Dynamic Visualization Techniques for High Consequence Software

Guylaine M. Pollock
Sandia National Laboratories, MS 1109
Computer Sciences Department
P.O. Box 5800
Albuquerque, NM 87185-1109
505-845-7463
gmpollo@cs.sandia.gov

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. 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. In conclusion, the most significant advantage of this tool is to provide a first step in evaluating specification completeness, and to provide a more productive method for program comprehension and debugging. The expected payoff is increased software surety confidence, increased program comprehension, and reduced development and debugging time.

TABLE OF CONTENTS

1. INTRODUCTION

The development of software for use in high-consequence systems--systems where errors cause loss of life or significant financial or material loss--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. [1], [2] This paper describes a prototype tool developed for monitoring of high-consequence software under an initiative at Sandia National Laboratories (SNL) to identify how we will develop ultra-reliable software in the 2010 time frame.

The initiative is called the High Integrity Software Program (HIS), and 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 [3]. 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 which investigates methods and techniques to render today's systems safer, more secure and more reliable. This project is being developed under the correctness by design track. Why is this program important, what are the key issues, and what approach have we taken for this project?

Once a software system has been developed, the 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? This project attempts a first step in this type of verification.

Traditionally, there have been three areas of research for verification of system implementations: logical verification, mathematical verification, and statistical verification. These are all excellent approaches. However, Berztiss [4] has advocated that every possible technique and method should be utilized to address safety concerns, as current methods to

DISCLAIMER

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 current 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. 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.

Therefore, in addressing the issue of verification as well as other issues, it is time to consider an additional category, visualization. Accordingly, several such efforts are underway in various laboratories and universities [2], [5], [6], including this investigation of software attributes visualization within the High Integrity Software program at Sandia National Laboratories.

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 [7]. 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 [8], while a number of projects have focussed on algorithm animation, at least in two dimensional formats [9]. (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 [2]. 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 [10], 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 project 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 for virtual reality applications. 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. Before discussing the tool, a brief review of the background leading to this project is presented along with clarification of the project goals. Related projects are briefly presented in the literature review. An overview of the tool is presented before discussing *SAVAnT* and SAGE in more detail. Finally, a few examples of usage are given before conclusions are presented.

2. BACKGROUND

The High Integrity Software Program (HIS) at SNL 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.

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 [12].

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" [13].

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" [14]. 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.

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.

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

As previously stated, 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. 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. Finally, traditional work in the area of visualization has primarily focussed on the use of two-dimensional flow-chart like structures. However, this project investigates the use of visualization in a mutiple dimensionsional environment to improve surety confidence.

Therefore, 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. This meets the project goal of examining techniques for assessing the correct implementation of select requirement constraints. Further, the project assesses software during the execution phase of the life-cycle process.

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. In addition, a goal was to provide a tool that would be easy to use. Therefore a preprocessor was developed to generate the required version of the executable program. Finally, portability was important. So standard programming languages were utilized. Before describing the tool in greater detail, similar projects are reviewed.

4. Literature Review

Briefly reviewing related literature, we examine two aspects: what is the state of the art for verification of software requirement specifications, and how is visualization being used to aid program development and comprehension. In reviewing the state of the art in requirements verification approaches, Yau's work is typical [15]. 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 visualization, 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. While Huff has clearly documented that visualization has been used in a variety of areas [8]; from his report, it is clear that the use of visualization for the application of software development lags behind the use of visualization for scientific applications. However in reviewing the use of visual techniques for software development, a number of interesting systems have been developed. Zeus developed by DEC is one tool to examine. Figures 1 (a) - (d) illustrate a heap sort animated with Zeus [9]. This tool uti-

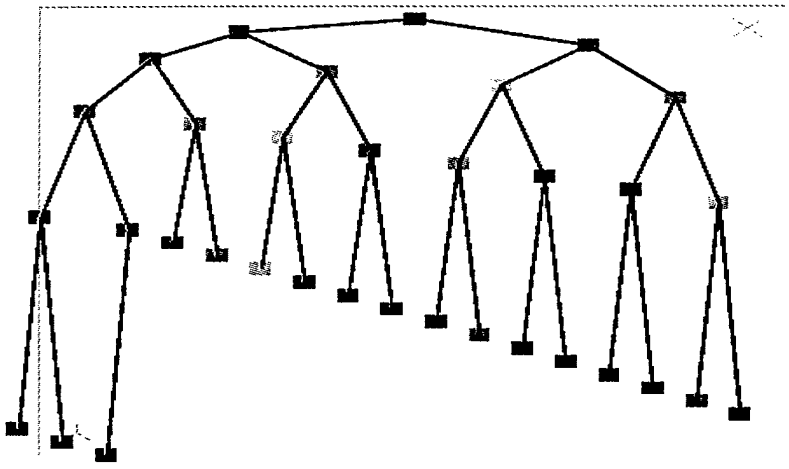

Figure 1 (a)**:** Zeus Sorting: View Down the Z Axis.

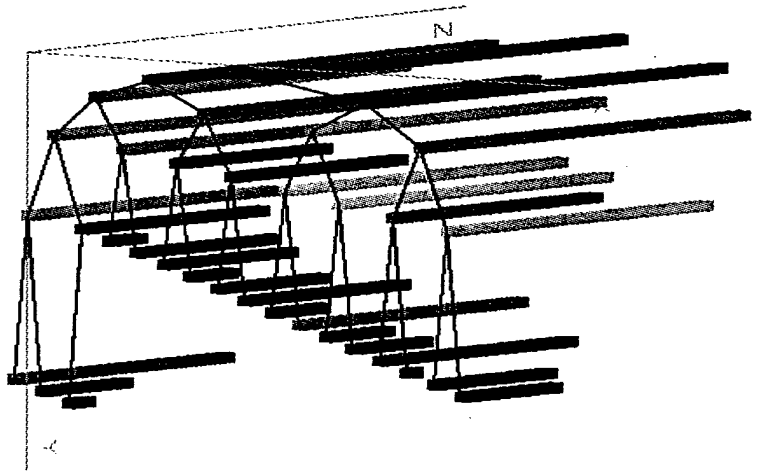

Figure 1(b)**:** Zeus Sorting: Rotated Around the Y Axis.

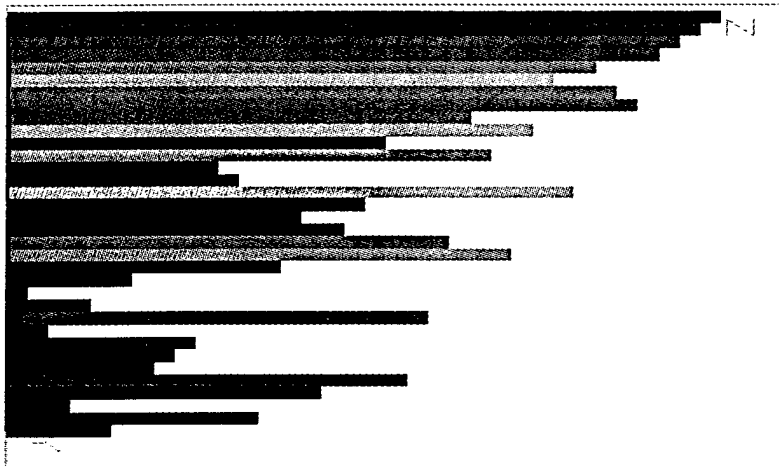

Figure 1(c)**:** Zeus Sorting: View Down the X Axis

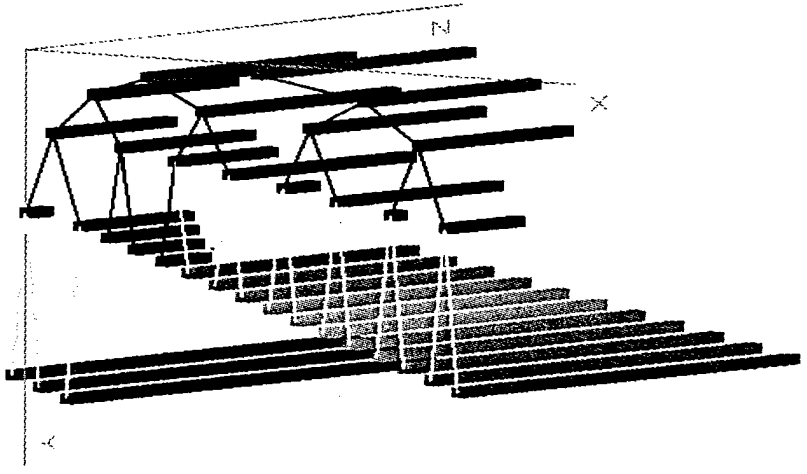

Figure 1(d)**:** Zeus Sorting: View After Partial Completion.

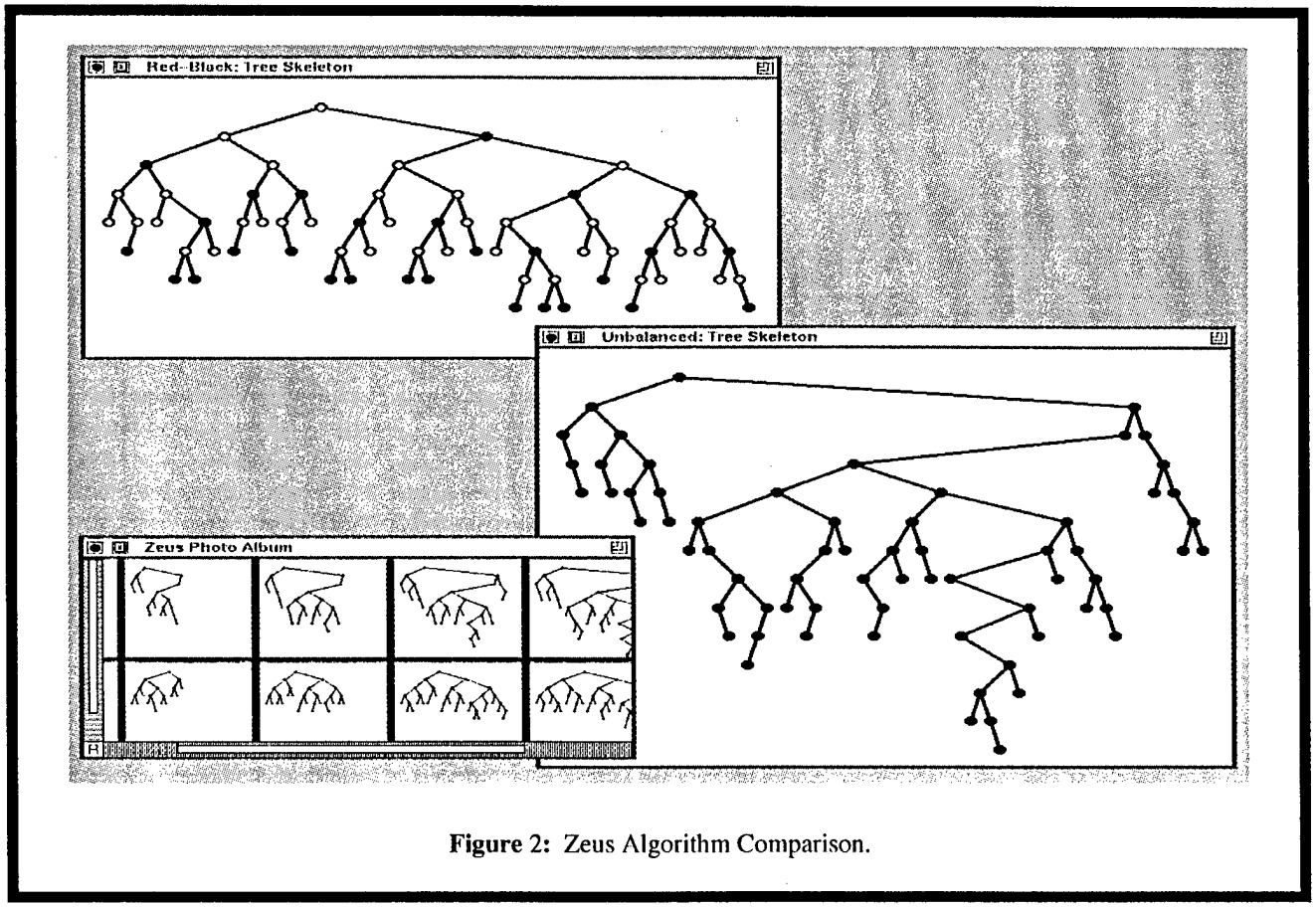

Figure 2: Zeus Algorithm Comparison.

lizes simple representations for algorithm animation. The colored bars represent the data values with the colors and the lengths of the bars indicating the relative value. Smaller numbers are represented in blue with increases in value represented by the increase in the color wheel.

Figures 2 illustrates another example of the Zeus prototype tool and how it can be used. This example illustrates the tree strductures generated by different algorithms for comparision.

Zeus was developed as a follow on from the Balsa system. Balsa was used to animate algorithms in Pascal programs for educational purposes. Balsa models are two dimensional, black and white models. Zeus, which supports multiple synchronized views of algorithms, has not been used outside a laboratory. Further, there have been no empirical evaluations performed on this tool.

Other tools for the reader to investigate are Tango, Anim, Genie, UWPI, SEE, TPM, Pavane, LogoMedia, and ObjectCenter. More detailed information on these tools and others can be found in [16].

5. Project/Tool Overview

What is *SAVAnT*, and how is it used? To start, figure 3 illustrates the semantic view of the system. A selected program is executed within the *SAVAnT* environment. The program must be 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 (RQL). 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. Muse is a multi-user synthetic environment used to emulate a virtual realitiy environment. 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 4 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

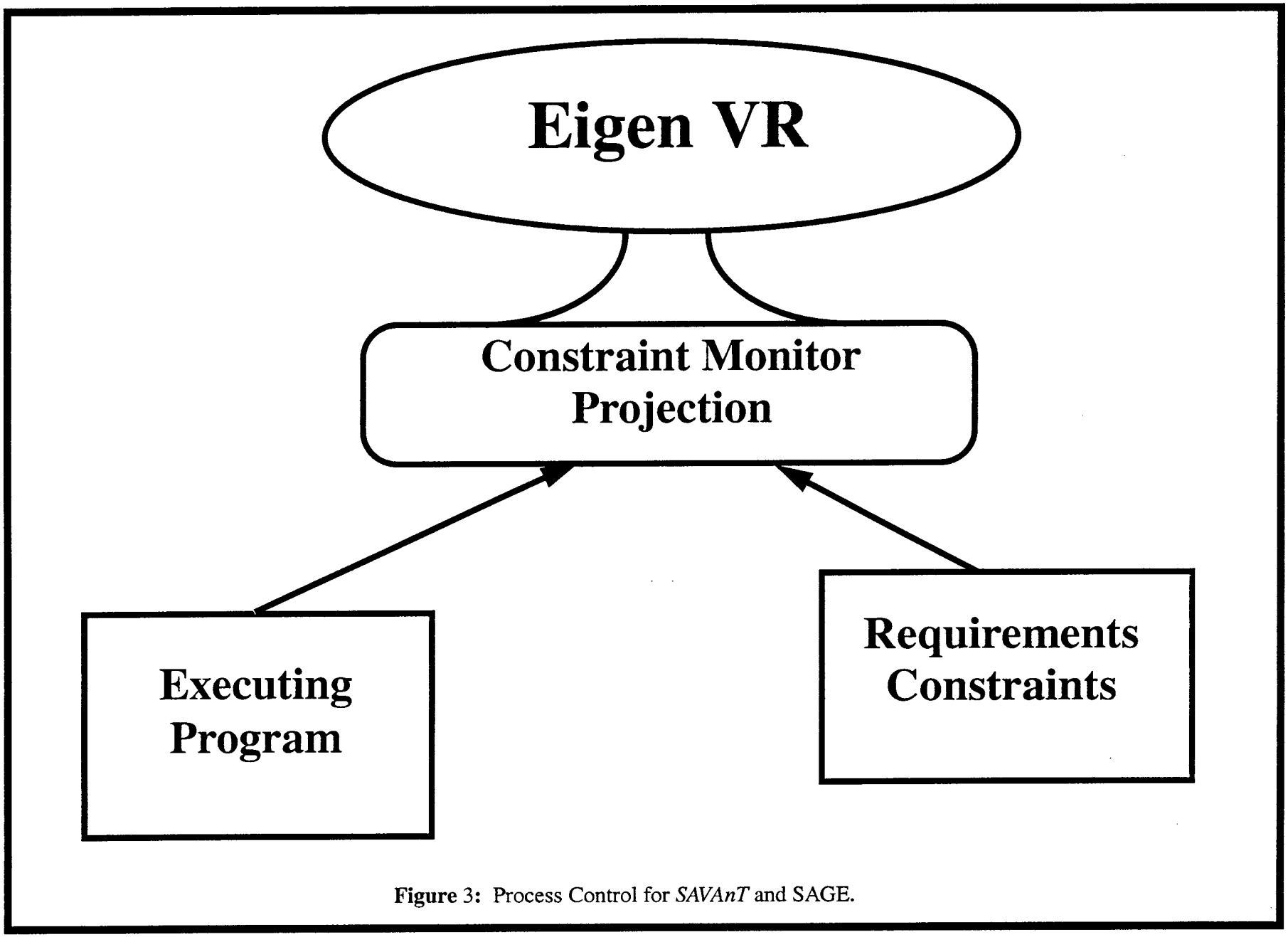

Figure 3: Process Control for *SAVAnT* and SAGE.

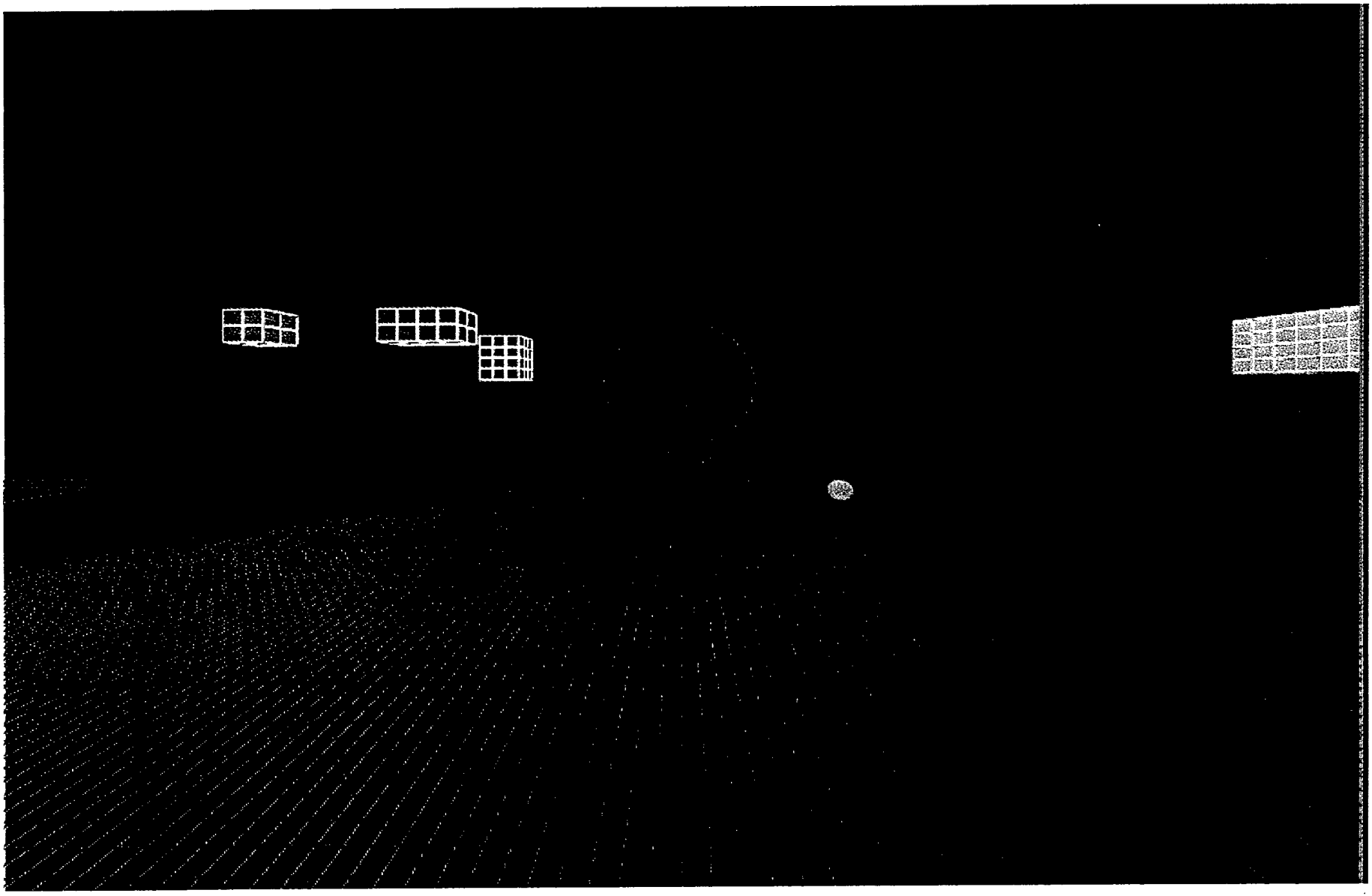

Figure 4: Sample Initial View of Generated Program Visual Model.

utomate 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. Later work will expand the available models.

Visual models 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 the data structures 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 5 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 6 shows a rotated view that the user sees while reorienting themselves through the "flight" capabilities, while Figure 7 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 8 shows the orientation when this feature is activated. The red axis is the *Z* axis, blue the *X* axis, and the *Y* axis is white. Figure 9 is a different view of the orientation. Let us now consider *SAVAnT* in more detail.

6. Software Attribute Visual Analysis Tool Description (*SAVAnT*)

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. The preprocessor, the executing program, the visualization routines, the constraint monitor and the controlling routines will be discussed.

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.

Executing Program

The executing program must be supplied by the user. It must be developed in C. As the tool is currently a prototype, the program selected 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

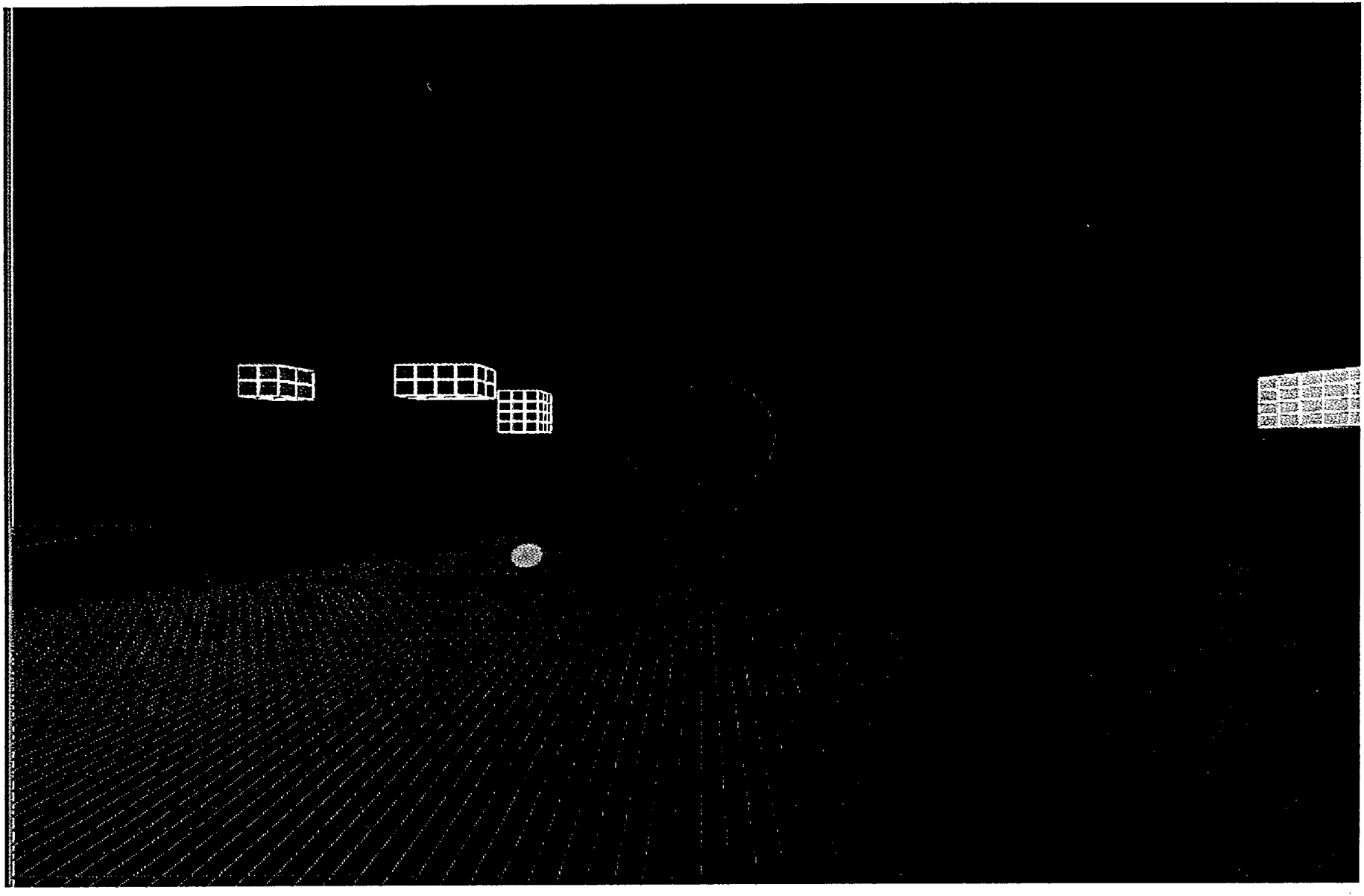

Figure 5: Subroutine Execution has Altered Position.

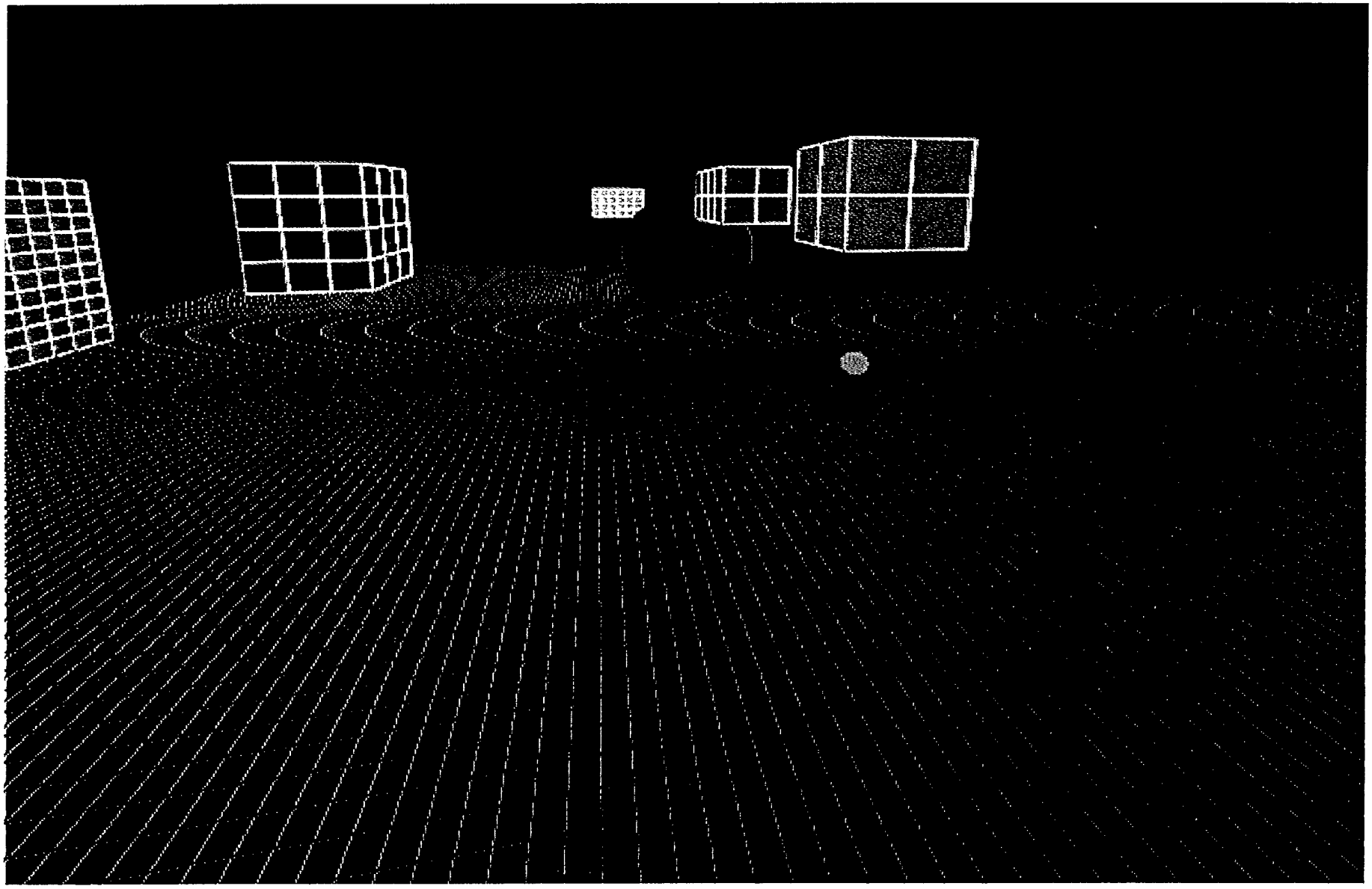

Figure 6: Original Model Rotated for Different View

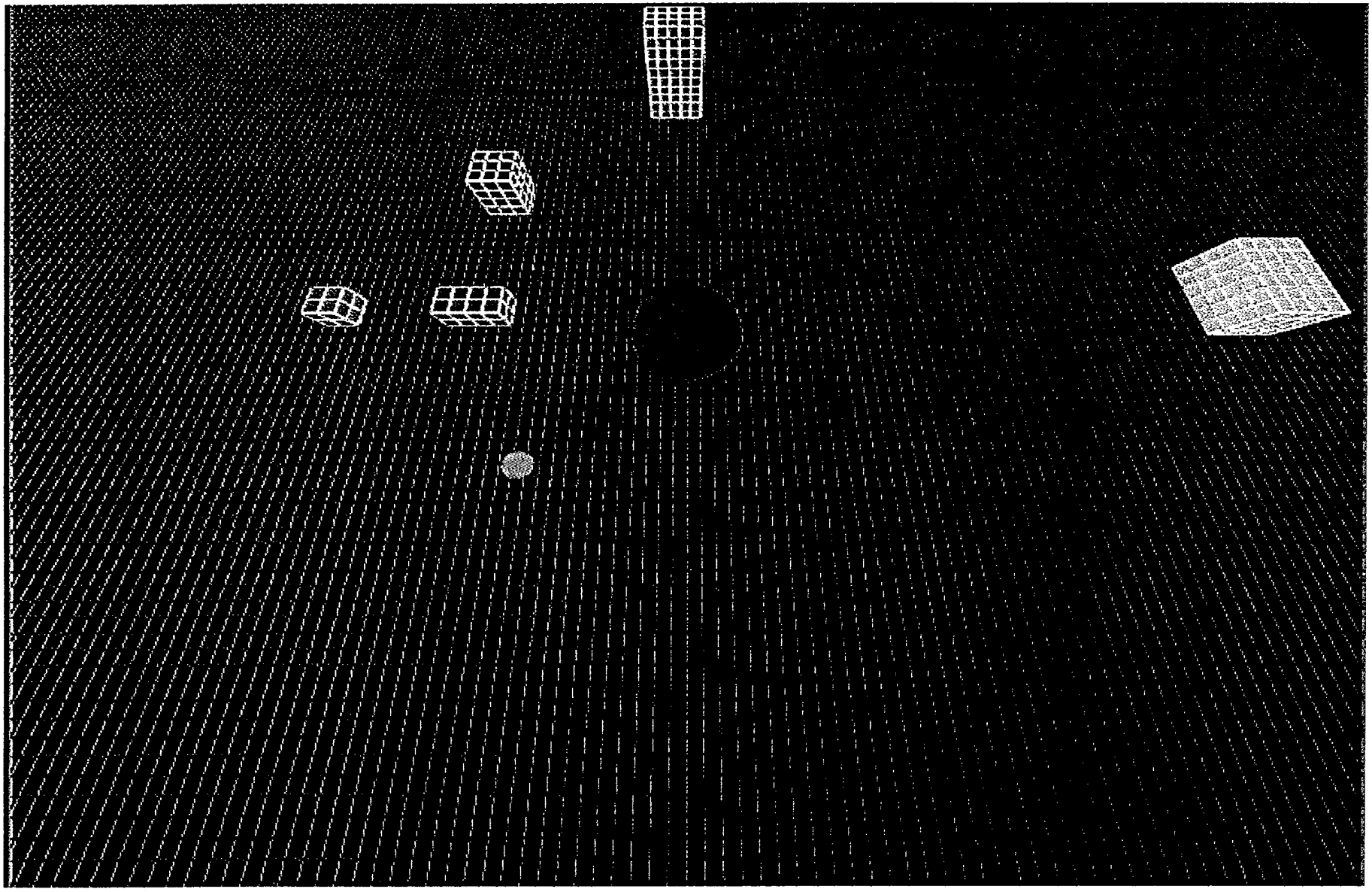

Figure 7: Overhead View of Visual Model.

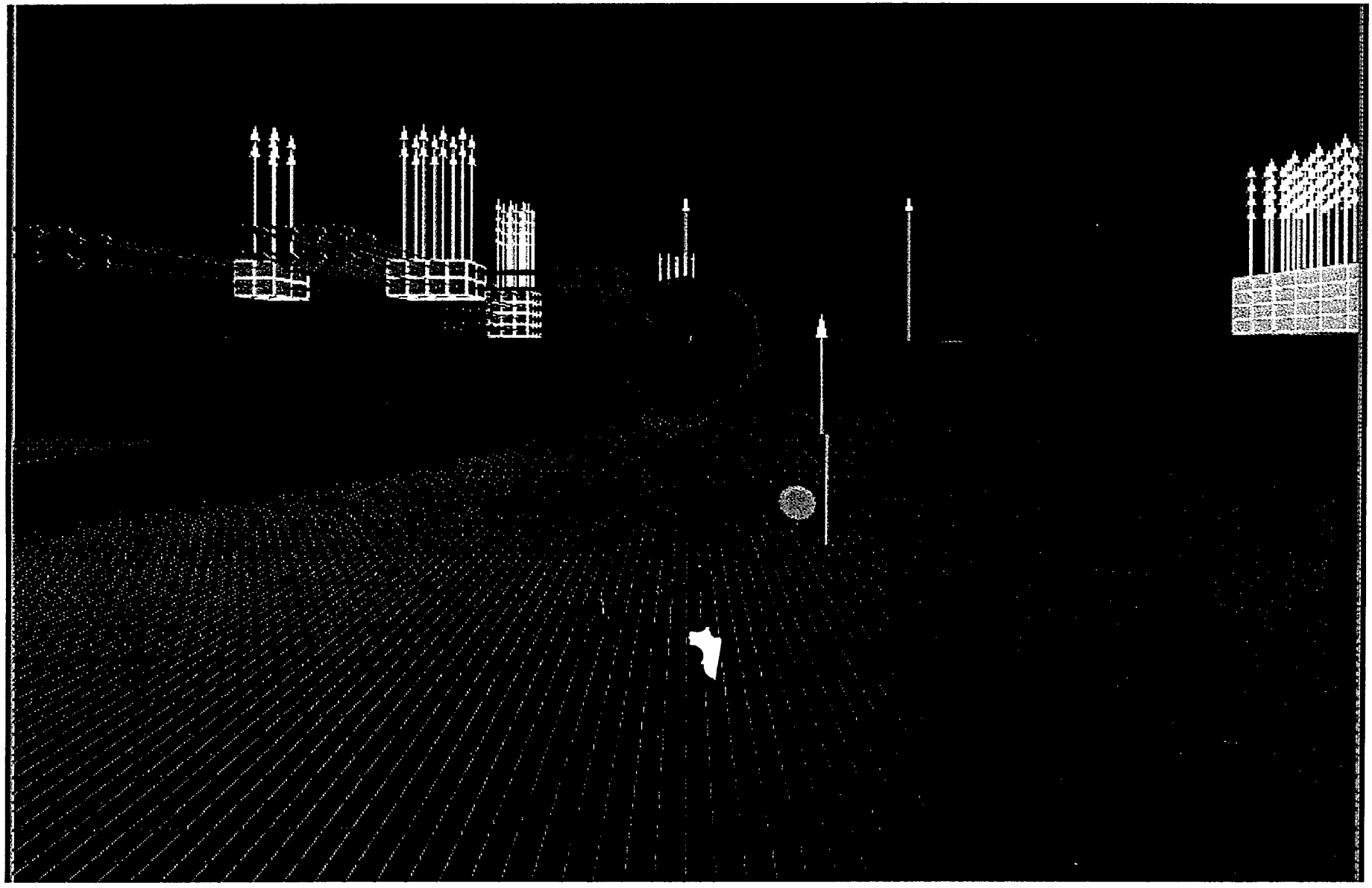

Figure 8: Visual Representation with Orientation Depicted.

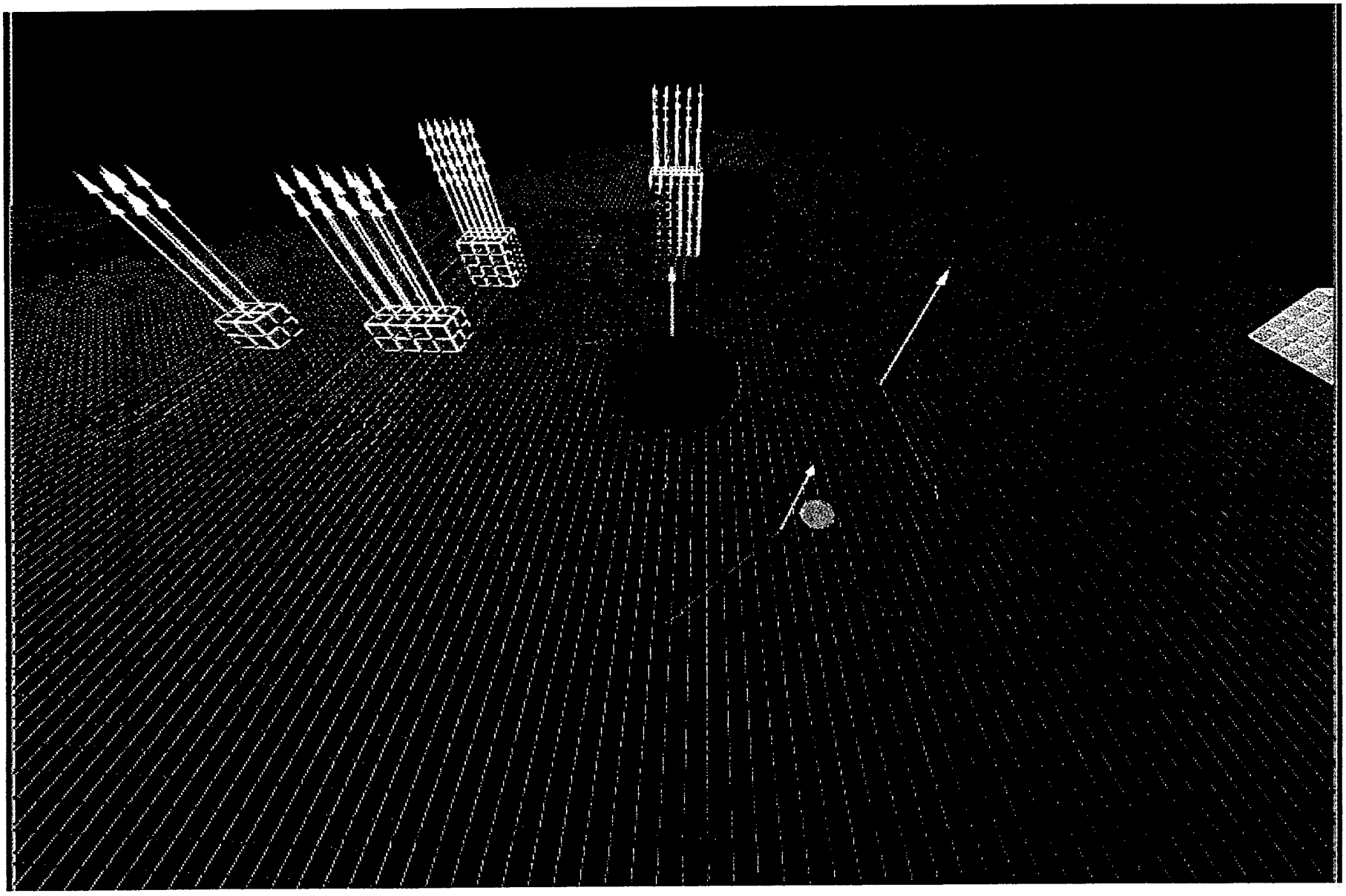

Figure 9: Orientation with Altered Perspective.

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 4-9 show the initial 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 may significantly improve software surety capabilities as well as debugging productivity, and program comprehension. Additional studies will be needed for conclusive documentation.

Consequently, 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. For example, if a routine must have some interaction within a specific time period, such as a monitoring routine, and it does not received the required "signal" within the necessary time frame, an unsafe condition may be triggered. Visually this could be represented as an object dropping towards the floor as it awaits a signal. If it goes beneath the floor plane it has "timed out". This would be analagous to a night watchman signalling "all's clear" at specific intervals. Watching the object drop would give a more noticable response to the user.

While not discussed in depth due to space limitations, the model can also identify situations that have not been addressed by the requirements constraint language. If an action is taken by the program that does not map to a constraint specification, an unspecified situation has occured if the complete specification is given. This has an important impact of providing 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 program 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 provides a first step in allowing the user 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 a particularly onerous requirement. In addition, the user must have a similar platform available.

Furthermore, 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. Finally, the user must utilize current visual models until they develop their own models.

Future Extensions

Future work will focus on incorporating multiple world views, providing more control over the model by the user, and expanding available visual models. Work will also progress in completing the requirements constraint monitor for the SAGE constraint programs. The prototype will be expanded to handle more typical real world applications. Once that work is completed. Studies will begin to test the effectiveness of the tool. Later work should expand the environment to visualize the specification phase of the software.

7. Requirements 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. In 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 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) [17], clp(fd) [3], ECL^{i}PS^e [18], CIAO [19], and Oz [20]. These systems generally consist of two levels, the underlying 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. 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 runtime to prohibit select, critical errors.

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.

Syntax Issues

The syntax for these operations are:

[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, ...* }

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 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.

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.

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. The constraints that might be coded to represent this type of function is as follows:

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

Surjection_Count == $\Sigma\, \forall\, A \Rightarrow B$;

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.

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.

8. Example Usage

Figure 10 *(a)-(d)* illustrates example usage of the prototype tool. Figures 10*(a)* and10*(b)* show various programs, and how they would appear initially. Of course, the actual visualization would appear similar to Figures 4-9. These examples are illustrated to minimize space. Figures 10*(c)* and 10*(d)* illustrate error conditions that could occur as desired by the user. Additional examples may include the following:

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

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

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

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

* Wrong array is accessed to retrieve or alter a value (invalid pointer)

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

* Statement alters data structures when it is not expected (side effects)

* 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)

* Program violates stated semantic patterns for execution sequences

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

* Conditions not set in proper order (concerning variable states)

* More statements executed than expected

* Changes in execution pattern

* Execution of rarely executed code

* Formation of discrepancies in link patterns

* Unusual formations of data structures

9. Conclusions

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 within the software specifications. 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. The expected payoff is increased software surety confidence. In addition, increased program comprehension and reduced development and debugging time are expected to be achieved. 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.

10. References

1. 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.

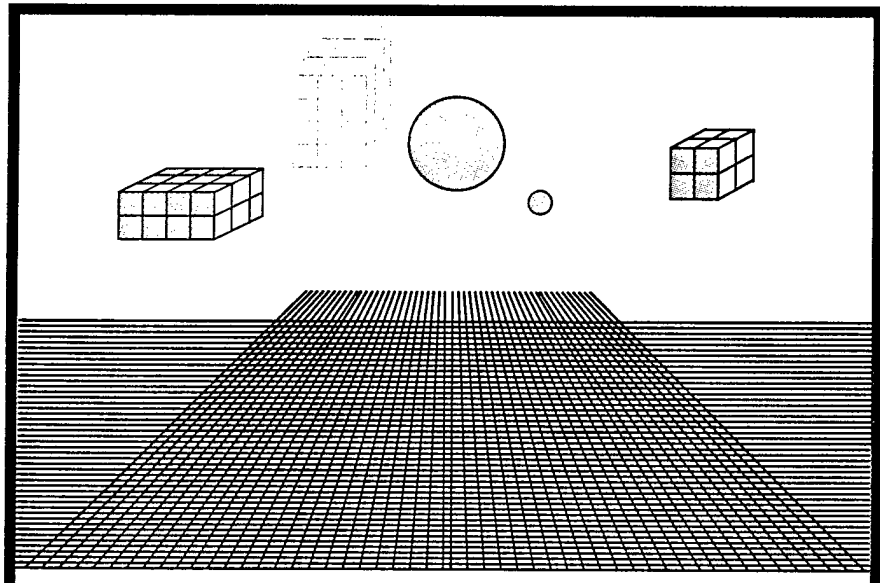

Figure 10 (a)**:** Sample Program Execution, Data Structures

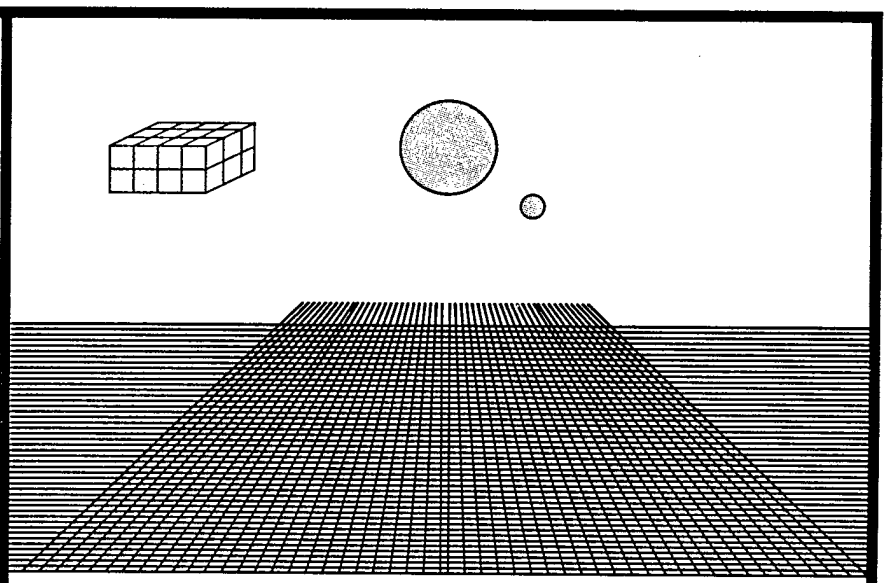

Figure 10 (b)**:** Sample Program Execution, Simple Model..

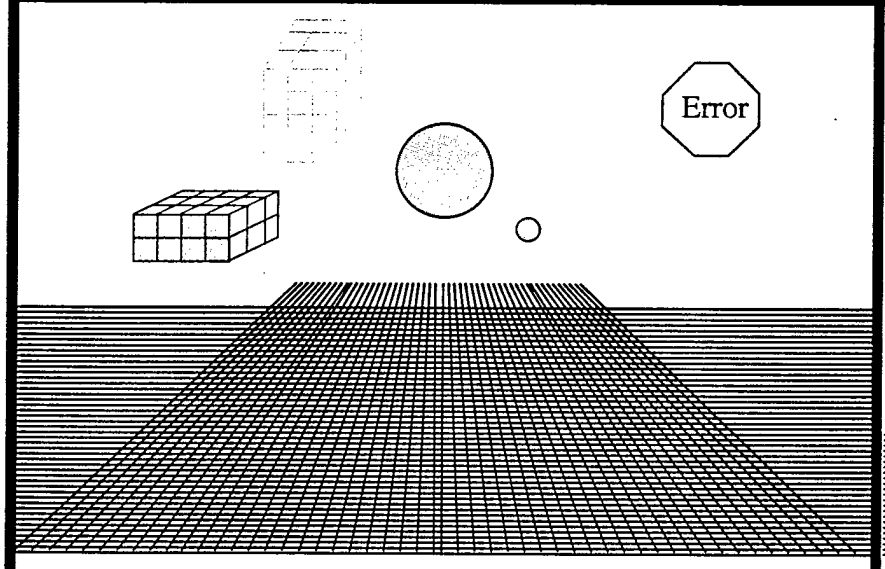

Figure 10 (c)**:** Sample Error Detection, Visual Model 1.

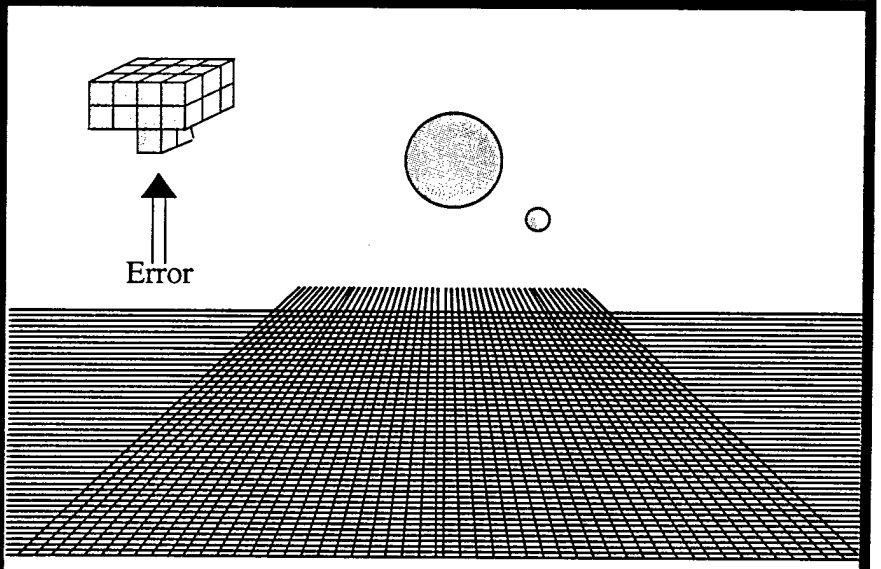

Figure 10 (d)**:** Sample Error Detection, Visual Model 2.

2. 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.

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

4. 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.

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

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

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

8. 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.

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

10. 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.

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

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

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

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

15. 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.

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

17. 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.

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

19. 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.

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

11. BIOGRAPHY

Guylaine M. Pollock, a Senior Member of the Technical Staff at Sandia national Laboratories, received a Ph.D. in computer Science from Texas a & M University and a BS in computer Science and Mathematics from East Texas State University, graduating with Academic Distinction and Highest Honors. She has served on Software Capability Evaluation Teams for the Battle Management Defense Organization of the Department of Defense. She has investigated software Reliability for massively parallel codes and is a member of the Sandia Reliability Working Group. Dr. Pollock is a member of the Board of Governors of the IEEE Computer Society, and currently is serving as 1st Vice President of Conferences and Tutorials. She previously lectured with the IEEE Computer Society Distinguished Visitors Program. Pollock has received several awards including the Richard E. Merwin Scholarship and Notable Women of Texas. She is a Golden Core Member of the IEEE Computer Society.

This work was supported by the United States Department of Energy under contract DE-AC04-94AL85000.

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.

Report Number (14) SAND--98-0426C
CONF-980319--

Publ. Date (11) 199802
Sponsor Code (18) DOE/MA, XF
UC Category (19) UC-900, DOE/ER

DOE