Visualization of Concurrent Computations: Doctor of Science Dissertation Proposal

Authors: Kenneth C. Cox

Visualization, defined as the graphical representation of symbolic objects and processes, is recognized as an important tool to aid human understanding. This is particularly true in the area of program visualization, which uses images to illustrate the execution of programs. This proposal describes research to investigate the visualization of concurrent computations. The research has two major goals: the development of a model of visualization suitable for concurrent computations, and the development of methodology for constructing visualizations. The proposed visualization model treats visualization as a function from the state of the computation to an image. This differs significantly from the approach used by existing (single-process) visualization systems, but seems highly suitable for concurrent computations. The proposed methodology uses the concepts of program correctness to identify the key properties of a computation and suggest methods of visualizing these properties.

... Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation

https://openscholarship.wustl.edu/cse_research/745
Visualization of Concurrent Computations: Doctor of Science
Dissertation Proposal

Complete Abstract:

Visualization, defined as the graphical representation of symbolic objects and processes, is recognized as an important tool to aid human understanding. This is particularly true in the area of program visualization, which uses images to illustrate the execution of programs. This proposal describes research to investigate the visualization of concurrent computations. The research has two major goals: the development of a model of visualization suitable for concurrent computations, and the development of methodology for constructing visualizations. The proposed visualization model treats visualization as a function from the state of the computation to an image. This differs significantly from the approach used by existing (single-process) visualization systems, but seems highly suitable for concurrent computations. The proposed methodology uses the concepts of program correctness to identify the key properties of a computation and suggest methods of visualizing these properties.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/745
VISUALIZATION OF CONCURRENT COMPUTATIONS
Doctor of Science Dissertation Proposal

Kenneth C. Cox

WUCS-89-32

June 1989

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899
Visualization of Concurrent Computations
Kenneth C. Cox

Abstract

Visualization, defined as the graphical representation of symbolic objects and processes, is recognized as an important tool to aid human understanding. This is particularly true in the area of program visualization, which uses images to illustrate the execution of programs. This proposal describes research to investigate the visualization of concurrent computations.

The research has two major goals: the development of a model of visualization suitable for concurrent computations, and the development of a methodology for constructing visualizations. The proposed visualization model treats visualization as a function from the state of the computation to an image. This differs significantly from the approach used by existing (single-process) visualization systems, but seems highly suitable for concurrent computations. The proposed methodology uses the concepts of program correctness to identify the key properties of a computation and suggest methods of visualizing these properties.
# TABLE OF CONTENTS

1. MOTIVATION ......................................................................................................................... 1

2. BACKGROUND ....................................................................................................................... 2
   2.1. Definitions and Motivation ............................................................................................ 2
   2.2. Survey of the Visualization Field .................................................................................. 3
       2.2.1. Visualization in Scientific Computing ................................................................. 3
       2.2.2. Visual Programming ............................................................................................. 4
       2.2.3. Program Visualization ......................................................................................... 5

3. GOALS .................................................................................................................................... 7

4. OBJECTIVES .......................................................................................................................... 8

5. APPROACH ............................................................................................................................ 9
   5.1. Visualization Model ......................................................................................................... 9
       5.1.1. Representation of the Domains of the Model ....................................................... 11
       5.1.2. Specification of Visualization Functions ............................................................ 13
       5.1.3. Visualization Algorithms ..................................................................................... 14
   5.2. Visualization Methodology ............................................................................................. 15

6. RESEARCH PLAN ................................................................................................................... 17

7. REFERENCES .......................................................................................................................... 19
1. MOTIVATION

Visualization is defined as the graphical representation of symbolic objects and processes. The research outlined in this proposal will investigate the visualization of large-scale concurrent computations running on multiprocess and distributed systems. The research will focus primarily on the design of a model suitable for investigating visualizations of such computations and on the development of a methodology for constructing visualizations under the model.

In recent years the computer field has seen the development of highly parallel computer systems, both through the use of parallel hardware and through the interconnection of numerous systems. These parallel systems provide great computational power and permit investigation of problems for which traditional systems are unsuitable. However, parallel systems also require distinctly different programming techniques than those used in sequential programs. These techniques are largely in a formative stage, and this has limited our ability to exploit the power of such systems.

A requirement for successful programming, both in the design and the testing phases of program development, is an understanding of program behavior. For sequential programs, a number of tools and software engineering techniques have been developed which allow programmers to break problems into manageable design units and observe the behavior of running processes. By using these tools and techniques, the programmer can obtain the necessary understanding of the program. Unfortunately, tools and techniques to aid in the understanding of large-scale, highly parallel programs are poorly developed.

Human ability to understand parallel computations is limited by the sheer volume of information which is involved. A large parallel program could involve thousands of processes and hundreds of thousands of inter-process communications. Simply collecting information about the computation involves a number of complex problems. Absorbing this much information is beyond human capabilities; merely grasping the general direction of computation is difficult without effective abstractions of the information. However, the abstraction of the computational state presents its own difficulties. Existing software engineering techniques and tools are based on the assumption that the program is, at any time, performing an identifiable action; this is reflected in everything from the tidy labeled boxes of flowcharts to the linearized sentences of specification languages. When the program consists of many independent processes which may be at different points in the performance of a variety of tasks, these
methods break down.

Visualization has been applied to a number of problems, both in computer science and in related fields; a brief survey is included in section 2 of this proposal. However, application of visualization as a tool for understanding parallel computations has not been deeply explored. It is my belief that visualization will prove to be useful in the understanding of parallel computations. This is based on the observation that humans can process image information faster than textual information — aptly expressed by the proverb, "A picture is worth a thousand words." The human ability to detect, recognize, and interpret shapes, patterns and colors is amazingly fast and powerful. Psychological and physiological studies seem to indicate this is because human image-understanding is itself a "hard-wired", highly parallel process, while reading is essentially linear.

The excellent human image-understanding abilities directly address the difficulties that traditional techniques and tools have with parallel computations. When translated into a (changing) image, the great volume of information involved in a parallel computation can be processed in parallel by the visual system. Similarly, by appropriate use of shapes, patterns and colors\(^{101}\) the information about the computation can be easily abstracted. These factors support the thesis that visualization will prove an effective tool to aid in the understanding of parallel computations.

2. BACKGROUND

2.1. Definitions and Motivation

In its broadest sense, visualization is the representation of information in a form meant to be viewed. However, as pointed out by Shu\(^{142}\), this definition includes writing — a fact which is somewhat more obvious when considering Chinese ideographs or Egyptian hieroglyphics. For my purposes, the more restricted definition of visualization suggested by the Workshop on Visualization in Scientific Computing\(^{102}\) is more appropriate: “visualization is a method of computing which transforms the symbolic into the geometric, enabling researchers to observe their simulations and computations.”

The primary impetus toward visualization has been the recognition that human beings generally understand information better when it is presented in image or graphical form than when it is presented textually. This is largely because information presented as text must be absorbed linearly. However, because the human visual sys-
term is believed to operate essentially in parallel, much higher processing speed can be obtained. This ability is recognized in the old saw “a picture is worth a thousand words”, as well as by numerous researchers. These researchers emphasize that imagery is a valuable tool in the areas of data presentation, man-machine interaction, and instruction.

However, it should be noted that a trade-off is involved when information is presented in image form. Textual information, while not conducive to overall understanding, is capable of great precision; image information seems better for presentation of gestalt knowledge but cannot give details. The primary use of visualization – as emphasized in the Workshop report – must therefore be to provide insight and abstraction of data, not to transmit large volumes of precise information.

2.2. Survey of the Visualization Field

The Workshop definition of visualization includes a large body of research, with a large number of basically independent areas. Various researchers, including Chang, McCormick et al, Myers, Raeder, and Shu, have suggested taxonomies of visualization applications and methods. The taxonomy used in this survey is largely based on Myers and Shu with additional classes suggested by the Workshop. The major classes of this taxonomy are visualization in scientific computing, visual programming, and program visualization.

2.2.1. Visualization in Scientific Computing

Visualization in scientific computing or ViSC is a term defined by the Workshop. The information processed by ViSC is primarily scientific data, such as is generated by supercomputer simulations, satellites, and measuring devices used in astronomy, meteorology, geology, and medicine. The number of such data sources and their data-generation rate has steadily (even exponentially) increased, outstripping both the growth of the number of researchers in the fields and the development of data-processing tools. As a result, much of the data cannot be processed and must be stored, perhaps never to be examined and interpreted.

The goal of ViSC is to present scientific data in image form for the purposes of accelerating the interpretation of the data and permitting scientists to interact with the processing of data. The Workshop proposal is to accelerate the development of tools for presentation of data in image form, making visualization techniques available to
research scientists and engineers.

Although ViSC is clearly an important area of visualization, it falls outside the main thrust of this proposal. However, the fact that so many researchers from academia and industry feel that visualization is an important and powerful tool for understanding is significant\textsuperscript{55}.

2.2.2. Visual Programming

Visual programming, also called graphical programming, is the specification of programs in two or more dimensions. (Textual programming languages are not visual languages because the program is a one-dimensional character stream.) This area has been the subject of a great deal of research and a number of systems have been developed. Shu\textsuperscript{42} and Chang et al\textsuperscript{61} have produced books dealing with visual programming and visual languages; the papers by Chang\textsuperscript{32}, Levis\textsuperscript{91}, Myers\textsuperscript{110,111}, and Raeder\textsuperscript{122} present brief surveys. Each of these authors provides a different taxonomy of visual programming systems; the most thorough treatment is probably that in Shu. A complete survey of visual programming systems is out of the scope of this proposal; only the more significant systems are described.

One of the earliest class of visual programming systems is called “diagrammatic systems” by Shu. These were originally based on flowcharts and related software engineering techniques. The 1969 Graal system\textsuperscript{47} compiled flow charts containing assembly language into programs; the similar (though more recent) FPL\textsuperscript{153} and Pascal/HSD\textsuperscript{41} systems generated Pascal code from flowcharts. Nassi-Shneiderman flowcharts\textsuperscript{112} have also been used as an input representation in, for example, the Programming Support System\textsuperscript{54} which generated PL/I, GAL\textsuperscript{5} which generated Pascal, and the PIGS (Programming with Interactive Graphical Support) and Pigsty systems\textsuperscript{119,120}, which generated Pascal. Pigsty, an extension of PIGS, is particularly interesting because it generates Pascal-CSP programs for concurrent processes, using Nassi-Shneiderman diagrams extended by guarded commands. Each Pigsty process is represented by a box; processes communicate by CSP-type semantics. Diagrammatic languages are also based on dataflow graphs (FGL\textsuperscript{83}, PROGRAPH\textsuperscript{100,117}, and LZ\textsuperscript{17}), Petri nets (VERD\textsuperscript{63} and MOPS-2\textsuperscript{2}), state transition diagrams (USE\textsuperscript{158,159}), and general graphs (AMBIT/G/L\textsuperscript{134} and UIMS\textsuperscript{79-81}).

A second major category of visual programming languages is the iconic systems, in which programs are constructed by combining small pictographs or icons in various ways. Many of these are extensions of the
diagrammatic languages, in which the remaining text (i.e., in a flow chart box) is replaced with icons. Such systems include HI-VISUAL\textsuperscript{71} which is essentially an iconized dataflow language, and IBGE\textsuperscript{154} which is flowchart-based.

The Upcon system\textsuperscript{15} superficially resembles the flow chart systems, but permits the user to interconnect individual UNIX\textsuperscript{128} processes through the UNIX pipe mechanism.

More recent languages use more powerful paradigms for combining icons. The VennLisp system\textsuperscript{88} represents LISP code graphically by enclosures. Tinkertoy\textsuperscript{45} and Proc-BLOX\textsuperscript{61} use "jigsaw-puzzle" icons which can only fit together in syntactically-meaningful ways. The Pict system\textsuperscript{60} permits icon definition and recursion; groups of connected icons can be defined and used within other icon definitions, or recursively. The BOXER\textsuperscript{44,140} and ConMan\textsuperscript{65} systems have similar capabilities. Show and Tell\textsuperscript{85,86,103} and PLAY\textsuperscript{152} are also similar, but intended primarily for instruction of children.

The systems characterized by Shneiderman\textsuperscript{141} as direct manipulation systems -- systems whereby a user modifies system or program information through icons -- form another important group. Such systems have greatly proliferated in recent years, due to their perceived utility as workstation interfaces. The Xerox Star system\textsuperscript{121,143} is typical; it is generally credited with the development of the "desktop" metaphor for man-machine interaction. Similar systems include Cedar\textsuperscript{155}, X-Windows\textsuperscript{136}, and the numerous proprietary workstation interfaces. By Shneiderman's definition, video games are also included in this area; the user interacts with the program data by manipulating icons.

A final group of visual programming systems is designed not as a method of representing programs or data, but as tools to aid in program development (including database and user interface development). These tools allow the program or database structure and text to be displayed in graphical form; most allow a debugging-type interaction whereby the program can be single-stepped and the results used to transform the image. Such systems include PECAN\textsuperscript{124}, GARDEN\textsuperscript{125,126}, InterViews\textsuperscript{94} ISQL\textsuperscript{8}, Macpeth\textsuperscript{148}, PegaSys\textsuperscript{107}, PV (Program Visualization)\textsuperscript{22,69,70}, and ThinkPad\textsuperscript{135}.

2.2.3. Program Visualization

Program visualization, also called algorithm animation, refers to the use of images to represent some aspect of the execution of a program. Both representations of program data structures and representations of program
execution are included. A number of visual programming systems, including VennLISP, Pict, Show and Tell, and PV, are able to trace the execution of the program and produce a graphical representation; this technically moves them into the area of program visualization. However, these systems are primarily oriented toward specification of the program, not execution, so are more closely related to visual programming.

Program visualization has a long history. Sutherland in 1966 identified the “coupling” of graphical input/output to program behavior as an important area of investigation. The earliest such systems (other than the console lights) were simply extended monitors which graphed memory accesses versus time; in 1972 Hatfield investigated the performance of paging systems using these graphs. In 1974, the ANTICS system for animating LISP programs was presented. Although somewhat limited by the graphics capabilities of the time, ANTICS presented 2-D image representations of data structures in running LISP programs. The similar systems presented by Baecker in 1975 also produced representations of data structures, but were intended primarily as aids to understanding algorithms; the 1981 film Sorting Out Sorting presented a number of sorting algorithms using these systems. In a somewhat bizarre reversal of the Programming Support System, Clark and Robinson in 1983 presented a monitor which displayed program execution as a sequence of Nassi-Shneiderman diagrams. This system is perhaps most notable because it was implemented as a preprocessor; Pascal code was filtered and appropriate commands to the graphics generator were automatically inserted.

One of the most ambitious and influential algorithm animation systems was introduced in 1983 at Brown University. The BALSA environment provided facilities for constructing, saving, and playing back animations of various algorithms. The system is primarily instructional, with the animations serving as aids to algorithm understanding. The methodology for animation used by BALSA is incorporated in many of the later animation systems. The algorithm designer/ animator determines events of interest in the algorithm; the algorithm code is then augmented with procedure calls to signal these events. The animation package maintains the image based on the interesting events generated by the running algorithm. Thus, image generation is treated as a side effect of the running algorithm.

Systems with a similar approach to algorithm and data animation include TANG, PROVIDE, Incense, Visible Pascal, PERUSE, the Smalltalk-based MVC system, and Object-Oriented Diagramming. All these systems require the animator to identify events and objects of interest; most require the insertion
of procedure calls to arrange for changes in the display.

Some recent systems have used different approaches to process visualization. TPM, or the Transparent Prolog Machine\textsuperscript{46}, automatically animates Prolog programs. The TPM animation mechanism is built into the Prolog interpreter and automatically presents the database search and unification attempts. The ALADDIN system\textsuperscript{68,78} uses a declarative approach to algorithm animation, allowing the programmer to specify the animation using a catalog of pre-defined graphical and animation primitives. Programming animations in the system is similar to that in BALSA, in that the user must embed update instructions in the program code; however, the update instructions are entered in an iconic notation! All ALADDIN graphical objects are separately declared; the update instructions modify the attributes of these objects. The PVS system\textsuperscript{51}, intended for the monitoring of manufacturing processes, uses a database of process variables (sensor readings) which can be accessed by a visualization system. A programmer constructs visualizations by defining icons with attributes determined by the data values; the visualization system monitors the database and presents the icons. The overall system is similar to PROVIDE, but does not require the underlying process to recognize key events.

3. GOALS

Although a great deal of progress has been made in the area of algorithm visualization, existing work is not directly applicable to the investigation of concurrent computation visualization. Existing systems are strongly oriented toward the visualization of single-process, sequential algorithms. This assumption affects all aspects of the visualization systems, from the algorithms which generate the images through the environments with which the user interacts with the system. In addition, in most of these systems visualization is treated as a side effect of the underlying computation – i.e., the computation sorts a list and also draws some pictures. This requires the constructor of a visualization to identify the key events in the program code and arrange for these events to trigger changes in the image (usually by inserting procedure calls to a subroutine library). Because of this use of small incremental changes to the image, construction of a visualization with an appropriate level of abstraction is difficult.

I propose to address these problems by developing a visualization model suitable for concurrent computations. The model will be based on a declarative approach to visualization, in which visualization is treated as the application of a function to the computation state, producing a collection of graphical objects (the translation of the
graphical objects to an image is mechanical and of little theoretical interest). This functional approach will decouple the construction of a visualization from the actions of the underlying computation, permitting the visual presentation of the state information at a level of abstraction appropriate to the understanding. The shared dataspace paradigm will be used as the underlying computational model because it conveniently abstracts the computational state and permits declaration of the visualization function in the form of brief yet powerful rules.

To make effective use of the model, I will develop a methodology for constructing visualizations in the model. The purpose of the methodology will be to provide a sound technical foundation for the construction of visualizations as aids to understanding. The methodology will be based on program verification, which seeks to explain computations by formally describing properties of the computation – both safety properties (conditions that the computation will not violate) and liveness properties (actions which the computation must perform). I believe that these computation properties are exactly those which best aid understanding and should be represented in the visualization; further, the form of each property may indicate an appropriate method for visualizing it.

4. OBJECTIVES

As stated above, the primary objectives of this research are a model and methodology for visualization of concurrent computations. The model will use a declarative approach to visualization in which visualization is treated as the application of a function to the computation state, generating a collection of graphical objects. The development of the model will encompass three major activities:

- Developing a formalization of the computation state and graphical objects. This requires selecting a computational model for distributed systems, defining what aspects of the computational state of the selected model should be visualized, and deciding on an appropriate universe of graphical objects.

- Defining the method for specifying visualizations as mappings from the computation state to collections of graphical objects. This will require development of appropriate methods for declaring the visualization functions in terms of the selected representations of the computational state and graphical objects.

- Exploring algorithms for performing visualizations in the model to demonstrate the feasibility of the model. This requires developing, analyzing, and implementing highly concurrent algorithms for implementing the visu-
alization function. These algorithms fall into two main groups: state-collection algorithms for producing a representation of the computational state, and function-application algorithms for producing an image from the collected state.

The methodology will aid in the construction of visualizations in the model by giving methods for translating key program properties into visual form. Important activities contributing to the development of the methodology include:

- Formulating rules for translating the formal properties of the program specification into visual representations. The rules should construct the visual representations based on the formalisms and logical structure used to express the properties; informal, intuitive knowledge of what the program "should do" should not form part of the methodology rules.

- Demonstrating the effectiveness of the methodology by construction of visualizations for actual problems. The methodology will be judged by the effectiveness of the resulting visualizations as aids to understanding.

Certain subsidiary and supporting work, such as the construction of a prototype distributed transaction system as a testbed for the development of visualization algorithms, will also be required.

5. APPROACH

5.1. Visualization Model

I consider a visualization to be a function from the state of a computation to an image. This is not a new concept, and indeed lies behind many of the existing systems for algorithm animation. The difficulty with most of these systems is that they fail to properly abstract the concept. For example, in BALSA\textsuperscript{24} and similar systems the point is made that the image represents the state of the computation. However, these systems do not view construction of the image as application of a function, but as a side effect of the execution of the computation. The user must identify significant program events which change the image and augment the program code at these events with subroutine calls to accomplish the changes. Construction of an animation in these systems is more concerned with modifications of various data structures representing the image than with the application of a function.
The ALADDIN system\textsuperscript{77} is somewhat closer to a functional representation, in that a collection of objects is declared and modifications to the objects are specified. However, ALADDIN still requires the modifications to be inserted into the program text (albeit in a graphical form). The PROVIDE\textsuperscript{106} and PVS\textsuperscript{51} systems are still closer to functional declarations. In these systems objects (icons) are specified with parameters depending on process variables; changes to the variables are automatically transmitted to the display mechanism. The principal shortcoming of these systems is the low level of abstraction. The graphical objects making up the image are necessarily closely related to the program variables, and no mechanism for higher levels of abstraction is available (for example, it is not possible to detect and display concepts such as "all variables have values greater than 10").

If, as I assert, visualization is best represented by a function, then the natural method of describing a visualization is to declare the function. This leads to declarative visualization, the approach I will take in this research. Declarative visualization is distinguished from imperative visualization, the technique used in BALSA and similar systems, which treats the visualization process as a side-effect of the underlying computation. In declarative visualization, the visualization is declared as a function from the computation state to images.

The declarative approach to visualization has a number of advantages, particularly in application to multi-process and distributed systems. The evaluation of the visualization function can be performed independently of the underlying computation, for example by a superimposed computation which collects state information and applies the function. Because of this independence of the visualization and the underlying computation, visualizations can be added to existing programs without modification of the existing code. Further, a visualization can be added to a running computation, or the existing visualization can be changed, simply by modifying the visualization function; the underlying computation is not perturbed by such actions.

The technical issues associated with the model are primarily concerned with the representation of the visualization function and its domain and co-domain. Letting $V$ represent such a visualization function, the domain of $V$ is the set of all possible computation states $S$ and the co-domain of $V$ is the set of all possible images $I$:

$$V : S \rightarrow I$$

The function $V$ could be declared in this fashion, with the programmer determining the color of each pixel in the image from the computational state. However, this introduces certain difficulties; in particular, the programmer is
too concerned with details of rendering the image and cannot devote proper effort to achieving a suitable level of abstraction.

The need to separate the operations of abstraction and rendering leads me to believe that the function \( F \) can best be declared as the composition of two functions. The abstraction function permits the necessary level of abstraction by mapping the computational state to a high-level collection of graphical objects (entities such as lines, circles, squares, and so forth). The rendering function then translates the collection of graphical objects into an image. Representing the abstraction function by \( A \), the rendering function by \( R \), and the set of all possible collections of graphical objects by \( O \), the visualization function \( V \) can then be specified by:

\[
V = A \circ R \\
A : S \rightarrow O \\
R : O \rightarrow I
\]

5.1.1. Representation of the Domains of the Model

In many languages for expressing concurrent computations\(^7\),\(^21\),\(^66\) the computation is modeled (directly or indirectly) as a collection of shared variables which are accessed by a number of processes, each of which executes some piece of code and has certain local variables. A state in such a computation (that is, a single element of the set \( S \)) would be represented as the values of all the variables, and the 'program counters' and code of all processes.

Similarly, the state of computations written in languages which use a message-passing paradigm\(^3\),\(^72\) can be represented by including the variables, program counters, and the contents of any messages currently 'in transit'. Such representations are perfectly acceptable and useful in many applications. However, I do not feel that it is suitable for this research. The primary focus of this research is on the visualization model and methodology, and feasibility. Using a computational model with the above attributes requires formulation of a mechanism for obtaining the state information from processes, which does not contribute to this research.

A class of languages which avoid the need to obtain state information from particular processes and permits a higher level of abstraction has been investigated by several researchers\(^4\),\(^52\),\(^58\),\(^127\),\(^129\),\(^131\). These languages use the shared dataspace\(^130\) communications mechanism. In this mechanism, state information is stored in a common, content-addressable data structure called the dataspace. Processes (or equivalent mechanisms) are able to examine and modify the contents of the dataspace. In most of the shared dataspace models, a high degree of data abstraction
is provided by organizing the dataspace as a set of typed tuples (vectors of data) and providing mechanisms to address the dataspace through the contents of the tuples (for example, by matching a given pattern against the tuples).

The fact that all information about shared-dataspace computations is represented in the common data structure obviates the need for a separate data-extraction mechanism to collect state information for visualization. The tuple-based representation of the dataspace and the ability to access information by patterns permits powerful abstraction mechanisms. Since all the information about the state of the underlying computation is readily accessible and external to the actual computation mechanism, a visualization can be added to an existing computation – even an executing one – without modifying the computation or code.

Because of these advantages, I have chosen the shared dataspace paradigm as the underlying computational model for this research. I will ensure that the model and methodology can be generalized to apply to any computational paradigm in which the concept of a state is well-defined, provided a suitable state-extraction mechanism is added.

The shared dataspace model I will use is one in which the dataspace is a finite set of tuples. Each tuple has a particular type and an arbitrarily-long vector of data elements; this is represented by the notation

\[ \text{type}(element_1, element_2, ..., element_n) \]

In the proposed visualization model, the space \( S \) consists of all possible computational states. Each element of \( S \) is therefore one of the above dataspaces, i.e. a finite set of typed tuples. For consistency, the sets \( O \) (all possible graphical objects) and \( I \) (all possible images) will be represented in the same way.

Each element of the set \( O \) is a collection of objects drawn from some primitive graphic object space. Each object can be represented as a tuple, where the tuple type is the class of object (line, circle, etc.) and the data elements are parameters specifying the particular instance (position, orientation, size, color, etc.). The selection of a suitable universe of primitive graphical objects is a secondary research effort; McCleary's survey\(^{101}\) suggests certain objects and object properties which are effective. The need to permit high levels of abstraction in the mapping from \( S \) to \( O \) encourages the selection of a large variety of object types and a rich selection of object parameters. Countering this is the requirement that the objects be rapidly and efficiently rendered as an image on some actual
device. Since the primary focus of this research is on the process of abstraction and not that of rendering, the universe of graphical objects will be large and powerful. (The rapid development of high-speed dedicated image-rendering hardware adds credibility to this design decision.)

Each element of \( I \) is an image. This can be modeled as an \( M \times N \) array of pixels, where each pixel can have any of several values (intensities or colors). An image in this model can be easily represented as a collection of tuples.

5.1.2. Specification of Visualization Functions

In declarative visualization, the construction of a visualization for a computation is accomplished simply by defining the function which maps the computational state to an image. In this research, the mapping is composed of two functions: an abstraction function \( A \) to map computation states to graphic object states, and a rendering function \( R \) to map graphic object states to images. The visualization function \( V \) is the composition of these functions.

Formally, a function \( F : A \rightarrow B \) is a subset of \( A \times B \); we say that \( F(a) = b \) if \( (a,b) \in F \). In practice, functions are rarely specified by enumeration of pairs; instead, rules are given for determining the value of \( F(a) \) from \( a \). In procedural languages, these rules are expressed by segments of code which are executed when the function is to be computed; the code segment and associated name and parameter information declare the function. Most shared dataspace languages do not use this method to declare functions, and indeed do not support functions per se.

Instead, they specify transformations to be applied to the dataspace. A common method of specifying such a transformation (particularly in systems intended for the implementation of expert systems\(^{39,52,113} \)) is the production rule. A production rule (called in some languages a transaction) has the general form

\[
\text{list of patterns} \rightarrow \text{list of actions}
\]

Such a rule indicates that, if a collection of entities (tuples) which match the list of patterns exists in the dataspace, the rule is "fired" and the actions are executed. A mechanism (typically bound variables) for carrying information from the pattern-matching to the actions is generally provided. Depending on the language, the permitted patterns can be quite powerful and permit strong abstractions (including negated and universal quantifiers). The general
term for such pattern-matches is a *query*.

I propose to adopt a similar notation for the specification of the abstraction and rendering functions. The abstraction function will be specified by a collection of rules of the form

\[
\text{query over tuples in computation space} \\
\rightarrow \\
\text{list of tuples in object space}
\]

The collection of rules defines the object state resulting from a particular computation state. For each rule, the query is evaluated, and for each successful collection of computation tuples which match the pattern the list of object tuples is computed. The object state consists of all the object tuples so produced. The rendering function can be similarly specified as a collection of rules performing queries over the object space and producing tuples in the image space; since the rendering function is not of primary interest, the development of such rules will not be emphasized.

5.1.3. Visualization Algorithms

Development of the declarative model of visualization will be a major part of the formal aspects of this research. To make the work of practical interest, I will design, analyze, and implement algorithms for performing visualizations in the model. Two types of algorithms are of major importance: state-collection algorithms to determine the computational state, and function-application algorithms to apply the abstraction function \( A \) to the collected state. Other algorithms, such as those for generating the image from the objects and those which permit the user to control the visualization, must also be developed; these are not as important as the previous algorithms, as they do not represent new work.

A visualization is a function from the computation state to an image. Therefore, in order to construct the image, a visualization system must gather information about the computation state. As explained earlier, the shared-dataspace paradigm will be used, in which the computational state is represented by a set of tuples. The shared-dataspace paradigm represents a considerable step in the state-gathering process, as the state-gathering algorithm does not need to interact with the computational processes; this was one of the major reasons for using the paradigm.
However, the shared dataspace paradigm does not represent the entire solution. This is due to the fact that the dataspace may be maintained in a form resembling a distributed database. This introduces a number of problems in determining the global state. These problems have been previously addressed, both in the literature of concurrent processing\textsuperscript{29,48,93,157} and in the database literature\textsuperscript{26}. However, existing solutions are primarily directed toward obtaining a single global state (the snapshot and database rollback problems). Satisfactory visualization requires a succession of global states, each produced from the previous by only a very few changes to the dataspace; this is called the global state tracking problem.

Several possible solutions to the global state tracking problem have been suggested\textsuperscript{132,146}. These will be investigated with three goals in mind. First, it must be determined if the algorithms can can produce the desired fine granularity between successive states. Next, adaptations of the algorithms suitable for the shared dataspace paradigm will be developed. Finally, the complexity of the algorithms will be examined, including the probable effects of adding a collection algorithm to a computation.

Once a global state has been produced, the collection of graphical objects must be generated by application of the abstraction function. This function is represented in the form of rules which map tuples in the state space to tuples in the object space. This is, of course, the model of the function; an important issue is how best to implement this representation.

The abstraction function is used to translate the state into a collection of graphical objects; the rules used to perform this may involve quite complex queries. Thus, the primary issue for the abstraction function is to develop fast algorithms for performing queries over the collected global state. Some existing algorithms – for example, the RETE pattern-matching algorithm used in several expert-system shells\textsuperscript{53} or the more recent PARS or TREAT parallel pattern-match algorithms\textsuperscript{105,138} – may prove suitable for this processing, particularly since the collected states differ by only minor incremental changes. These algorithms will be investigated and adapted to the problem of applying the abstraction function rules to the collected state.

5.2. Visualization Methodology

Formulation of the visualization model will address a number of the shortcomings of existing systems. However, the model alone is not sufficient for constructing visualizations. A methodology is necessary to make effective
use of the model. Methodologies for constructing visualizations in existing systems are extremely ad-hoc and tend to be directed toward the specific problem instance. This is obviously to be avoided, and the question arises: What generally-applicable methods are suitable for guiding the construction of a visualization?

The purpose of this research is to investigate visualization as a tool for understanding concurrent computations. Understanding requires not just knowing what a computation does but why. This issue of understanding why also arises in the area of program correctness. The concept of program correctness was originally formulated for sequential algorithms\textsuperscript{14, 35, 42, 64, 104, 114} but has proved to be extremely useful in distributed parallel applications\textsuperscript{6, 9, 15, 33, 40, 73, 84, 89, 92, 115, 116, 137, 145}. Recent work has emphasized the use of program correctness in program derivation\textsuperscript{30, 99, 139}, where the desired program properties are first formulated and then the program is written based on these properties. Several researchers believe program derivation will become increasingly necessary for the development of large concurrent systems where the size and complexity of the problem overwhelm intuitive understanding.

The principal tools of program correctness are properties, formally-expressed statements about how the program behaves over time. Properties are generally divided into two categories: safety properties tell what the program must not do, while progress properties tell what the program will do. By formulating and proving such properties hold for the program, the programmer can demonstrate that the program behaves correctly. A number of systems for expressing such properties have been developed\textsuperscript{13, 30, 90, 98}. These systems are generally directed toward shared-variable models of computation. A proof system and programming logic suitable for shared dataspace languages is being developed by Cunningham\textsuperscript{37}.

It is my belief that the similarities between program correctness and visualization (when used as tools to aid understanding) can be exploited to develop a methodology for visualization. Specifically, the correctness properties – both safety and liveness – capture important aspects of the program behavior which should be represented in visual form. Further, the syntactic structure of the property (in whatever proof system is used) may serve as an indication of how the property should be treated. For example, the violation of safety properties should be visualized, while for liveness properties the fulfillment of the property should be captured. A simple example of this sort of visualization appears in a recent paper\textsuperscript{132}. 

The primary technical issue is therefore: Given a program property, how should it be represented in the visualization? The principal advantage of the use of formal program properties to guide the construction of the visualization is that this introduces discipline and order to an activity which is currently characterized by ad-hoc approaches. The formalism has another significant advantage; a large number of concurrent programs have been developed and proved correct, and this provides a useful base of programs and properties for testing the methodology.

A secondary technical issue deals with the layout of the visualization. This term refers to the basic form of the image, which obviously varies greatly depending on the underlying computation. Previous animation work includes layouts for sorting, searching, tree splaying, continued-fraction expansion, and region-labeling, among others. Although algorithms for automatically constructing some types of layouts exist\textsuperscript{12, 28, 123, 144, 160}, the variety of computations and the influence of human factors make it unlikely that any firm rules can be developed for designing layouts (indeed, ‘attractive’ layout of graphs is known to be NP-complete\textsuperscript{57, 150}). Because of this, I will not devote much time to this issue; the old ad-hoc methods of finding ‘nice’ layouts will be used.

6. RESEARCH PLAN

A number of tasks will have to be performed in order to conduct the research outlined in this proposal. Although all of these tasks contribute to the research effort, some are less directly involved in the main goal – development of a model and methodology – than others. However, many of these peripheral tasks are in themselves useful, as they involve the development of systems of general use in concurrent-processing research.

The principle tasks (arranged approximately in the order in which they will be performed – although some tasks will be performed concurrently with others) are:

(1) Developing a support system for shared dataspace systems. This system will support a distributed database of tuples, with tuple access by pattern match. The target hardware is the NCUBE Corporation NCUBE-71. The system design has already been developed, as described in a recent technical report\textsuperscript{26}, and the system is partially implemented.
(2) **Completing the development of the visualization model.** This task involves defining the types of visualization rules (both queries and object lists) that will be permitted. It also involves selecting an appropriate set of graphical objects and properties. Mallgren\textsuperscript{97}, McCleary\textsuperscript{101} and Tufte\textsuperscript{156} have suggested objects and properties which are appropriate for conveying information graphically, and their work will be used as a basis.

(3) **Completing development of the visualization methodology.** The methodology goal is to develop rules for constructing visualizations for computations. The general approach – use of program correctness properties – has already been identified. The specific heuristics which transform program correctness properties into graphical representations need to be developed.

(4) **Examining strategies for implementation of a shared-dataspace language fragment.** The primary purpose of the implementation will be to provide computations to be visualized; it is therefore not necessary that the implementation include the more complex types of shared-dataspace operations. A restricted subset of shared-dataspace operations, such as that outlined in a recent paper\textsuperscript{133}, can be used.

(5) **Developing algorithms for state-collection and state-tracking, and for producing objects using the visualization rules.** This work was discussed in section 5.1.3 of this proposal. The key criteria for these algorithms is that the collected information be fine-grained enough to support effective visualization.

(6) **Implementing a 'rendering function' package.** The rendering function transforms the graphical objects into image form. A package which generates an image on the target hardware, given as input the desired graphical objects with their properties, will be required so visualizations can actually be viewed.

(7) **Constructing visualizations of actual computations.** This task is obviously contingent on the previous tasks, and serves as the proof of the methodology. The computations used for visualization must be ones not previously examined. The methodology will be judged by how effectively the key properties of the algorithm are captured in the visualization.
7. REFERENCES


