dearth of information and experience in visualization of software system attributes. It is quite clear from this survey of the state of the art of visualization, that the visualization of software quality attributes is not the focus of any current research. Visualization of program execution and potentially visual programming are the only areas of on-going research that is applicable to HIS. As a part of this work, we have identified a large number of organizations, professional activities, tools, publications and samples on the subject of visualization. This information should aid in characterizing the current state of scientific visualization and to act as a seed repository of information on this subject.

1 Introduction

This work is aimed at providing a snapshot of the state of scientific visualization to help focus potential research and experimentation in software visualization.

To this end we have identified a large number of organizations, professional activities, tools, publications and samples on the subject of visualization. The results of this survey work can be found in the following appendixes:

Annotated Visualization Bibliography
Summary of Organizations involved with Graphic & Visualization
Scientific Visualization Sampler
Visualization & Graphical Tools
Visualization Related Conferences

Readers of this report are highly encouraged to scan this material. This material is being made available in paper and electronic form to the sponsors of this report. Where possible, we have attempted to provide a World Wide Web Uniform Resource Locators (URL's) to provide readers with easy access to additional information on a particular citation, conference, organization, tool or visualization sample. An overview of the appendix material is presented at the end of this report.

2 What is Scientific Visualization?

In general terms, Scientific Visualization can be thought of as any method which presents scientific information in a manner to facilitate the conceptualizations of scientific phenomenon or statistical information [Hughes]. The term Scientific Visualization was formalized in practice as the result of a National Science Foundation panel which published in 1987 the "Visualization in Scientific Computing" report. The original goal of this panel was to provide a focus to unify the disciplines of computer graphics, image processing, computer vision, computer-aided design, signal processing and the study of human computer interfaces [Rosenblum].

Through scientific visualization, researchers across a range of scientific disciplines have taken advantage of visualization technology to display and clarify vast quantities of otherwise incomprehensible data. Since the data is presented in a pictorial form, researchers are able to use the brain's ability to make analogies and links between the visual image and existing ideas--links that are not likely to be made when data appears as columns of numbers or lines of text. A good scientific visualization[1] system allows the researcher to make discoveries not otherwise possible and provides him with a powerful new interface to his data [Price].

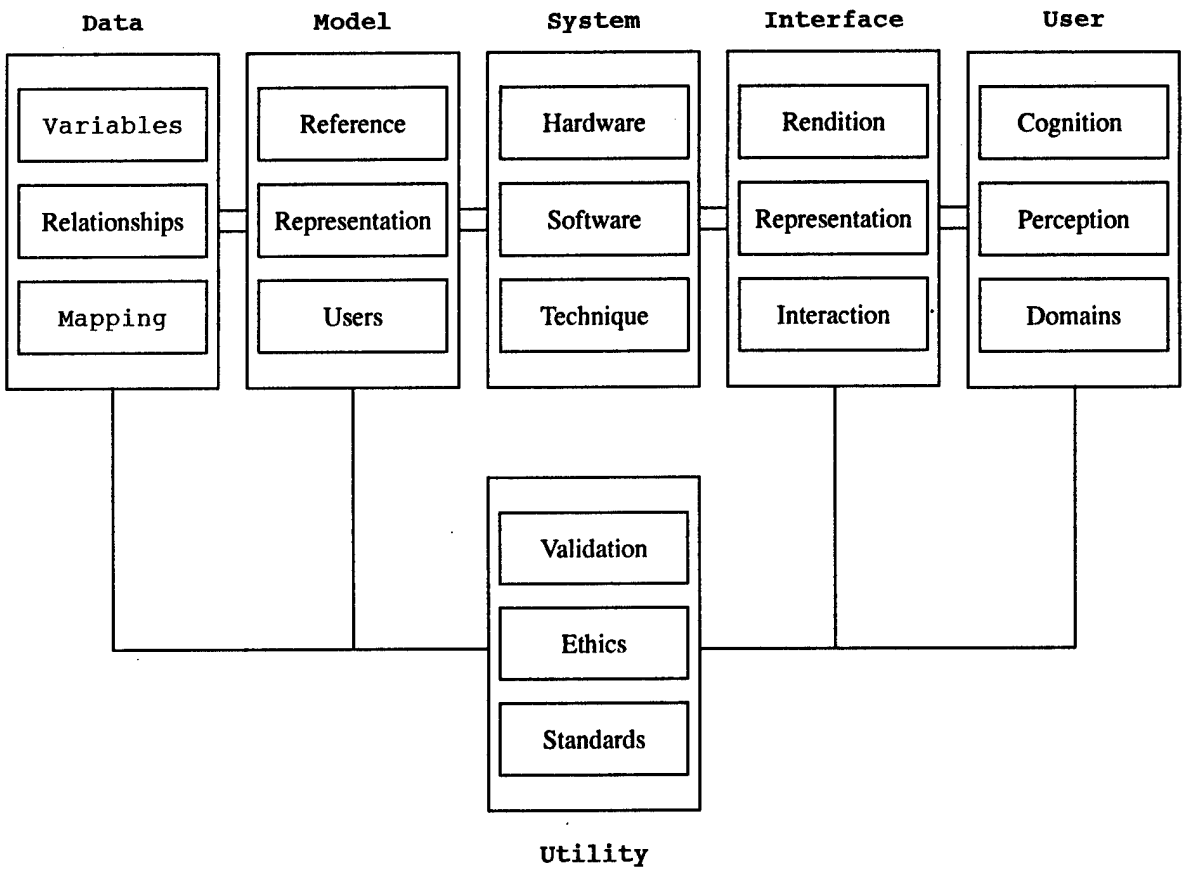
Put simply, scientific visualization is the use computerized imagery to gain insight into complex phenomena or information [Hughes].

1. Many excellent examples of visualization can be found Appendix C: Scientific Visualization Sampler.

3 The State of Scientific Visualization

There are many components to what comprises visualization as a general field and should be considered in assessing the state of visualizations today. One such model is outlined below [Williams]. For the purposes of this report, we will use this model as a roadmap to guide the discussion on the state of Scientific Visualization. We will briefly describe each component[1] of the model and describe the state of the art for that aspect of visualization. Then we will relate what we believe the state is relative to what might be required for advanced software visualization and visual manipulation as envisioned by the HIS program.

Visualization Model



1. For a more detailed discussion on each element in this model, we recommend reading the original paper from which this model is taken. This paper also provides an excellent general overview of scientific visualization history and research.

3.1 Data Component

The data component is concerned with the raw material of visualization. Data may be generated in a variety of ways and from a number of sources. Data may be collected from nature, generated from laboratory experiments, produced by simulation or abstracted from objects and processes by humans or machines. Data typically represents selected variables, relationships and values of the target objects and processes of a visualization. The variables, relationships and values used in a visualization may be the result of a well established theory or may represent a proposed theory or model which the visualization is meant to help verify. In other cases visualization is intended to support the discovery of new structures, relationships, hypotheses, models or theories. Those responsible for the construction of a visualization and those responsible for the use of a visualization must understand the data used as well as its visual representation. Since a visualization is a mapping of domain data onto an array of visual clues where it is then rendered and displayed, the management, preparation, structure and mapping of the data are critical parts of a visualization solution [Williams].

The mapping of data to visualization parameters is the central focus where cognitive science, domain knowledge, computational science and computer graphics intersect. The dataflow paradigm[1] is currently the primary high level technique used to implement the mapping. This approach allows the user to concentrate on the visualization mapping, but forces trade-offs between power and flexibility due to a fixed set of mappings which may not satisfy all visualization applications [Williams].

One area of the data component receiving significant research attention is the area of management of large data sets. There are researchers talking about managing terabyte size data sets coming from new generations of Earth resource satellites. There are also discussions about access to data sets via relational database technologies. There are also discussions of using object oriented database technologies for managing complex data sets [Rosenblum].

---

1. The dataflow paradigm is based upon connecting a set of data processing modules. Each module performs a specific action on the data. The connections between these modules represent the dataflow between them.

In general, we believe data gathering and management is not viewed as a significant roadblock to the production of most scientific visualizations. In the realm of software visualization, the mapping of data to a visual representation is an important area for research especially how this relates to user issues of cognition and perception.

3.2 Model Component

The model component is concerned with the abstraction which describes central elements of a domain and their behaviors, interactions and interfaces. A number of different standard models have been proposed like a reference model, data model, user model, time model and device model.

A reference model can standardize terminology, identify core elements, identify constraints and limitations and help to compare systems. Data models have been proposed, but none incorporate a generic data description which includes all data types as well as the semantics of the data for a wide range of visualization applications. Data types[1] include geometric, kinematics, dynamic, physical characteristics, etc. It is important for developers and user to operate at a high level of abstraction and yet preserve the integrity of the data structures. Models of users can be based on their application domains, the types of visualization tools needed, the methods of operating their visualization tools and their level of computing expertise. A time model is needed that can formally describe a time variable and its relationship with all processes in which it is involved. Device models are needed to describe the types of data that devices can accept and the functionality of devices in processing the data. A device model would incorporate video audio, head mounted displays, 3-D positioning and orientation and multidata inputs and outputs [Williams].

One very important model not considered by the previous discussion is a model for advanced visual programming based on attribute visualization and component construction of the kind envisioned by the HIS program. The seeds of this model may come from the current and future generations of visual programming languages.

---

1. This does not refer to programming sense of data types like integer, real, pointer, etc.

There are many visual programming researchers who concede that the goal to make programming and program understanding simpler by representing programming constructs, elements and concepts visually is a far more complex and difficult task than anyone would like [Freeman].

Current visual programming models program at a relatively fine grain level of detail roughly equivalent to one or a small number of lines of code in a conventional textual programming language. This creates a basic problem that must be addressed and solved: when a large collection of constructs used in textual programming is translated into a large collection of visual constructs, textual complexity is merely replaced with visual complexity. Compounding this problem is the additional complexity that occurs when there is no clear relationship between visual symbols and the concepts they represent [Freeman]. The management of complexity is a key problem which needs to be addressed by coupling research in enhanced semantics for visual languages along with human cognition and perception to graphical, auditory and haptic (feel) presentation techniques.

Also a burden on many visual languages is their visualization in only 2 dimensions (2-D). Many of these visual languages have been inspired by the pictures programmers draw when they are sketching outlines of their programs, flowcharts, or data dependencies. Current research in visual languages is now focused on a 3-D presentation. By extending the visual space to 3-D, researchers believe they can reduce some of the complexity that results from the limitations of 2-D space. Additionally, researchers believe they can also take advantage of the extra dimension in representing concepts and program structure [Freeman].

We believe that without models in hand, like the ones described above, progress in visualization systems including software visualization systems will be difficult. Our research indicates that slow but steady progress is being made in the general modeling areas indicated above. However, there is significant work required to improve and create visual programming models which are essential to software visualization of the kind proposed by HIS.

3.3 System Component

The system component is concerned with the hardware platforms, software and graphical techniques used for visualization.

Hardware

In the hardware arena, we are most interested in computing engine performance and input and output devices for human interaction. The current pace of hardware computing performance improvement is holding around 18 months for microprocessor performance to double. At this rate, one computer in the 25 years will be as powerful as all the computer in Silicon valley today. [Patterson] Current high end RISC-based microprocessors such as DEC's Alpha, IBM's PowerPC and Sun's Sparc have SPECmark performance ratings from 100 to 300. By the year 2000, processors will easily have a SPECmark performance of 1000 [Weiss].

A fair amount of commercialization and research has been done to create a wide array of input and feedback devices for human computer interaction (HCI). These HCI devices include technologies for emmersive virtual reality environments which include head mounted displays, stereoscoptic systems, holography displays, audio feedback, haptic displays, data gloves, hand and eye tracking devices [Williams]. We are now beginning to see the impact of this technology even in cost sensitive areas like the consumer-oriented entertainment market. A good example of this is Nintendo's new Virtual Game Boy which incorporates a monochromatic (red) 3-D head mounted display.

Software for 3-D Presentation

As seen in the attached Scientific Visualization Sampler (Appendix C), 3-D representations and rendering techniques are widely used. Due to the foundational significance of 3-D representation and rendering as a key system component, we believe it is important to note the current state of industry based on 3-D standards.

Currently, there are two relatively new non-proprietary industry developed standards garnering significant attention. These are OpenGL from Silicon Graphics and QuickDraw 3-D from Apple. Both

OpenGL and QuickDraw 3-D are intended to be environments for developing cross platform 3-D graphics applications. Silicon Graphics OpenGL is an application programming interfaces (API's) intended to be vendor neutral and a cross platform industry standard. OpenGL has already found a home with Microsoft's Windows NT and the recently released Windows 95.

Apple's QuickDraw 3-D is also a cross-platform application program interface (API) for creating and rendering real-time, workstation-class 3-D graphics. It consists of human interface guidelines and toolkit for a consistent user interface, a high-level modeling tool kit, a shading and rendering architecture, a cross-platform metafile format for storing 3-D objects 93DMF) and a device and acceleration manager for plug and play hardware acceleration. QuickDraw 3-D is available now for Power Macintosh systems, the Windows version will ship later this year. Apple has released the 3-D metafile specification for Macintosh, Windows and UNIX platforms.

A significant derivative of OpenGL is VRML (Virtual Reality Markup Language). VRML is an open, platform-independent, file format for 3-D graphics on the Internet's World Wide Web. Similar in concept to the Web standard for text, Hypertext Markup Language (HTML), VRML encodes computer-generated graphics into a compact format for transportation over a network. As with HTML, a user with an appropriate VRML-compatible viewer can view the contents of an interactive 3-D graphics file as well as navigate to other VRML "worlds" or HTML pages. A number of research organizations now have VRML-based content to visualize and explore chemistry related information such as biomolecules.

Beyond cross platform foundational 3-D standards like OpenGL and QuickDraw 3-D, we note that Microsoft has begun a multipronged strategy of making their presence known in both the consumer 3-D graphics and high-end 3-D markets. In the consumer market, Microsoft is reported to have plans to create a "new standard for fast, cheap 3-D software and hardware along with easy-to-use visualization-compatible packages incorporating sound, images and animation [BusinessWeek]. It appears for high end CAD modeling, Microsoft will continue to rely on the OpenGL 3-D standard. but for the consumer oriented 3-D market, Microsoft is developing two different application programming interfaces (API). Their low level 3-D API is reported to be called Direct 3-D. Direct 3-D is intended as a low-level API for software developers especially in the game market who require fast 3-D performance. Their high level API is reportedly called

RealityLab 3-D which is intended for software developers writing consumer, business, and virtual reality applications [Byte].

Techniques

Beyond 3-D standards, our research noted a great deal of interest in advanced research in improved volume visualization techniques. This is particular true for those working on medical imaging and visualization. Research topics in this area center around improved volume graphic techniques and improved real-time rendering, and enriching volumes with knowledge about that volume - such as automated segmentation of the volume (e.g., automated identification of different tissue types or organs found in medical volume visualizations) [Rosenblum]. It is unclear to use whether this line of research will be any direct interest in software visualization.
Overall in terms of software and graphical techniques used for visualization, there is already a rich set of techniques to work with and can be readily used for software visualization research in the near term. Long term it is quite likely that new graphics techniques may evolve to meet the specialized needs of effective software visualizations.

While there are some very complex and time consuming visualizations being done today, we believe the system component should not be viewed as a significant roadblock to the production of most scientific visualizations. In the next few years, it is unlikely that it will be necessary to push any aspect of the system component technology beyond what is already ongoing or will naturally occur to achieve progress in software visualization. So in this area, we anticipate the need only to monitor progressed in this system component area and plan for the incorporation of new and relevant technologies as they become available.

3.4 Interface Component

The interface component is concerned with the Human-Computer Interface (HCI). HCI provides the presentation and interaction capabilities, which in the best case, is matched to human cognitive and perceptual capabilities. A well designed and implemented HCI facilitates the creation, enhancement and navigation necessary to support the user's need for exploring a

visualization solution [Williams].

Here there are a number of HCI active research issues of note. There is the need for improved user interfaces that permit steering through data sets. There is a need for improved virtual reality (VR) interfaces to provide an emmersive 3-D environment that takes full advantage of visual, auditory and tactile senses for visualization and manipulation. As VR matures, interfaces that support the immersion of the user in a 3-D space with the presentation and navigation tools necessary for controlling and manipulating the environment presents significant challenges in terms of performance, rendering and data management. VR visualization applications in the domains of fluid flows, quantum mechanics and astronomical events have already shown promise for the scientific and engineering communities [Rosenblum][Williams].

There are other VR applications which employ the sense of feel. These sorts of interfaces include force-feedback systems to help "feel" the strength of a bond between atoms or molecules, or feel the molecular surface of a tooth or permit simulated or remote surgical operations [Hughes].

In general, we believe the HCI interface component to be a rapidly maturing technology base; which left alone on its current path has and will yield suitable HCI for the purposes of software visualization.

3.5 User Component

The user component is concerned with human elements of perception and cognition as well as essential domain knowledge. Visualization attempts to take advantage of the fact that vision is the most highly developed human sense for the reception, recognition and understanding of information in our environment. Visual perception and cognition are leveraged as the main tool in the analysis of pictorial data. Designing an effective visualization requires knowledge of human visual perception and the cognitive processing of visual information. Likewise, an effective visualization requires domain knowledge which must be incorporated into the visualization application [Williams].

As can be seen by the visualization sampler (Appendix C), the software domain does not have a rich set of visual metaphors for

representing software domain knowledge. Extensive research into developing appropriate visual metaphors for software is required. This includes research into visual metaphors for visualizing quality attributes as envisioned by the HIS initiative. These visual metaphors need to rely on perception and cognitive knowledge and to embody software domain knowledge. We need to identify what mix of visual elements like size, shape, color, texture, movement, animation, dimensional presentation (e.g., 3-D) can be used as an effective basis for visual metaphors for software. Additionally, we need to determine what software domain knowledge and what granularities of this knowledge are best represented in these visual metaphors for software.

We believe there is a sufficient base of existing perception and cognition knowledge to begin an investigation to identify a set or sets of visual metaphors appropriate for representing software for the purposes of understanding and constructive manipulation.

For those pursuing issues of perception and cognition and visual metaphors, we highly recommend they examine the works of E.R. Tufte [Tufte83][Tufte 90] and Peter & Mary Keller [Keller]. Tufte is often cited for providing valuable insights and guidelines for effective as well as ineffective methods of presenting data based on the interaction of human perception and the display presented to the user [Williams]. The Kellers have a very insightful book on "Visual Cues - Practical data Visualization".

We find there is a need for greater automated support for visualization to reduce the reliance upon multidisciplinary teams of people to create high quality visualizations. Today it can take a small staff of people like a graphic programmer, graphic design artist, researcher and cognitive physiologist. To reduce this reliance on human availability of expertise, it may be useful to make visualization programs incorporate some form of AI-based design assistant substituting for members of a visualization team. With this type of assistance, it will greatly improve the ability for a single researcher or engineer to create high quality visualizations - enabling an age of unassisted high quality desktop visualization.

There is also a need for standardizing aspects of visualization so that communications among researchers, users and developers is less ambiguous and more precise. For example, the current

state-of-the-art visualization terminology is ambiguous, conflicting and imprecise. Likewise, file formats have not been standardized to permit easy interchange between different visualization applications. There are however a number of formats competing to become a standard [Williams].

In the are of standardizing software visualization terminology, we found Blaine A. Price, et.al. work on "A Taxonomy of Software Visualization" to be exceptionally noteworthy. This work proposes a novel and systematic taxonomy of six areas making up thirty characteristic features of software visualization technology. Their taxonomy is presented and illustrated in terms of its application to seven systems of historic importance and technical interest [Price].

3.6 Utility Component

The utility component is concerned with the applicability, accuracy and relative utility of using visualization solutions. Like all tools and techniques, visualization is not value neutral. Visualization solutions require validation and verification of their results. Since decisions made in constructing visualization solutions are not value free, there are ethical issues involved. The old saw about telling lies with statistics also applies to visualization [Williams].

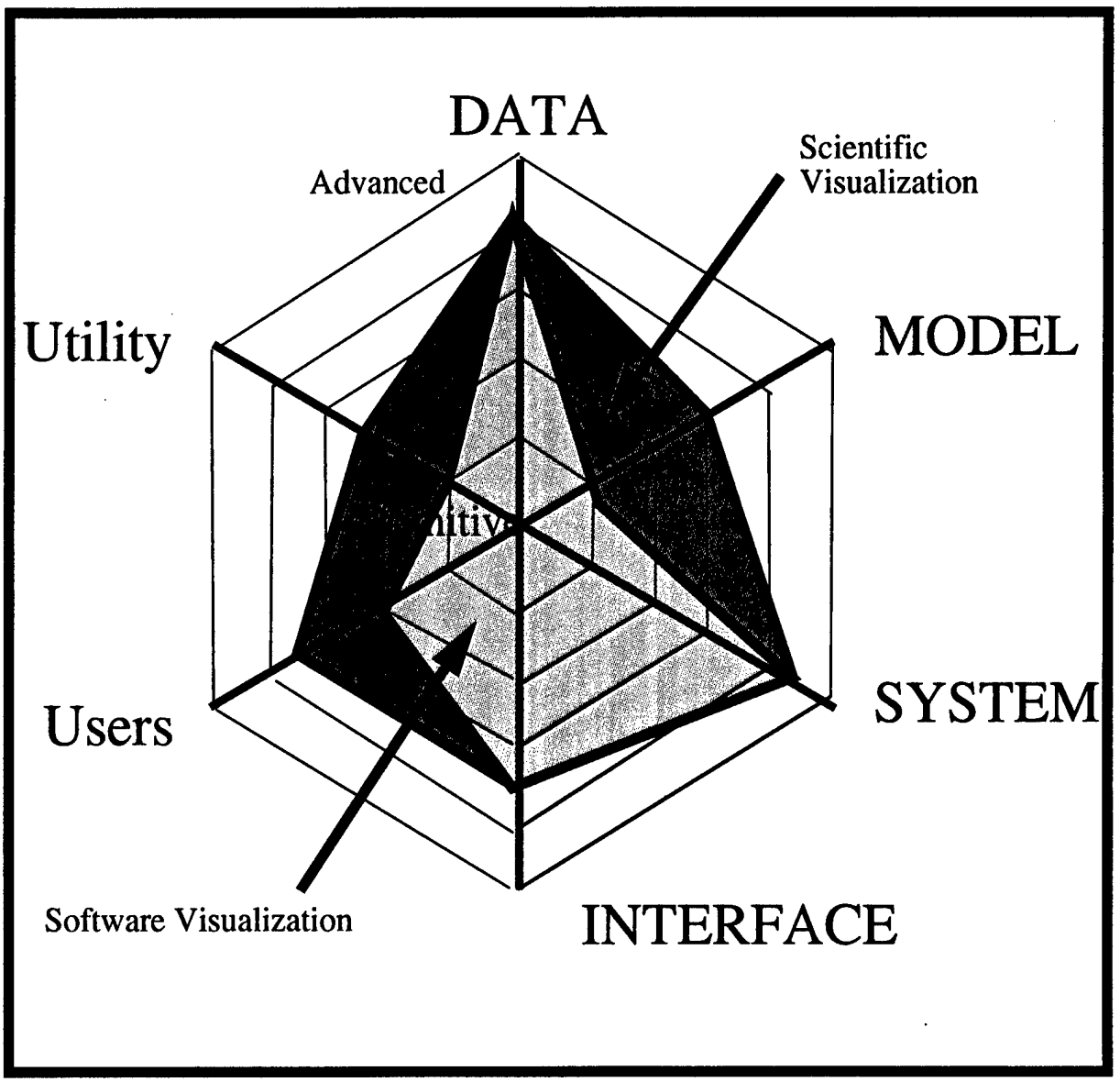
In general, we believe validation and verification to be important for any visualization endeavor and it is highly important to the HIS goal. Our research found one reference that would tend to indicate that the state-of-the-art in this area is not well developed and requires ongoing attention for the foreseeable future[Uselton].

4. State of the Art Summary

In general, we find that the General State of the Art of Scientific Visualization is relatively advanced in all areas of the roadmap visualization Model. With the areas of Users, Utility and Modeling lagging behind the other areas. For the State of the Art of Software Visualization, we find that the User, Utility and MOdeling area are even more primitive and require significant attention.

The chart below is a Kiviat chart which depicts our assessment of the General State of the Art of Scientific Visualization versus the State of the Art of Software Visualization. This assessment is based upon the previous narrative and is purely subjective.

Each of the six areas of our Visualization model are represented on the six axis of the Kiviat chart. Each axis has a scale from primitive to advanced. The mapping of advancement in each area onto the chart creates a "footprint" by which to access the general state-of-the-art of visualization versus the specific state-of-the-art of software visualization.



5 Conclusion

Scientific Visualization is a rapidly developing technology which has not been significantly exploited or pursued by the needs of the software development community. The state of the practice for scientific visualization is quite advanced for many domains outside of software development. In the realm of visualization of software and visualization as a tool for software developers, the current state of this visualization domain appears to be far behind. Principally this is due to visualizations in other scientific disciplines being based much more on recognizable physical attributes than on the artificial immature abstract attributes as found in software development.

We have found there is a dearth of information or experience in visualization of soft3ware system attributes. It is quite clear from this survey of the state of the art of visualization, that the visualization of software quality attributes is not the focus of any current research. Visualization of program execution and potentially visual programming are the only areas of on-going research that are applicable to HIS.
Substantial research is required to improve the science of visualization for software development. We need to identify what we want to see, how we want to see it and how we want to interact with it. We need better models of software in general and visual programming. Visualization of quality attributes as envisioned in the HIS initiative will require research and experimentation to identify the right set of visual metaphors to represent attributes and their interaction. A practical and rich visual environment for software development is years into the future.

Nevertheless, we believed a rich software visualization environment is an important technology necessary to achieve HIS goals. One of these goals being a modeling-simulation environment in which developers manipulate representations of a system (e.g., architecture descriptions, specifications, requirements), to carry out various analysis and "correct by construction" synthesis tasks.

We have one final closing note of caution to balance out this survey on the state-of-the-art of visualization and its relation to visual programming. There are some very respected individuals who believe that efforts in the direction purely visual programming are doomed to fail. One individual is Fred Brooks

who in 19878 remarked:

> *"A favorite subject for Ph.D. dissertations in software engineering is graphical, or visual programming - the application of computer graphics to software design... Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will." [Brooks]*

More recently in 1993, L. O'Brien has written:

> *"...Beware the claims of visual programming. Drawing lines between objects becomes bafflingly web-like. Purely visual programming is not yet and may never be viable." [OBrien]*

Hopefully, there will be found techniques and technologies which overcome these objections and speculations.

6 Appendix Summary

This summary highlights the contents of 5 appendixes to this report. Included in this section is notable information about the State-of-the-Art in Scientific visualization that did not fit into the Visualization Model directly, but have an important bering to this subject.

The material in Appendixes A-D are organized into different categories due to the large number of items in these appendixes (e.g.,all the organizations associated with a US university or all the visualization samples for program visualization). There are category overview pages included before the material in the appendix to aid in understanding the appendix organization and for later location of individual items.

6.1 Appendix A: Annotated Visualization Bibliography

The visualization bibliography contains 174 citations arranged in 24 different categories. Of the 174 citations, 70% of them

have been annotated with summary abstracts. In most cases, the abstracts are taken directly from source material as written by the original authors. Of the 174 citations, all are available in paper format unless the citation is for a book. 30% of the citations are available in electronic format - either Portable Document Format or Postscript. Carnegie Mellon University's Library Information System (LIS), the Library of Congress Marvel access system, and search engines-libraries on the World Wide Web (e.g., CMU's Lycos[1], AOL's Webcrawler[2], Colorado University's Harvest Computer Science bibliography[3], and InfoSeek[4]) were particularly rich sources of material gathered in the bibliography.

6.2 Appendix B: Graphic & Visualization Organizations

The list of organizations[5] which have visualization interests consists of 121 different organizations organized into 10 categories. Many of these organizations exist as a visualization center or laboratory. The sheer number of groups dedicated to visualization is a good indication of the wide spread interest in scientific visualization in general.

Among all the research organizations identified, we were most impressed with the Graphics, Visualization, and Usability Center at Georgia Institute of Technology. This organization has done extensive work in scientific visualization coupled with related fields such as animation, virtual environments, medical informatics, software visualization, user interface software, multimedia, educational technology and human factors.

We were also impressed by the work done at Brown University in the are of 3-D user interfaces for desktop and emmersive environments, and interactive 3-D toolkits for visual programming to provide insight into software programs, their structure and their execution. It is notable that Brown University is part of the five-university Science and Technology Center for Computer Graphics and Scientific Visualization

1. http://lycos.cs.cmu.edu
2. http://webcrawler.com
3. http://harvest.cs.colorado.edu
4. http://www.infoseek.com
5. NASA's Numerical Aerodynamic Simulation annotated scientific visualization web sites bibliography was a key source of material gathered in this bibliography [NASA].

consortium dedicated towards improving the fundamental and intellectual basis for computer graphics. The Center was founded in 1991 with support from the National Science Foundation and the Advanced Research Projects Agency. The other universities in the consortium are California Institute of Technology, Cornell University, University of North Carolina at Chapel Hill and University of Utah.

Also of note is Sandia National Laboratories Synthetic Environment Laboratory's Multidimensional, User-oriented Synthetic Environment (MUSE) project organization located at Albuquerque, New Mexico. This organization has done work in a wide range of different visualization categories. The focus of the MUSE project is to develop an open, multi-purpose software interface between general classes of scientific information and a highly interactive, multi-dimensional visualization system - including the incorporation of emmersive systems often referred to as virtual reality systems.

Finally, a couple of notes and observations about Microsoft. As noted earlier, Microsoft is putting their stake in the ground in the 3-D applications market. This is having an effect on the traditional 3-D market leader Silicon Graphics. For example, Microsoft has acquired SoftImage for porting of SoftImage to 32-bit Windows-based platforms. Previously, SoftImage was a high-end 3-D graphics package which ran only on Silicon Graphics platforms. In an effort to stem migration of market leading graphics packages like SoftImage of Silicon Graphics platforms to less expensive platforms, Silicon Graphics has acquired Alias Research and Wavefront Technologies. Products from both of these newly acquired companies have established themselves as market leading 3-D graphics products operating exclusively on Silicon Graphics platforms.

Our last observation about Microsoft comes from Richard F Riesenfeld, University of Utah computer scientist, who notes that Microsoft has quietly assembled "the largest collection of graphics talent under one roof in the world." Among this talent is Alvy Ray Smith, co-founder of Pixar Corporation and James Kajiya from the California Institute of Technology [BusinessWeek].

6.3 Appendix C: Scientific Visualization Sampler

The visualization sampler consists of 186 different examples of scientific visualization organized into 37 categories. These examples span from the physical to the abstract. From simple 2-D graph visualization to highly detailed interactive photorealistic animated visualizations. Included in this sampler were visualizations of atoms molecules, human organs, aircraft,

7 References

[Brooks] F.P. Books, Jr., *No Silver Bullet*, IEEE Computer, April 1987, pages 10-19.

[Byte] *Coming: A Better Multimedia Platform*, Byte, Oct 1995, p.27.

[BusinessWeek] *3-D Computing*, Business Week, September 4, 1995, pages 70-77.

[Csinger] Andrew Csinger, *The Psychology of Visualization*, Department of Computer Science University of British Columbia, November 1992.

[Freeman] Elisabeth Freeman, et.al., *In Search of a Simple Visual Vocabulary*, IEEE Symposium on Visual Languages, September 5-9, 1995.

[Hughes] Matt Hughes, *What is Visualization*, University of Minnesota, URL http://www.msi.umn.edu/SciVis/whatisviz.html

[Keller] Peter Keller, Mary Keller, *Visual Cues - Practical Data Visualization*, IEEE Computer Society Press, 1993.

[NASA] *Annotated Scientific Visualization Web Site Bibliography*, NASA's Numerical Aerodynamic Simulation Group, URL http://www.nas.nasa.gov/RNR/Visualization/annotated URLs.html

[OBrien] L. O'Brien, *Issues of Programming*, Computer Language, January 1993, pages 45-52.

[Patterson] David A Patterson, *Microprocessors in 2020*, Scientific American, September 1995.

[Price] Blaine A. Price, Ian S. Small, and Ronald M. Baecker, *A Taxonomy of Software Visualization*, Journal of Visual Languages and Computing 4(3), 1993.

[Rosenblum] L. Rosenblum, et.al., *Scientific Visualization: Advances and Challenges*, Academic Press Ltd.,

1994.

[Tufte 83] Tufte, E. R., *The Visual Display of Quantitative Information*, Graphics Press, 1983.

[Tufte 90] Tufte, E. R., *Envisioning Information*, Graphics Press, 1990.

[Uselton] Sam Uselton, chair, *Validation, Verification and Evaluation*, Proceedings of Visualization '94, IEEE Computer Society, 1994.

[Weiss] Ray Weiss, *SPARC Returns, Drives Health VMEbus Base*, Computer Design, August 1995.

[Williams] James Williams, *Visualization*, in *Annual Review of Information Science and Technology*, Vol 30, Learned Information Inc., 1995.

DISTRIBUTION:

2	MS 0535	L. J. Dalton, 2615
2	0535	L. C. Kidd, 2615
2	1109	A. L. Hale, 9224
2	1109	G. M. Pollock, 9224
1	9018	Central Technical Files, 8940-2
2	0899	Technical Library, 4916
2	0619	Review & Approval Desk, 12690 For DOE/OSTI
1	0161	Patent and Licensing Office, 11500

Report Number (14) SAND--98-0359

Publ. Date (11) 199803

Sponsor Code (18) DOE/HR, XF

UC Category (19) UC-900, DOE/ER

DOE