UCRL-LR-119304

Dependency Visualization for Complex System Understanding

J. Allison Cory Smart
(Ph.D. Thesis)

September 1994

Lawrence Livermore National Laboratory

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

UCRL-LR-119304
Distribution Category UC-900

Dependency Visualization for Complex System Understanding

J. Allison Cory Smart
(Ph.D. Thesis)

September 1994

LAWRENCE LIVERMORE NATIONAL LABORATORY
University of California • Livermore, California • 94551

Dependency Visualization for Complex System Understanding

By

J. ALLISON CORY SMART

B.S. (Northwestern University) 1980

M.S. (University of California, Davis) 1986

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

1994

Dependency Visualization for Complex System Understanding

By

J. ALLISON CORY SMART

B.S. (Northwestern University) 1980

M.S. (University of California, Davis) 1986

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Rao Vemuri

Meera M Blattner

[illegible]

Committee in Charge

1994

To my wife

Barbara Anne Smart,

to my children

Jason Edward Smart and *Jessica Anne Smart*,

and to the warm and lasting memory of my parents,

Edward Kenneth Smart and Sophie Rosalie Smart

Acknowledgment

The author wishes to express his gratitude to the Computer and Communication Engineering Division of the Electronics Engineering Department, the Computations and Engineering Division of the Safeguards and Security Department, and the Fission Energy and Systems Safety Program at Lawrence Livermore National Laboratory for their support of this effort. Many of the examples presented in this paper were from actual software systems developed by these organizations. The author would also like to express his gratitude to Professor V. Vemuri of the University of California, Davis for his many long hours of coaching this effort, to Professor Meera Blattner of the University of California, Davis for her many excellent suggestions, and to Dr. Richard Watson of Lawrence Livermore National Laboratory for his insight into what a Doctor of Philosophy is all about.

Table of Contents

List of Figures

List of Tables

1. Introduction

In order to develop and maintain large scale, complex systems, it is essential that implementors and maintainers possess an adequate understanding of a system's design and organization. This understanding is necessary as it provides a framework for evaluating design decisions, guiding implementation alternatives, and isolating maintenance concerns. Achieving this understanding, however, has proven to be very difficult. Systems containing hundreds of interconnected modules, each with many thousands of components, are widespread. The complexity of these systems can often overwhelm even the most experienced engineers.

Of particular interest are software systems which now occupy a significant segment of the system engineering community and exhibit similar complexity characteristics. Software systems have grown considerably over the past decade. Systems with many thousands of modules and millions of lines of code (delivered executable machine instructions or programming language statements) are commonplace. The ability to comprehend these large systems remains one of the major challenges facing industry.

One of the most common factors contributing to the complexity of modern systems arises from the intricate connectivity structure associated with a system's many interdependent components. As the size of systems continue to grow, these structures are increasingly difficult to visualize. Without a suitable visualization mechanism, these structures become nearly incomprehensible, hindering the development and retention of any concise conceptual framework for the system's continuing design, implementation, and maintenance.

A method for exploring and untangling the complex dependency structures that appear in these systems is therefore needed. This method would be applicable to many modern systems engineering problems. While a general dependency visualization model is sought in this work, software dependency analysis will be used throughout this paper to illustrate the underlying concepts. Application to general systems analysis is a natural extension.

1.1 Software Dependency Visualization Background

Software visualization has a long history. Several graphical techniques for representing program flow control were developed early on [Tr'88]. As programs continued to increase in size, the importance of block structure became apparent; techniques for visualizing this structure quickly followed [Na'73, Be'80]. The concept of program visualization became well established by the mid 1980's [Kr'83, My'86, Re'85].

As the issues associated with flow control and block structure subsided, emphasis shifted to visualization of conceptual design, algorithm execution, and program behavior [Br'88, Re'87, Li'89]. With the advent of large software development projects, techniques and tools which focused on the architectural aspects of complex programs and the information they process emerged [Re'89, Ro'91].

With the volume of software in production use dramatically increasing, the importance of software maintenance has become strikingly apparent [Os'90]. Techniques are now sought and developed for reverse engineering [CC'90] and design extraction and recovery [Bi'89, Ru'89, Ri'90, CS'90]. At present, numerous commercial products and research tools exist which are capable of visualizing a variety of programming languages and software constructs [Om'90]. The list of new tools and services continues to grow rapidly.

Although the scope of the existing commercial and academic product set is quite broad, these tools still share a common underlying problem. The ability of each tool to visually organize object representations is increasingly impaired as the number of components and component dependencies within systems increases. Regardless of how objects are defined, complex "spaghetti" networks result in nearly all large system cases.

The needs for "untangling software" were presented in [SV'92, Sm'92] and popularized in [Pe'93].

While this problem is immediately apparent in modern systems analysis involving large software implementations, it is not new. As will be discussed in Chapter 2, related problems involving the theory of graphs were identified long ago [Tu'63]. This important theoretical foundation provides a useful vehicle for representing and analyzing complex system structures [Wa'74, Ve'78]. While the utility of directed graph based concepts in software tool design has been demonstrated in [Bi'91, BS'92, Ga'92], these tools still lack the capabilities necessary for large system comprehension. This foundation must therefore be expanded with new organizational and visualization constructs necessary to meet this challenge. This dissertation addresses this need by constructing a conceptual model and a set of methods for interactively exploring, organizing, and understanding the structure of complex software systems.

1.2 The Dependency Visualization Problem

Before adopting a specific solution strategy, it is first instructive to review the importance of dependency visualization and reveal the difficulties associated with this process. A series of simple examples will help illustrate these issues.

One of the most common visualization techniques used throughout software engineering involves the analysis of a program's dependencies. This analysis is a vital component in modern system understanding [Sc'93]. Dependencies are typically represented in graphical form using as a dependency diagram, a popular output format generated by many of today's software tools. Dependency diagrams are essentially directed graphs where each node in a graph represents some element or aspect of a system and an edge from one node to another signifies a dependency of the first element on the second. Dependency diagrams appear in many different forms including module structure charts, call sequence diagrams, entity-relationship diagrams, flow charts, finite state machines, menu trees, petri nets, etc. However, dependency diagrams carry semantic information that can not be captured in a purely graph theoretic model. The information associated with each node and the meaning implied by each edge can only be assigned and interpreted within the context of the system at hand. The goal of this work then is to construct a model for capturing and manipulating this dependency information in a manner conducive to visualization.

<u>Example 1.1</u> Consider a program with the following eight modules: MAIN, INPUT, COMPUTE, OUTPUT, SCALAR, VECTOR, MATRIX, and IO. In this simple program, suppose the module MAIN calls the input routine INPUT to read in the problem parameters, formats these parameters, and then passes the appropriate data to the routine COMPUTE where the desired computation is performed. Upon completion, MAIN then calls the module OUTPUT to display the result. Suppose that the module COMPUTE makes use of matrix, vector, and scalar operations via the modules MATRIX, VECTOR, and SCALAR, respectively. Suppose the modules INPUT and OUTPUT also require the use of a low-level input/output system module named IO.

If we are unfamiliar with the structure of this program and were tasked to reverse engineer a module call diagram, our first step would be to examine each module and tabulate its dependencies. A convenient data structure for capturing dependency information is the *adjacency list*, a list of all dependents for each element. That is, for each element v, we associate a list, $Adj(v)$ of all its successors.

Making no assumptions based on the name of a particular module, we would begin our examination, say, in alphabetical order as the modules appear in a directory or program library listing. A module is considered to be dependent upon another module if it references any of the other module's resources (e.g. via a **with** or **use** clause). The results of this examination are shown in Table 1.1. □

Table 1.1 Module Dependencies in Example 1.1

Module	Dependents
COMPUTE	MATRIX, SCALAR, VECTOR
INPUT	IO
IO	
MAIN	COMPUTE, INPUT, OUTPUT
MATRIX	SCALAR, VECTOR
OUTPUT	IO
SCALAR	
VECTOR	SCALAR

Using the information in Table 1.1, a directed graph representation of this example program can be easily constructed. A layout for this graph can be generated by simply positioning each module alphabetically in row-column format with each module represented as a circle and each module dependency represented as a line or arc from one node to the other. The resulting layout is shown in Figure 1.1.

Despite the fact that the program in Table 1.1 contains only eight modules, its organization is not readily apparent from Figure 1.1. The method used to position modules in a diagram obviously impacts design comprehension. Although easy to generate, the row-column graph layout approach is certainly not very effective in conveying useful organizational information about the program. Nevertheless, this simplistic approach often does reflect a software developer's/maintainer's initial view of an unfamiliar system.

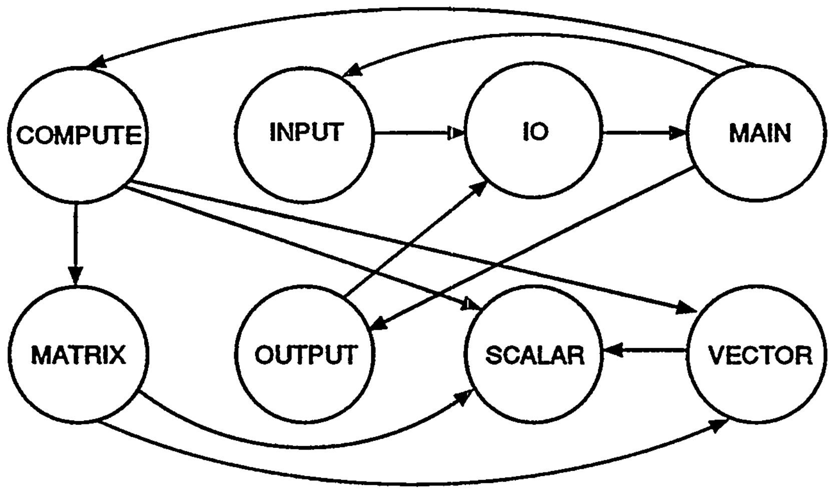


Figure 1.1 Default module layout of Example 1.1

<u>Example 1.2</u> Given the same program in Example 1.1, we will now proceed to generate a more suitable layout. First, certain modules such as operating system interface packages and low-level input/output routines can appear frequently throughout implementations. In many instances, the viewer will already be familiar with the purpose of these modules and their intended use. Components of this type will often be of little interest to the viewer when examining a new system since knowledge of the system's overall structure will generally receive higher priority. Under these assumptions, we may prudently choose to eliminate the module IO from view since its use is of little organizational consequence. The motivation and consequences of this and other similar types of decisions is described throughout Chapter 3.

Suppose we are also not interested in viewing transitive dependencies (i.e. a dependency of the form $A \rightarrow C$ is transitive if there exists a sequence of dependencies $A \rightarrow B_1$, $B_1 \rightarrow B_2$, ..., $B_{n-1} \rightarrow B_n$, and $B_n \rightarrow C$ for $n \geq 1$). This form of dependency appears frequently in systems and may often be considered redundant information. We can then choose to ignore the following dependencies: COMPUTE→SCALAR, COM-

PUTE→VECTOR, and MATRIX→VECTOR. This process known as filtering will be described with other similar functions in Chapter 4. Next, we can apply more appropriate layout criteria such as top-to-bottom layering of dependencies, left-to-right temporal ordering (i.e. a module which is executed before other modules appears to their left), graph symmetry, edge crossing minimization, etc. The definition of these criteria, for the time being, will be left unspecified, but will be thoroughly reviewed in Chapter 5. The resulting layout is given in Figure 1.2. □

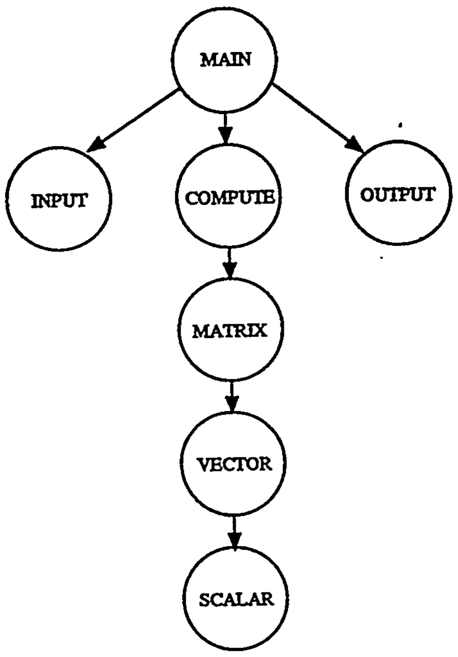


Figure 1.2 Improved module layout of dependencies in Table 1.1

Examining Figure 1.2, the structure of the program in Table 1.1 is now much more readily apparent. The three phases of the program's execution (input, computation, output) and the abstraction hierarchy of the computation (compute, matrix, vector, scalar) are easily identified. With only eight modules, the diagram in Figure 1.2 could have admittedly been generated by hand. It is fairly simple to deduce much of the architectural information we gained via Figure 1.2 from careful inspection of Figure 1.1.

Consider, however, a module dependency diagram for an actual program with over 250 modules [Sm'87] as shown in Figure 1.3. This figure was generated as before by placing each module, as it alphabetically appears in the program library, using the default row-column layout technique from Example 1.1. The dependency diagram in Figure 1.3 obviously conveys very little useful information as it neither captures the designer's original organizational intentions nor does it effectively reveal any of the system's important structural properties. Yet, it does accurately reflects a software maintainer's initial examination of the program's dependencies. After considering all the criteria that a viewer may wish to apply in visualizing an arbitrary dependency structure, the simple manual inspection techniques used to analyze Figure 1.1 are clearly no longer adequate. The need for an automated tool is readily apparent. Such a tool is developed in this dissertation and will be used in Chapter 8 to untangle Figure 1.3 and present a greatly simplified version.

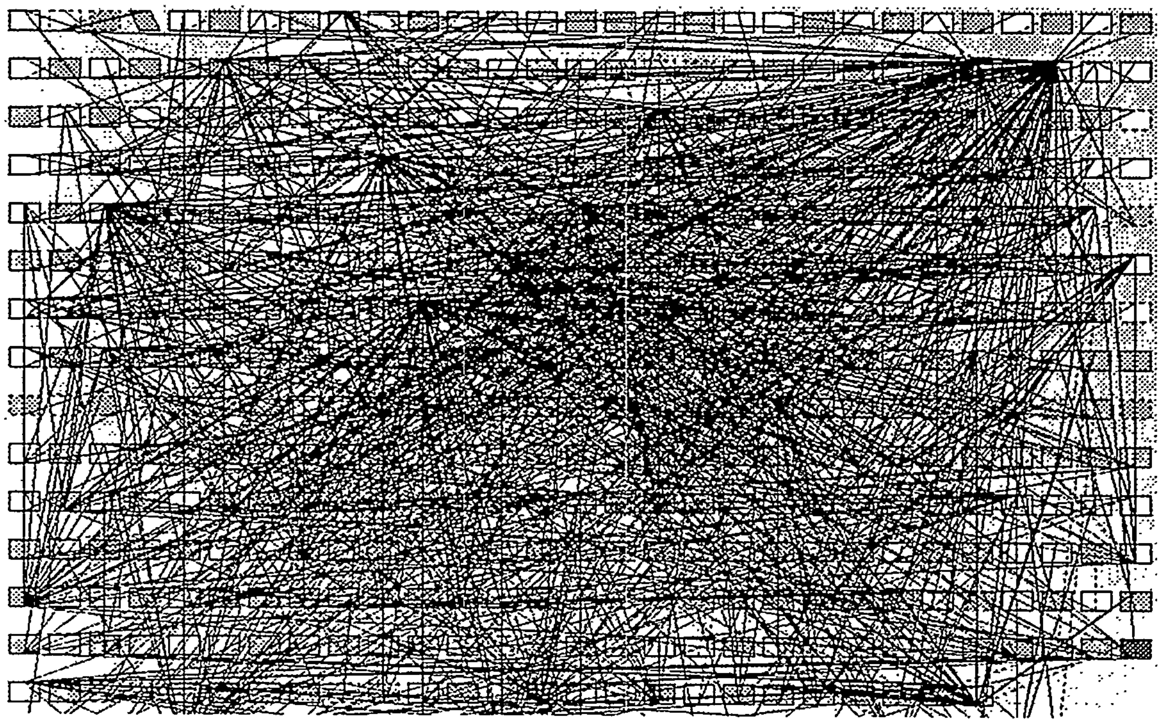


Figure 1.3 Unordered module dependency diagram

When generating layouts for exceptionally large (yet increasingly common) systems, determining which criteria to apply and in what order becomes exceedingly difficult as there are many alternative strategies on how to organize and view the same information. What information must our tools extract or deduce? How should relationships between elements be represented? How can we characterize the many different types of dependencies? And how once all this information has been assembled do we best present it to a user? The A-Vu model presented in this dissertation has been developed with these issues in mind.

While the example program above was tailored for this discussion, it does illustrate an additional important point. Given access to a visualization tool during the initial design phase of a new system, the developer of the system would have the ability to select an architecture which yields a simpler visual representation than otherwise possible. Hence, the system design process and the system visualization process would be closely coupled. The close relationship between design comprehension and visual complexity has, in fact, long been recognized [We'52]. Adaptation of this fact during system development may well be an important means of controlling system complexity in the future.

1.3 Research Objectives

The goal of this research is to develop a method for generating meaningful visual representations of complex system dependency structures (e.g. Figure 1.3). A transformation process which accepts input for an arbitrary dependency structure and outputs a meaningful visual representation in a suitable display format is envisioned. A block diagram of this process is shown in Figure 1.4. To adequately define the issues, seven fundamental questions listed below have been identified.

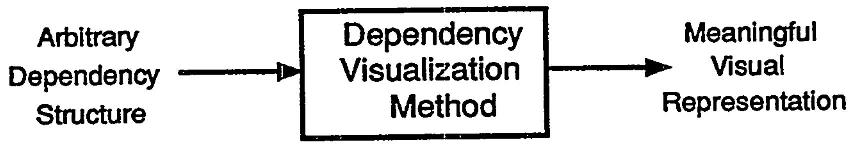

Arbitrary Dependency Structure
Dependency Visualization Method
Meaningful Visual Representation

Figure 1.4 Dependency Visualization Process

In addressing these issues, several contributions are made by this research. A model which effectively captures complex dependency structure information is defined. A catalog of useful operations for manipulating and viewing dependency structures caste in this model is developed. A suite of functions for evaluating the complexity of dependency structure arrangements is assembled. An optimization technique and an automated sequencing mechanism which integrates these items is developed. Finally, a prototype software tool for interactively exploring complex dependency structures using these techniques is described.

1). <u>How may complex dependency structures be specified and represented in a manner which facilitates visualization?</u> Traditional dependency display tools have few mechanisms for capturing domain specific information and hence, lack the expressive power to describe many system dependency characteristics. The development of a model which addresses these issues is necessary. This research presents a model which unifies dependency analysis and the interactive visualization process. This type of integrated approach to the problem has not previously been demonstrated. (See Chapter 2)

2). <u>Given an adequate dependency description model, what techniques for manipulating this model are required to achieve a desired visualization result?</u> Although a wealth of algorithmic knowledge can be applied in this area, the criteria and circumstances under which each of these algorithms should be applied has not previously been adequately established. This dissertation presents a comprehensive assessment and a catalog of effective algorithmic measures to fill this void. (See Chapter 3)

3). What specific visualization techniques can be applied within this framework to better promote system understanding? Traditional display tools portray dependency structures using simple directed graph depictions. Advancements in visualization technology, however, can be applied to further improve comprehension. This dissertation advances the graph layout field by incorporating several of these visual techniques including alternative rendering for nodes and edges, the application of three-dimensional perspective, the aggregation of subgraphs, and the use of visual filtering. This approach is unique in its ability to integrate all of these issues. (See Chapter 4)

4). How can the notion of a meaningful visual representation be defined? A quantitative measure of visual effectiveness has been lacking in the field. Most methods have relied on fixed criteria, subjective evaluations, and/or intuition. In order to automate this process, an objective evaluation method must be established. This dissertation addresses this problem by identifying a configurable collection of criteria that provides a precise definition of visual complexity. (See Chapter 5)

5). Given an effective evaluation measure, how can this entire process be optimized and subsequently automated for increased ease of use? Seeking a near optimal visual representation using only manual manipulation techniques is clearly inadequate for large systems. This dissertation addresses this issue by incorporating a powerful optimization technique in a manner that lets the user control the level of computational investment. The interactive optimization method employed is unique in the field. (See Chapter 6)

6). How can this entire process be applied in a manner that is both natural and flexible for the user? Current dependency visualization methods are relatively inflexible, enforcing a single focused paradigm with limited customization abilities. The majority of these tools are batch-oriented providing only limited user interaction. This work develops a powerful set of integrated techniques which allow a user to freely explore complex dependency structures from many different perspectives. The resulting software tool is unique in its ability to simultaneously address these many concerns in an interactive environment. (See Chapter 7)

7. Can the method adopted achieve all of these objectives, yet maintain an adequate level of performance? A performance assessment of this process is needed to validate its design and overall effectiveness. The results tabulated in this dissertation demonstrate how these original research objectives have been met. Greater insight into alternative node placement methods was also obtained. (See Chapter 8)

This dissertation works towards addressing each of the above questions as indicated. The results presented here provided an extensive formal framework for complex system understanding and dependency visualization discussion. This framework provides an important base for related dependency analysis research in the future.

1.4 Design Requirements

To focus this research effort, the development of a prototype software tool was conducted in unison with the development of the dependency visualization method. Throughout this dissertation, the tool design and the method development are treated synonymously. The specific requirements for the tool's design are identified in this section. A conceptual block diagram of the resulting method and its corresponding tool implementation is shown in Figure 1.5. This block diagram illustrates the integration of five basic components. The requirements pertaining to each of these components and their integration are itemized below. While not specifically appearing in the block diagram, performance requirements are also itemized below in response to Question 7 in Section 1.3.

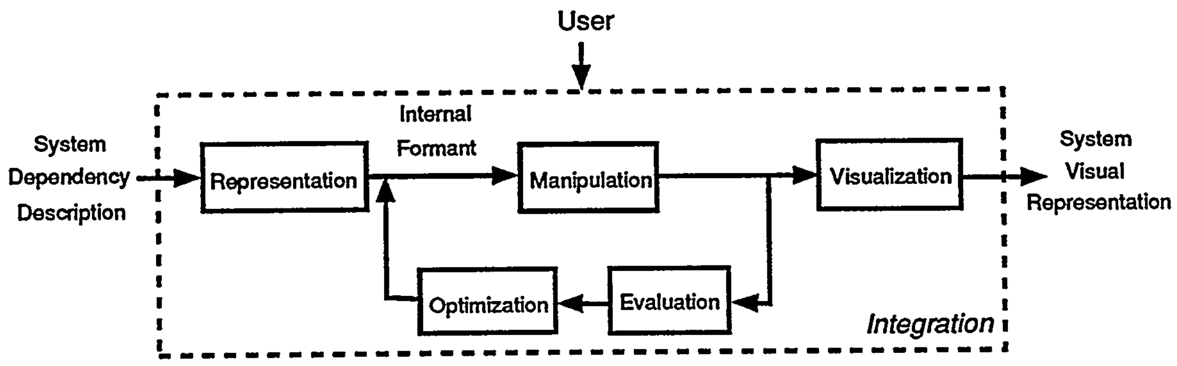


Figure 1.5 Conceptual Block Diagram

The design requirements itemized here are organized in the same manner as the research issues listed in Section 1.3 and are presented in the same order. The block diagram components shown in Figure 1.5 reflect this organization. The analysis and implementation of each of these items will be provided in respective chapters throughout this dissertation as outlined in Section 1.5.

1). Representation Requirements - The following requirements reflect the needs concerning specification and representation of dependency structure information for complex systems:

- The tool developed must be capable of representing large systems which may contain hundreds of tightly interdependent elements.
- A general purpose representation framework must be established that allows complex dependency structures from numerous system sources to be input. If necessary, a standardized input language and a series of translators may be introduced to achieve this requirement.
- Specifically, the prototype tool should be capable of extracting dependency information directly from existing software program libraries to verify the effectiveness of the method.
- The representation framework selected should be conducive to visualization. That is, a simple transformation should exist between the representation framework and a suitable display environment.
- In recognition of the fact that structural information alone is generally not sufficient for thorough system understanding, an abstraction mechanism for capturing domain specific knowledge associated with system elements and element dependencies should be included in the design.

- The issues associated with extraction and representation of system dependency information should be identified, cataloged, and addressed in the tool design.

2). <u>Manipulation Requirements</u> - The design of this tool should address the following dependency structure manipulation requirements:

- The tool should provide a set of operations for manually editing dependency structures including the ability to copy, cut, and paste elements.
- The tool should automatically manage and update element dependencies as these operations are executed.
- The tool should allow users to automatically generate layout forms of frequent interest. In achieving this requirement, a catalog of useful layout operations and a survey of the criteria leading to their formulation should be compiled.
- The tool should enable a user to examine subsets of a dependency structure and ignore those portions the user deems to be of little interest.
- The tool should allow users to create composite elements by collapsing selected portions of dependency structures as a means of reducing and controlling visual complexity.

3). <u>Visualization Requirements</u> - The following requirements describe how dependency information should be portrayed:

- The tool should provide a method for displaying dependency structures on a suitable graphical display screen. This display should be maintained and immediately updated upon completion of each user operation.

- The tool design should enable a user to examine complex dependency structures from many perspectives. A survey of the types of perspectives that are considered most useful should be completed in conjunction with this requirement.
- A method for exploring nested dependency structures should be provided.
- A method for filtering undesirable or unnecessary dependency information from view should be provided.
- A survey of parameters that allow users to view complex dependency information in this manner should be be identified and characterized in conjunction with this effort.

4). Evaluation Requirements - The requirements listed here describe the needs concerning quantitative assessment of dependency structure visual representations:

- The user should have access to a suite of functions or metrics for evaluating the complexity of the visual representations that are generated. A compilation of these functions and a characterization of their utility and performance should be included in this effort.
- The evaluation techniques available to the user should consider both structural complexity (i.e. directed graph measurements) and semantic complexity (i.e. domain specific measurements).
- A technique for combining evaluation techniques to synthesis new metrics should be provided.
- A method for adjusting the relative importance of individual evaluation components should also be included.

5). Optimization Requirements - The following requirements pertain to the automation of the visual representation generation process:

- An automated means of finding an "optimal" visual representation of a complex dependency structure should be provided.
- The method should allow the seamless integration of the evaluation techniques previously described.
- The user should have the ability to place various constraints on the optimization to limit the search space and improve performance. A compilation of effective constraints and their relative merits should be generated as part of this effort.
- The user should have the ability to stop, interrupt, modify, and continue optimization operations as desired to control the desired level of computational investment. The best interim result obtained during optimization operations should be returned as the final value.
- A method of combining sequences of manipulation, visualization, and optimization operations such that they can be archived for later use should be provided.

6). Integration Requirements - These following requirements pertain to the integration and implementation of all the above techniques:

- This tool should allow a user to freely browse and explore dependency structures in order to gain a thorough understanding of a system's organization. Consequently, this tool should be interactive rather than batch-oriented in nature.
- This tool should be implemented in conformance with an established representation model to help guide its users in exploring complex dependency structures.

- The user should be capable of executing operations at their own discretion or via an automated sequence of their own creation. All of the operations identified in this work should be accessible in either manner.
- The tool should be implemented using standardized windowing and user interface techniques (i.e. X-Windows/Motif).
- Typical housekeeping functions associated with user tools of this type should be included such as window manipulation, scrolling functions, file access and results archival, performance profiling, etc.

7). <u>Performance Requirements</u> - To meet interactive performance standards, the following requirements are established:

- The algorithmic complexity of all operations should be less than or equal to $\Theta(n^2)$ or $\Theta(k^2)$ where n is the number of system elements and k is the number system element dependencies being represented.
- An analysis of each of operation and evaluation measure defined in this tool should be performed to verify the above performance constraints.
- The tradeoff between run-time performance and solution optimality should be under the control of the user.

A thorough description of each of the components shown in Figure 1.5 along with a survey of the many issues associated with their implementation, integration, and execution have been compiled in this dissertation. The requirements identified above served as a guide for the development of these components. The aggregation of information associated with each of these components provides an important template for future complex dependency analysis research and product development.

1.5 The A-Vu Strategy

In order to meet the growing demands of complex system visualization in accordance with the above requirements, a solution strategy must first be adopted. Within this dissertation, this strategy is referred to as A-Vu, a term originally selected as an acronym for "Abstract Visualization utility" and derived from the compound past form of the French verb *voir*, meaning "one has seen." Alternatively stated, this strategy provides a means for generating "A View" of large, complex dependency structures, but a view that is both comprehensible and meaningful.

The A-Vu strategy is comprised of seven distinct research and development tasks or phases. Each of these tasks correspond to one of the research questions that were posed in Section 1.3 and their respective requirements outlined in Section 1.4. A summary of each task is provided below and is presented in the same order. This summary serves as an outline for the remainder of the dissertation.

1). Model Definition: A unified model is developed for representing complex dependency structures and supporting the operations that must be performed on these structures. The primary concepts presented in this model include the notion of a *dependency graph*, a *visualization space*, a *layout*, a *binding*, and a *configuration*. Configurations are composites of the previous concepts and are the primary focus of the dissertation. (Chapter 2)

2). Configuration Manipulation: With the establishment of the configuration concept, techniques for manipulating these structures are developed next. The ability to perform editing operations, search operations, and selection functions are established as a foundation for all subsequent configuration processing algorithms. With these configuration manipulation mechanisms in place, algorithms for generating configurations with specific properties are then developed. These algorithms make use of the wealth of graph theoretic knowledge applicable to the structure underlying the configuration concept. (Chapter 3)

3). <u>Configuration Visualization</u>: The ability to organize and manipulate complex dependency structures then leads to the development of operations for viewing resulting configurations. Methods for displaying, reducing, and/or consolidating the quantity of visual information that must be conveyed to a user are developed. This process involves the establishment of numerous display primitives, several basic view transformations, a composite space navigation technique, and a series of dependency information filtering operations. (Chapter 4)

4). <u>Configuration Evaluation</u>: In order to automatically generate configurations that are useful (i.e. readily convey meaningful information in an efficient manner), a quantitative measure for this process is established. Evaluation of complex dependency structures, however, involves numerous criteria. This criteria is often subjective and may frequently conflict. An objective measure that reflects these concerns is constructed. (Chapter 5)

5). <u>Configuration Optimization</u>: With a quantitative evaluation technique defined, it is then possible to use this measure as a guide for automatic configuration generation. A strategy for generating optimal or near-optimal configurations is presented. A variety of analysis techniques are then available. A mechanism for automatically applying these techniques in the desired order is developed next. The concept of an automated configuration *sequence* is defined which allows a set of operations to be generated, saved and re-applied to any arbitrary configuration. (Chapter 6)

6). <u>Configuration Integration</u>: The last development task involves the integration of all of these techniques into a single tool which can be used for interactive system dependency analysis. The design and implementation of the A-Vu prototype system is presented. A comparison is made between the A-Vu system and the dependency analysis methods presented earlier in the dissertation. (Chapter 7)

7) Configuration Performance: With all these tools in place, a performance analysis of the various aspects of the A-Vu model and its associated manipulation operations, evaluation functions, and optimization techniques is then be performed. Several test samples taken from actual software development efforts are used to validate the process and illustrate its effectiveness. (Chapter 8)

At the completion of these seven task descriptions, the dissertation concludes with a summary of the implications of this work, a review of how this work evolved to meet its research objectives, and a discussion of the many potential areas for further exploration and tool implementation. (Chapter 9)

2. The A-Vu Model

"Any program is a model of a model within a theory of a model of an abstraction of some portion of the world or of some universe of discourse." [Le'80]

The initial step in the A-Vu strategy is to construct a unified model for effectively capturing complex system dependency information. This model is developed in this chapter in several stages. An overview of the model is presented in Section 2.1. Starting in Section 2.2, a formal foundation is established using the theory of directed graphs. A visualization mechanism is built upon this foundation in Section 2.3 and Section 2.4 by incorporating the necessary constructs to give visual meaning to the notion of a system dependency graph. Next, the ability to capture application domain specific semantic information is added to the model as described in Section 2.6 through Section 2.9. A method for describing the properties of systems within this framework is then discussed in Section 2.10. An important enhancement is made in Section 2.11 to capture nested organizational constructs. A discussion concerning the recursive nature of resulting dependency structures is presented in Section 2.12. Finally, several additional enhancements are made to the model in Section 2.13 in anticipation and support of the tools to be developed in later chapters.

2.1 A-Vu Model Overview

The central idea of the A-Vu model is to determine an optimal placement of system elements in a multi-level set of two-dimensional diagrams which best promotes design comprehension and system understanding. This model is developed below beginning with a *directed graph* representation of system dependencies. Each system element is represented as a node in a graph and each dependency as an edge. A *layout* is associated with the graph by defining positions of each of its nodes. The layout concept is refined by introducing the notion of a *visualization space*, a set of vectors from which each node is assigned a value. A graph together with its corresponding layout constitute a system *configuration.*

The next step is to incorporate semantic information into the configuration definition. This is performed by associating a set of attributes with each node and each edge in the graph. Finally, the concept of a *composite* configuration is introduced to address the need for nested layouts. The resulting definition forms the basis of a new complex graph layout strategy. Near-optimal layouts are obtained via the integration of a unified, multiple-step optimization process described below in Chapter 6.

Conventional graph layout approaches typically incorporate fixed layout criteria [Ea'90, Ga'93] such as:

1) edges in same direction
2) even distribution of nodes
3) edge crossings minimized
4) arcs straight as possible

The A-Vu approach, however, incorporates the use of multiple, user-selectable criteria based not only on graph node-edge structure, but also on semantic information extracted from the system. Conventional graph layout also typically consists of four basic steps [Ea'90]:

1) Make directed graph acyclic
2) Layer acyclic diagram
3) Order nodes in each layer
4) Position nodes

In contrast, the A-Vu method employs an interactive approach to complex system visualization. This method allows users to examine and explore dependency structures from many different perspectives. This interaction is similar to the graph browsing and editing concepts described in [He'87, Pa'90, Ro'87], but incorporates numerous enhancements for dealing with large systems. With access to many layout algorithms, integrated evaluation techniques, and automated sequencing options, the user has direct control over the layout process. Several of the visual aspects of this approach have been motivated by [Bu'84, Ha'88].

2.2 Directed Graph Representations

The theory of directed graphs is a useful vehicle for representing the structure of complex systems. A *finite graph* $G = (V, E)$ is defined as a finite set of vertices $V = \{v_1, v_2, v_3, ..., v_n\}$ and a finite set of edges $E = \{e_1, e_2, e_3, ...; e_k\}$, with $E \subseteq V \times V$. Here $n = |V|$ denotes the number of vertices and $k = |E|$ the number of edges. If the vertex pair (v, w) associated with an edge e is an *ordered pair*, then G is a *directed* graph.

Using this notation and terminology, software systems, from a variety of perspectives, can be defined in terms of vertices (or nodes) representing software elements and directed edges representing element dependencies. The problem of understanding the complex dependency structure of a software system with a set of modules $M = \{m_1, m_2, m_3, ... , m_n\}$ with dependencies $D \subseteq M \times M$, can therefore be equated to the layout and visualization of a dependency graph or its equivalent directed graph.

One may wish to first perform a variety of graph theoretic operations [Ta'89], such as, (a) subdivide the graph to reveal the layered structure of software products, (b) subdivide the graph into its *strongly connected components* to reveal closely interdependent elements for possible aggregation, (c) subdivide the graph by separating it at its *articu-*

lation points, (d) check the *planarity* of a graph and subdivide it into planar subgraphs, (e) check the isomorphism of two graphs, and so on. One may also wish to perform more pragmatic operations such as (a) identify common interconnection patterns like source-to-sink paths, (b) feedback loops, (c) closed paths or *cliques*, (d) hub-like connections, and so on.

Our task then is to perform certain operations on G and display the result. One of the first difficulties we encounter is in determining which type of graph operations aid comprehension. While there exists a wealth of graph theoretic knowledge, the time complexity of many graph theoretic algorithms is also notoriously sensitive to the manner in which the problem is stated; even a minor modification in the statement of the problem can lead to a significant difference in the complexity metric. For example, one way of facilitating the display of a complex graph is to partition the graph into planar subgraphs. This simplistic approach is likely to be counterproductive for two reasons. First, given a graph $G = (V, E)$, the problem of partitioning E into $E_1, E_2, ..., E_k$ for $k \geq 1$ such that $G_i = (V, E_i)$; $1 \leq i \leq k$ is planar is an NP-hard problem if we are interested in finding the smallest value of k. Second, while a planar decomposition is of theoretical interest, it is of little practical use in understanding a software system's organization. This type of decomposition would reflect only the graph theoretic properties of the system and not the designer's original intent.

In order to exploit the expressive power of a graph in its visual representation, it is necessary to go beyond strictly graph theoretic considerations. Node type, shape, size, and placement in a diagram are frequently used to convey design information. For example, a software system generally possesses some degree of inherent hierarchy; that is, certain modules are expected to appear at certain positions relative to other modules. Sometimes these relationships are best understood if they are laid out in the shape of a tree structure. At other times, the purpose is better served if the intermodule dependencies are shown with a parent module at a central hub and all children laid out evenly around the perimeter of the hub. Sometimes, the relationships between modules are best understood if any inherent functional or structural symmetry in the problem is revealed. It may also be instructive to associate the physical size of the icon on the screen with some measure of the complexity of the module, or perhaps to associate a meaning to the position occupied by a module (or its icon) in the screen coordinates. Many of these issues will be addressed in the following sections.

2.3 Layouts

While directed graphs are a useful vehicle for representing system dependencies, there is no visualization mechanism inherent in their definition. The graph is simply an abstraction for modeling the system's dependencies. Graph layout, however, is a common process. In its simple form, the layout for a graph can be described as a set of locations for each node in the graph and a set of line segments or curves connecting the nodes. In a planar layout, node positions would be restricted to a simple (x, y) coordinate system; edges would be restricted to a sequence of (x, y) coordinates.

Under the simple planar scenario, a layout L can be initially defined as the two-tuple $L = (P, Q)$ with $P = \{p_1, p_2, p_3, ..., p_n\}$ where p_i is the (x, y) coordinate for node v_i, and $Q = \{q_1, q_2, q_3, ..., q_k\}$ where q_j is the set of (x, y) coordinates defining the edge e_j. The exact form of the set q_j is not defined here, but could be a pair of end points that define a line segment connecting the two nodes, a sequence of line segment end points, the control points of a spline function, etc. If we desire to perform multi-planar layout (eg. [Ch'79]), we could modify the definition of L slightly, allowing coordinates of the form (x, y, z) with z indicating the plane of a particular node or edge. This definition seems to suffice for simple planar-type layouts, but is too restrictive as there is no mechanism for capturing nested graphs, edges which span planes, semantic information, etc. We will therefore continue to refine this definition.

2.4 Visualization spaces

When we generate a graph layout for a system, we may wish to require that the elements lie (a) within the boundaries of some abstract design space, (b) at discrete points within the design space, predefined by row and column positions, (c) anywhere in the design space such that their iconic representations on the screen do not overlap, etc. To meet these needs, we continue by generalizing the layout concept.

We associate with each graph G, a space S and call it the *visualization space* of G. For each element $v_i \in V$ we assign a unique vector $p_i \in S$ which denotes the position of node v_i in S and for each each edge $e_j \in E$ we assign sets of vectors $q_1, q_2, q_3, \ldots, q_k$ with $q_k \subseteq S$, which denote the positions describing edge e_j in S. The set $P = \{p_1, p_2, p_3, \ldots, p_n)$ and the set $Q = \{q_1, q_2, q_3, \ldots, q_k)$ contains the position vectors for every node v_i and edge e_j in V and E, respectively, as before. However, let a layout L now be defined as the three-tuple $L = (S, P, Q)$. The tuple L describes how G is embedded in S. The introduction of a visualization space enables us to examine dependency structures in a variety of new ways, breaking away from strictly planar organizations.

The space S can be continuous, discrete or hybrid. The continuous case is the most general as it allows nodes to be positioned at any point within the design space boundaries. Continuous spaces are required when the geometric properties of the system diagram precludes discrete point assignment such as with star network topologies requiring rotational symmetry enforcement.

While a continuous space is most flexible, it can often be approximated with a discrete space where position is identified by a grid location. With discrete spaces, system elements, depending upon their complexity, occupy one more of the cells created by the grid. Each cell is identified by its row/column location. Position selection with discrete spaces is simpler as there are no longer an infinite set of positions that can be assigned to system element vertices. While the use of a discrete layout can considerably limit the choice of vertex positions, it offers improved computational efficiency. Discrete layout visualization is appropriate for applications which exhibit a great deal of regularity such as switching circuitry, printed circuit board layout, and city street networks.

Using the so-called hybrid scheme, each node must be located within a specific (discrete numbered) plane in the visualization space. Within a particular plane, however, a node can be assigned any real value position. This scheme is particularly suitable if geometrical symmetry of system element location is a consideration, but the elements themselves can be grouped into a discrete number of subsystems. All three schemes appear suitable for software system visualization. Figure 2.1 captures the essence of the three schemes.

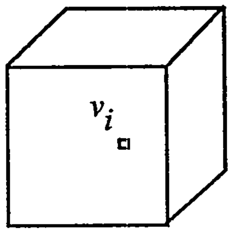


Figure 2.1(a) Continuous space

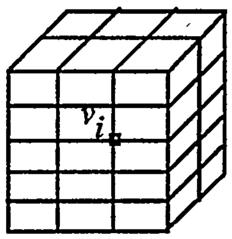


Figure 2.1(b) Discrete space

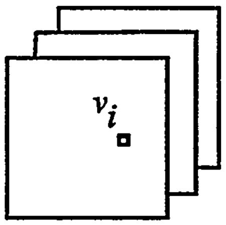


Figure 2.1(c) Hybrid space

Figure 2.1 Three possible types of visualization spaces

2.5 Configurations

Given a software dependency structure with a graph G and a layout L, we can define a *configuration* $C = (G, L)$ for the structure. In this simple form, a configuration dictates how a particular diagram for the structure is to be drawn by specifying the position of each node.. For simplicity, we again assume all edges are represented by a sequence of linear segments or control points and that they can be readily determined knowing only the positions of the nodes at each end. This is a reasonable assumption based on [Ta'88, Ea'90]. Generalized arc and spline presentations have been addressed in [Ga'93].

Note that the above definition of a configuration is based solely on a system's graph description (i.e. its node/edge relationships). This definition would suffice if we were only interested in performing traditional graph layout. In order to more thoroughly understand the organization of a complex system, we must go beyond basic graph structure and examine the meaning associated with each node and edge. Our model must be capable of capturing traditional software engineering practices such as modularity, layering, information hiding, and so on. Other constraints addressing perceptual groupings and additional aesthetic concerns must also be considered.

We address these issues by extending our basic configuration definition. By assigning various attributes to the nodes and edges of our software dependency diagrams, we can capture much of the semantic information that is not accessible in a purely graph theoretic framework. Recall our previous definition of a configuration, $C = (G, L)$. We proceed by modifying this definition as $C = (G, L, A)$ where $A = (A_v, A_e)$. A_v contains the attribute information for all of the nodes associated with the system and A_e contains the attribute information for all of the edges associated with the system. The definitions for A_v and A_e are expanded below.

2.6 Node Attributes

Let ς be a set containing all possible node attribute values. Let $A_v = \{\varpi_1, \varpi_2, \varpi_3, \ldots, \varpi_n\}$ where $\varpi_i \subseteq \varsigma$ and $n = |V|$. The element ϖ_i of A_v is a set containing those attributes which are associated with node i. This set is used to capture the semantic information associated with the directed graph nodes. The A-Vu model predefines several node attribute values. A node can possess zero or more of these attributes. The following is a list of those attributes (i.e. members of the set ς), their meanings, and a graphical representation:

Universal

Universal nodes are the most fundamental node form, representing system elements which possess no explicit semantic information. A node is assumed to possess the *universal* attribute if no other attributes have been specified. This attribute is provided for representation and manipulation of general-purpose directed graphs. Universal nodes are graphically depicted using a circle.

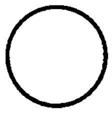


Figure 2.2(a) Universal

Procedural

Procedural nodes represent system elements which specify a sequence of actions or steps to be performed. Information is passed to procedural nodes using set of parameters. Likewise, information can be returned from procedural nodes via these parameters. Examples in software systems are procedures and subroutines. Procedural nodes are graphically depicted as a box with a single notched corner.

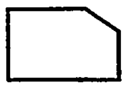


Figure 2.2(b) Procedural

Functional

Functional nodes represent system elements which also specify a sequence of actions to be performed, but which return only a single value (or set of values) as a result of their execution. Functional elements typically refer to the function construct in common programming languages such as C, Ada, and Pascal. They are depicted as a double notched box.

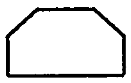


Figure 2.2(c) Functional

Parallel

Parallel nodes represent system elements which operate concurrently with other portions of the system. Tasks and processes are typical examples. Parallel nodes are portrayed using a parallelogram.

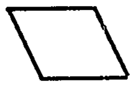


Figure 2.2(d) Parallel

Aggregate

Aggregate nodes represent elements which are collections of logically related entities such as groups of type definitions, objects of these types, and procedures and functions with parameters of these types. The Ada and Modula-II package construct is an example. Aggregate nodes are depicted as a simple box.

Figure 2.2(e) Aggregate

Standard

Standard nodes represent system elements which are predefined or are included by default in the system environment. Examples include operating system interfaces specifications, IO packages, general math routines, etc. The *standard* attribute can be applied to nodes which possess any other attribute. This attribute is depicted using double thick lines. A standard, procedural node, for example, is shown as follows:

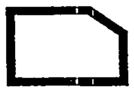


Figure 2.2(f) Standard

Generic

Generic nodes represent parameterized elements which are software templates for elements such as structural, procedural, or functional elements. The generic compilation unit construct in the Ada programming language is an example. The *generic* attribute is depicted using dashed lines. For example, a node which possesses both the *procedural* and *generic* attributes node is shown below:

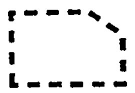


Figure 2.2(g) Generic

Instantiation

Instantiation nodes represent elements which are instances of a particular generic element (i.e. a generic instantiation). The instantiation is distinguished as a dashed lined figure within another figure. An instantiation of a generic procedural node, for example, is shown as follows:

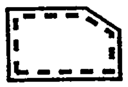


Figure 2.2(h) Instantiation

Specification

Specification nodes represent elements which describe the interface to an implementation element, but do not provide details on how the element is internally organized. If unspecified, all nodes are treated as specification nodes unless they contain the *implementation* attribute. The *specification* attribute is depicted as a clear figure. Each of the figures (a) - (h) drawn above represent nodes with the *specification* attribute. The following figure represents a node with both the *procedural* and *specification* attribute.

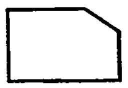


Figure 2.2(i) Specification

Implementation

Implementation nodes represent elements which contain items which perform actual operations within the system. The *implementation* attribute is identified using as a figure with a gray background. The following is a functional, implementation node:

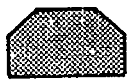


Figure 2.2(j) Implementation

Foreign

Foreign nodes represent elements which are components of external systems. Foreign nodes are typically limited to implementation nodes as the details of the foreign system are generally completely hidden. For this reason they are represented as an opaque figure. The following is a foreign, aggregate node:

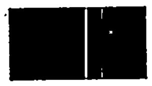


Figure 2.2(k) Foreign

Composite

Composite nodes represent system elements which contain other system elements. Composite nodes are used to group related nodes into a single node in order to reduce the complexity of a particular configuration. The organization of composite nodes will be described below in Section 2.11. The *composite* attribute is depicted by a three-dimensional figure. The following is a representation of a composite, aggregate node:

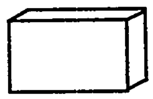


Figure 2.2(l) Composite

Figure 2.2 Node attribute representations

Visible

The *visible* attribute is a dynamic attribute that can be assigned to any node. This attribute is used by the visualization system for node filtering. In order for a node to be visible, it must be properly contained within a visualization space. Visible nodes are depicted as described above depending upon the node's other attributes. Nodes without the *visible* attribute are not displayed. For instance, all nodes except IO in Example 1.2, possess the *visible* attribute.

In summary, the discrete set ς at this time is defined to be ς = { *universal, procedural, functional, parallel, aggregate, generic, instantiation, specification, implementation, composite, visible*}. For notational convenience, we define a series of boolean functions $f_{\varsigma}(v)$ which will allow us to test whether or not a particular node possesses a specific attribute from ς. For each element α in ς, we define a function $f_{\alpha}(v)$ which when passed a node $v \in V$, returns true if v possess the attribute ϖ and false otherwise. If we let v be the element represented in Figure 2.2(l), the following boolean expressions apply:

UNIVERSAL(v) = FALSE;
PROCEDURAL(v) = FALSE;
FUNCTIONAL(v) = FALSE;
AGGREGATE(v) = TRUE;
GENERIC(v) = FALSE;
INSTANTIATION(v) = FALSE;
SPECIFICATION(v) = TRUE;
IMPLEMENTATION(v) = FALSE;
COMPOSITE(v) = TRUE;
VISIBLE(v) = TRUE;

Example 2.1 Returning to our program previously defined in Example 1.1 and Example 1.2, suppose we are able to extract the following information:

- COMPUTE is subroutine
- INPUT is a system library subroutine
- IO is a low-level interface package
- MAIN is a main subroutine
- MATRIX is a user library package
- OUTPUT is a system subprogram
- SCALAR is a user library package
- VECTOR is a user library package

The attributes for this system are shown in Table 2.1. Applying the same layout technique used in Example 1.2, the resulting diagram is shown in Figure 2.3. □

Table 2.1 Example Program Node Attributes

Module	Attributes
COMPUTE	*procedural, visible*
INPUT	*procedural, standard, visible*
IO	*aggregate, foreign*
MAIN	*procedural, visible*
MATRIX	*aggregate,instantiation, visible*
OUTPUT	*procedural, standard, visible*
SCALAR	*aggregate, instantiation, visible*
VECTOR	*aggregate, instantiation, visible*

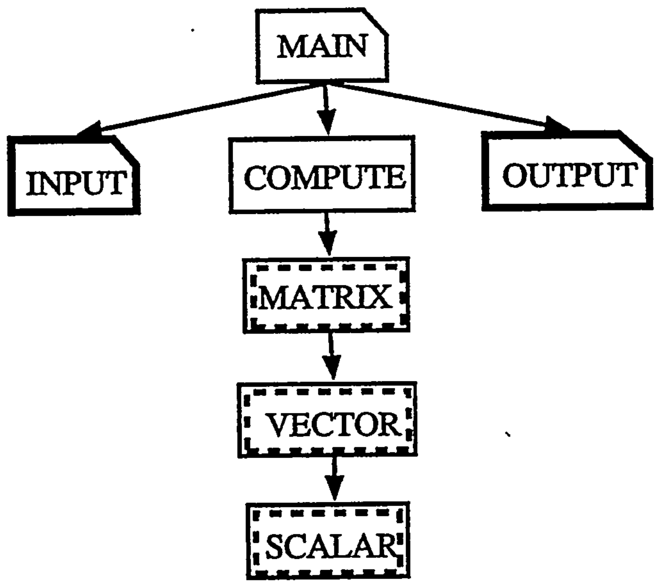


Figure 2.3 Improved layout with node attributes

2.7 Auxiliary Node Information

In addition to items listed in the previous section, there are numerous other characteristics of a software system element that may need to be captured as node attribute information. A brief compilation and summary of these items are listed here.

Name

Perhaps one of the most fundamental areas of system design, the naming process provides a mechanism for designating and referencing system elements. A name usually consists of a symbol such as an alphanumeric string which may be used to uniquely identify an element within an appropriate context. The issues associated with naming are numerous [Sa78]. In addition to their identification purpose, names can also convey useful semantic information. In each of the three previous examples, the name of each module was used to help derive a comprehensive layout. While naming was quite useful in these exercises, a number of difficulties are introduced.

Most programming environments pose few restrictions on name selection. Hence it is up to the designers and implementors of the system to select names wisely. Systems also evolve considerably as they move through their life cycle. Consequently, names can be misleading. In some environments, names may be computer generated, carrying little if any useful semantic information.

With an appropriate design standard or programming discipline in place, however, a consistent naming convention can be invaluable in structural analysis and system understanding. In response to this concern, a name pattern matching operation will be integrated into the A-Vu strategy, described in Section 3.3.

Location

Often related to the name of the system element associated with each node is the location of the system element itself. In software systems this typically relates to a source code file name specification such as a pathname or directory designation, but could alternatively be a reference to a particular page in a document, a set of paragraphs, a range of program line numbers, a catalog index, etc. The A-Vu implementation described in Chapter 7 treats this attribute simply as a character string which can be displayed and passed to other tools as necessary.

Characteristics

Associated with every system element are a number of implementation characteristics that may be of some utility within a visualization environment. These characteristics may include attributes such as the author, owner, creation date, version, programming language type, compilation date, source code size, binary image size, exchange format, library index, etc. These items currently remain undefined in the A-Vu model, but could be easily added as necessary. Visual techniques such as color, shading, size, and even animation and sound [Bl'8x] could be used to convey these characteristics.

Contents

Associated with every node in our model is the actual contents of the system element it represents. In software systems, this typically will be the module source code or the programming language statements that implement the module. In emerging software environments, however, the contents of a module may actually be much broader in scope and could include additional items such as functional requirements, functional specifications, formal descriptions, source code annotation, revision history, bibliography, test plans, and associated user documentation.

Each of the above items could in turn be broken down into numerous other attributes that could be accessed, manipulated, and graphically portrayed by the visualization system. Both the academic area and commercial marketplace, however, are flooded with tools for creating, viewing, and modifying these items. Our discussion, therefore, assumes that the contents of a node are not directly visible but are represented by a descriptor that is sufficiently general to allow the A-Vu system to activate an appropriate tool for viewing and modifying these items. For example, using the *name* attribute, the l*ocation* attribute, and a content descriptor indicating that the associated system element is a source code module, the A-Vu system could pass the *name* and *location* attribute values to a typical text editing tool. Similarly, if the contents of a node represented a particular database relation, the name of the relation and the location of the database that contains it could be passed to an appropriate forms display or spread sheet manipulation program.

Since it is not unlikely that a system element may have a variety of content types in addition to multiple names, locations, and implementation parameters that describe these contents, each of the attributes outlined in this section should be considered as an attribute set or category. Advanced implementations would incorporate an appropriate data structure for representing these various categories. For example, the *contents* attribute would most likely be represented as a list of content descriptors. The precise details of this implementation, however, are beyond the scope of this discussion.

Numerous dynamic attributes for nodes could also be considered such as an indication of the node's current execution state, its resource utilization history, a measure of its run-time performance, and a summary of its computational or source code complexity.

2.8 Edge Attributes

Just as attrubutes were assigned to graph nodes, attributes can also be assigned to graph edges. Let $A_e = \{\varepsilon_1, \varepsilon_2, \varepsilon_3, \ldots, \varepsilon_k\}$ where $\varepsilon_i \subseteq \delta$ and $k = |E|$. The set δ contains all possible edge attribute values; the set ε_i contains those attributes which are associated with a specific edge. The A-Vu model predefines several edge attributes values. A edge can possess zero or more attributes. The following is a list of those attributes (i.e. members of the set δ), their meanings, and graphical representation:

Universal

Universal edges are the most fundamental edge form, representing system element dependencies which possess no explicit semantic information. The *universal* attribute is provided for representation and manipulation of general-purpose directed graphs. A universal edge is the most common form of edge attributes and signifies a direct reference between two system elements such as a use or with statement in common programming languages. An edge is assumed to possess the *universal* attribute if no other attributes have been specified.

Universal edges are graphically depicted using the traditional directed arc or arrow. The following diagram represents two universal nodes with a single universal edge:

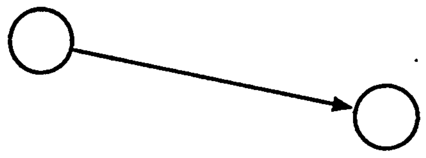


Figure 2.4(a) Universal

Implied

An implied edge represents an assumed, tightly coupled dependency between two elements in a system. This attribute is used to represent a dependency between two elements where the existence of one element immediately implies the existence of the other. This attribute is typically used to capture the dependency that exists between a software element's specification and its implementation. Implied edges are depicted using a dashed line arrow as follows:

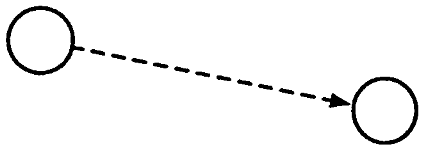


Figure 2.4(b) Implied

Restricted

The *restricted* attribute is used to represent the decompositional dependency that exists between a system element and its subelements. This form of dependency is common in systems which employ a hierarchical, top-down decomposition strategy. The subunit or separate construct in Ada is a typical example. The *restricted* edge attribute is depicted using a double width arrow as follows:

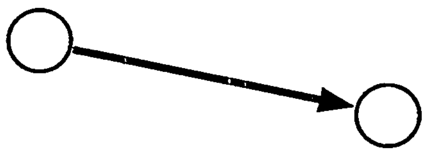


Figure 2.4(c) Restricted

Inherited

Whenever a composite system element *A* contains another element *B* which is dependent upon a third element *C*, the composite element *A* is said to *inherit* the dependency A→*C*. The *inherited* edge attribute is used to capture this relationship. The following diagram represents an inherited edge between a composite aggregate node *A* (containing node *B*) and a procedural node *C*:

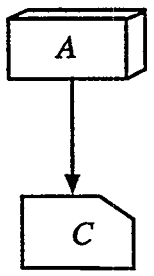


Figure 2.4(d) Inherited

Induced

Whenever a system element *A* is dependent upon a node *B* which is contained within a composite system element *C*, the dependency A→*C* is said to be *induced* on C. The *induced* edge attribute is used to capture this relationship. The following diagram represents an induced edge between a procedural node A and a composite aggregate node C which contains node B:

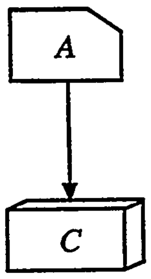


Figure 2.4(e) Induced

Visible

The *visible* attribute is a dynamic attribute that can be assigned to any edge. This attribute is used by the visualization system for edge filtering. In order for an edge to be visible, both nodes must be properly contained within the same visualization space. Visible edges are depicted as described above depending upon the edge's other attributes. Edges without the *visible* attribute are not displayed. For instance, all edges in Example 1.2 except INPUT→IO, OUTPUT→IO, COMPUTE→SCALAR, COMPUTE→VECTOR, and, MATRIX→VECTOR possess the visible attribute.

Note than an edge can contain any one or more of the above attributes. When more than one attribute is applied to an edge, the graphical features of each attribute are combined for the resulting depiction. For example, the diagram in Figure 2.4(f) contains an *implied*, *restricted*, *induced*, *inherited*, *visible* edge.

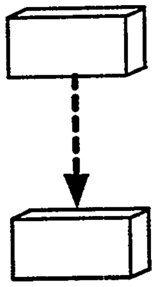


Figure 2.4(f) Combined attributes

Figure 2.4 Edge attribute representations

In summary, the set δ = { *universal, implied, restricted, inherited, induced, visible}*. For notational convenience, we also define a series of boolean function $f_\delta(e)$ which will allow us to test whether or not a particular edge possesses a specific attribute from δ. For each element β in δ, we define a function $f_\beta(e)$ which when passed an edge $e \in E$, returns true if e possess the attribute β and false otherwise. If we let e be the edge represented in Figure 2.4(f), the following boolean expressions apply:

UNIVERSAL(e) = FALSE;
IMPLIED(e) = TRUE;
RESTRICTED(e) = TRUE;
INHERITED(e) = TRUE;
INDUCED(e) = TRUE;
VISIBLE(e) = TRUE;

Example 2.2 Continuing with the same program from the previous examples, suppose the following module dependency information is extracted from its source:

- COMPUTE calls subroutines in MATRIX, SCALAR, and VECTOR
- INPUT and OUTPUT call subroutines in IO
- MAIN defines COMPUTE, INPUT, and OUTPUT as submodules
- MATRIX calls subroutines in VECTOR and SCALAR

- VECTOR calls subroutines in SCALAR

The attribute set A_e reflecting these module dependencies is shown in Table 2.2. Applying the same layout technique used in the previous examples, the resulting diagram is shown in Figure 2.5. In contrast to Figure 2.3, new information is gained about the tighter binding between MAIN, COMPUTE, INPUT, and OUTPUT. The utility of edge attributes, however, becomes more dramatic as the number of edges in the system increases. The application of the *implied*, *induced*, and *inherited* attributes will provide a convenient mechanism for reducing visual complexity. Edge attributes will be used as the basis for several of the configuration manipulation operations in Chapter 3 and configuration evaluation functions in Chapter 5. □

Table 2.2 Example Program Edge Attributes

Dependency	Attributes
COMPUTE - MATRIX	*visible*
COMPUTE - SCALAR	*universal*
COMPUTE - VECTOR	*universal*
INPUT - IO	*universal*
MAIN - COMPUTE	*restricted, visible*
MAIN - INPUT	*restricted, visible*
MAIN - OUTPUT	*restricted, visible*
MATRIX - SCALAR	*universal*
MATRIX - VECTOR	*visible*
OUTPUT - IO	*universal*
VECTOR - SCALAR	*visible*

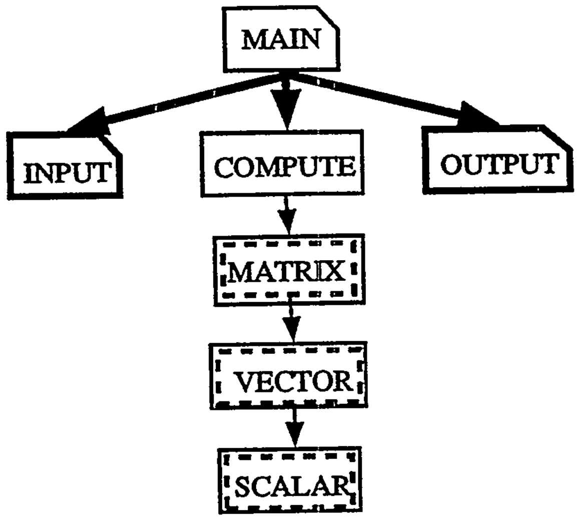


Figure 2.5 Improved layout with edge attributes

2.9 Auxiliary Edge Attributes

While the attributes described in the previous section provided a useful structural characterization of dependencies, by no means do they form an exhaustive attribute set. Just as a wealth of information can be associated with the nodes representing system elements, so to can numerous attributes be defined for their edges. Recalling that an edge represents a dependency between two system elements, an alternative method of characterizing these dependencies is to regard a dependency as an *interface* between two system elements. An interface consists of a set of conventions for exchanging information between two system elements. An interface consists of three components [Wa'81]:

· A set of visible abstract objects and for each a set of allowed operations and associated parameters.

· A set of rules governing the legal sequences of these operations.

· The encoding and formatting conventions required for operations and parameters.

The process of defining a system's abstract objects and their operations is perhaps one of the most widely accepted practices in modern software engineering. Based on this organization, we will assume that each abstract object is represented as a node within a configuration. We will also assume that each distinct operation that can be initiated by an object and performed on another abstract object is represented as an edge within this configuration. The attributes which may be associated with an edge consist of the operations, the parameters associated with these operations, the rules for exchanging these parameters, and the encodings and formatting conventions used in this exchange. A wealth of attributes can therefore be associated with an edge, if desired. A summary of these attribute types is given here using this characterization.

Operations

Within a typical programming language, a common operation that is performed is a subprogram invocation such as a procedure or function call. Other operations include message exchange, data references such as variable or constant access, context clauses (such as the Ada with statement), visibility directives (such as the Pascal with or Ada use statement), task rendevous, remote procedure calls, and exception passing. The types of attributes characterizing an edge operation might therefore include the *name* of the operation (e.g. PRINT) and the specific operation *mechanism* (e.g. *procedure call*). Dynamic properties such as frequency or probability of use, occurance history, average response time, and communications path could also be defined to aid in testing, debugging, and performance measurement.

Parameters

Associated with each abstract operation is a series of zero or more parameters that comprise the information that is to be exchanged during the course of an operation's execution. These parameters specify the quantity and type of information that a particular operation must process or must produce upon completion. In a typical programming language, a distinction is generally made between formal parameters which denote the named entities within the operation's specification, and actual parameters which denote the particular entities that are associated with these corresponding formal parameters and that are actually processed during the execution of the operation. The *class* attribute is used to carry this distinction. The *number* of parameters, the *type* of data structure associated with each parameter, the static *name* of each formal and actual parameter, and the dynamic *name* and *value* of each actual parameter are examples of several other useful edge attributes.

Rules

Associated with each operation is a set of explicit or implicit rules that describe the information transmission including the exchange *mechanism*, the parameter exchange *mode*, the proper operation *sequence*, the operation *synchronization* techniques, the necessary methods of *protection*, the parameter *associations*, and the *error control* mechanism.

The *mechanism* attribute is concerned with the specific method of exchanging parameters between objects. In conventional programming languages, parameters are typically exchanged using one of the following popular techniques described in [Pr76]:

- Exchange by value
- Exchange by value result
- Exchange by reference
- Exchange by location
- Exchange by name
- Exchange by simple name

The *mode* attribute associated with an edge specifies whether the actual parameter associated with operation is supplied by the originator of the operation or the recipient of the operation or both. The **in**, **out**, and **in out** parameter mode designation used in Ada and Modula-II is an example.

The *sequence* attribute is useful in determining the proper order of parameter exchange throughout the execution of an operation. Parameters may be passed collectively as a set and immediately accessible or they may be exchanged individually and accessible only serially via a queue or stack. Alternatively, return parameters such as a function value or a message acknowledgment, may be accessible only at the completion of the operation.

Associated with every edge is also an attribute that describes the synchronization mechanism implied by the dependency. This dependency might imply a specific elaboration sequence; it may indicate sequential or parallel execution of the two corresponding nodes; it may describe a particular execution interaction such as a task rendevous or message communication; or perhaps dictate an explicit method of state information control such as single context subprogram execution versus multiple context coroutine operation.

Related to both the *sequence* and *synchronization* attributes, is the need for an *association* attribute that binds each actual parameter of an operation to an identifier declared in the formal parameter specification, providing a method for referencing each actual parameter. Three popular methods for performing this function are to use either name association where a parameter is bound by linking it via an identifier contained in the formal parameter specification, positional association where a parameter is bound by virtue of the order it occurred in the actual parameter description, or type association where an actual parameter is bound to its formal parameter by virtue of its type characterization.

The exchange of parameters often involves the use of a security mechanism for limiting access to information via methods such as encryption, password access, an authentication sequence, a protected address space, a rights or capability identifier, and an access list. The *protection* attribute is used to describe the particular scheme(s) associated with the edge.

The *error control* attribute is used to indicate the particular method used in handling abnormal conditions. Notification of an error condition can be performed explicitly via a predetermined parameter associated with the operation such as a status variable or condition code, via an exception or interrupt mechanism where the current operation is abandoned and an alternative operation sequence is initiated, or via a separate interface where an alternative operation is initiated as a result of the error condition.

Encodings

With each parameter in an operation specification, we can associate a *representation* attribute. This attribute describes the mapping or mapping criteria of a parameter's data type onto the detailed features of an underlying machine architecture or system implementation. Example uses of this attribute include the amount of memory allocated per parameter and individual parameter fields, memory architecture delimiters such as bit, byte, word, or block boundaries, and physical location references such as specific memory or file data block addresses or offsets.

Obviously, the diversity of potential edge attributes that could be defined and associated with a particular edge within a configuration is enormous. The cataloging just presented primarily provides a summary of the issues that are associated with every system dependency or system interface. There are a variety of other system aspects, however, that are not necessarily associated with any particular system element/dependency nor with any particular node/edge representation. These aspects are examined in the following section.

2.10 Configuration Properties

The discussion up to this point has been primarily concerned with the organizational aspects of a system as it pertains to the structure of its elements, the dependencies between these elements, and the information associated with each element and with each dependency. Understanding of a complex system design, however, involves a great deal more than an enumeration of these items. Several fundamental properties of a system must be abstracted from this information in order to provide a thorough characterization. These properties may be organized into the following four basic groups [Ch'90]:

- Structural
- Functional
- Dynamic
- Behavioral

Before proceeding with our discussion, it is important to show how these properties may be captured using the configuration structure developed so far. Some minor enhancements to our configuration definition which enable time dependency will be required for this purpose.

The structural properties of a system pertain to the decomposition of a system into subsystems, elements, modules, etc., and the description of how resources or information are exchanged among these components through their interfaces. The graph structure $G = (V, E)$ of the configuration definition captures this property.

In contrast to structural properties, the functional properties of a system describe not just the information exchanges within the system, but the meaning of those information exchanges. The edge attribute mechanism provided in the A-Vu model is intended for this purpose. Additional attributes can be added to A_v and A_e as necessary to sufficiently characterize each element dependency.

As presented throughout Section 2.7 and Section 2.9, the dynamic properties of a system can also be captured via attribute characterization, $A = (A_v, A_e)$. However, the values of these attributes must not be considered as static, but rather allowed to change if necessary over time. That is, $A = f(t) = (A_v(t), A_e(t))$.

The behavioral properties of a system are described in terms of the relationships among system elements, their attributes, and their dependencies. As such, the A-Vu model provides no explicit mechanism for capturing this behavior, However, by treating a configuration as a function of time, $C = f(t) = (G, L(t), A(t))$, and recording the node and edge attribute information, a profile characterizing the system's behavior can be obtained.

In contrast to conventional graph layout, our configuration definition provides a mechanism for associating considerable semantic information with each node and edge. This mechanism allows layout of the configuration to be performed using this information in addition to the conventional node/edge relationships. From a visualization standpoint, this mechanism provides a convenient method of presenting many of the other characteristics of a system. Each node and edge in the configuration now possesses state information which can be visually expressed using graphical techniques such as color, shading, and motion, providing a modeling basis for animating a system's structure and its execution. For example, the color of a node could be used to represent the current execution state of its corresponding system element and the activity of any associated dependency.

While the A-Vu prototype tool described in Chapter 7 focuses on the structural properties and functional properties of a system, the A-Vu model is now sufficiently general to accommodate many of a system's dynamic and behavior aspects as well. The natural enhancements proposed for the A-Vu system which would incorporate these features are presented at the conclusion in Chapter 9.

Throughout our discussion thus far, attribute information has been associated either with a specific node or a specific edge within a configuration. In modern systems design, techniques are widely used which provide a step-wise refinement process for decomposing a system system into various subsystems and modules. The criteria for this decomposition process are well established [Pa'72]. Further enhancements must still be made to our configuration definition to reflect this need. A method for allowing attributes to be collectively assigned to groups of nodes and edges will now be presented.

2.11 Composite Configurations

In Section 2.6, the notion of a composite node was briefly introduced, but a formal definition has not yet been given. This situation will be rectified in this section by introducing the concept of composite a node, layout, or configurations and showing how this mechanism can be used to further aid complex system understanding.

In a typical software system, it is not unusual for the number of element dependencies to be orders of magnitude greater than the number of elements (e.g. module dependencies). As a result, the structure of a system becomes exasperatingly difficult to understand as the number of elements increases. Recalling Figure 1.3, the graph structure of even a modest size system quickly becomes obscured due to the large number of edges that are drawn in its diagram.

To address this issue, a mechanism for reducing the number of nodes in the graph without jeopardizing (and hopefully improving) our ability to understand the system must be introduced. A commonly accepted software engineering solution to this problem is to cluster or group highly cohesive system elements into a single subsystem. The A-Vu model incorporates this ability by allowing collections of graph nodes to be collapsed into a single composite node. On the "inside" of this new node, a separate visualization space is created, allowing the dependency structure of its members nodes to also be visualized. This process can be recursively applied, providing multiple levels of nesting.

Returning to our basic definition of a configuration, $C = (G, L, A)$, we can now allow L to be a set of layouts rather just a single layout. That is, $L = \{L_1, L_2, L_3, \ldots , L_l\}$ where l is the total number of layouts in the configuration and each layout is of the form $L_i = (P_i, S_i)$. In order to construct composite layout structures, a mechanism is needed for associating a node to a particular layout. This can be accomplished by introducing the notion of a *binding* between a node and its composite layout. This relationship is defined as an ordered pair $b = (v_i, L_j)$ where $v_i \in V$ *and* $L_j \in L$. Alternatively stated, $b \in B$ where $B \subseteq V \times L$.

Our configuration definition can now be updated as follows: $C = (G, L, A, B)$. That is, a configuration is a directed graph, a set of layouts for subsets of the directed graph, a set of attributes characterizing each node and edge of the directed graph, and a set of bindings which link a node in the directed graph to a layout.

In order to create a composite node, an additional element v_c must be added to the set V defined for G, a new layout L_c must be created and added to the layout set L, and an additional binding $b = (v_c, L_c)$ must be added to the binding set B. As nodes are inserted into a composite layout, additional edges may be introduced as a result and must be added to the edge set E. The rules for assigning the *inherited* and *induced* attributes must be applied to all intervening nodes as described in Section 2.8. The details of the insertion operations are described in Section 3.2.

To aid in identifying the contents of a composite node, an attribute inheritance precedence is established, inspired by [BG'92]. When a single node is contained within a composite node, the composite node inherits all of the attributes of that node in addition to the *composite* attribute. When two or more nodes are consolidated in a composite node, the new node may inherit some or all the combined attributes of the member nodes. The following attributes are inheritable by the composite node only if *all* of the composite node's member nodes possess the same attribute: *universal, procedural, functional, parallel, standard, generic, instantiation, implementation,* and *foreign.* The *specification* and *aggregate* attributes are inherited by a composite node when *any* of the composite node's member nodes possess these attributes. A composite node is also assigned the *aggregate* attribute when its member nodes contain two or more of the following attributes: *universal, procedural, functional,* or *parallel.* These simple inheritance rules enable a user to determine the general contents of a composite node from an outer configuration without having to actually select the composite node's visualization space.

Example 2.3 Returning to our example program, suppose the system modules MATRIX and VECTOR are typically used in conjunction as a linear algebra package. It would be desirable, therefore, to consolidate these two nodes into a single composite node labeled LINEAR_ALGEBRA. Next, recall that in Example 1.2 the module IO was removed since its use was assumed to be of little consequence. Rather than eliminate IO from the configuration, an alternative approach would be to combine IO with both the modules INPUT and OUTPUT, forming two additional composite nodes. If the internals of composite nodes INPUT and OUTPUT were then to be examined, the reference to IO would be apparent. In this manner, all of the structural information for the system can be preserved while still retaining a simplified top-level system view. The resulting configuration is shown in Figure 2.6.

The formal configuration description for Figure 2.6, assuming static attribute associations, is given by $C = (G, L, A, B)$ where $G = (V, E)$, $A = (A_v, A_e)$ with V and A_v shown in Table 2.3 and E and A_e shown in Table 2.4. The layout set $L = \{L_1, L_2, L_3, L_4\}$ defines the node positions with their corresponding visualization spaces. The original layout is L_1 and the layouts for the the three new composite layouts are L_2, L_3, L_4. The binding set $B = \{$(INPUT_IO, L_2), (LINEAR_ALGREBRA, L_3), (OUTPUT_IO, L_4)$\}$.

Table 2.3 Example Program Node Attributes

Module	Attributes
COMPUTE	*procedural, visible*
INPUT	*procedural, standard*
INPUT_IO	*composite, procedural, standard, visible*
IO	*aggregate, foreign*
LINEAR_ALGEBRA	*composite, aggregate, visible*
MAIN	*procedural, visible*
MATRIX	*aggregate,instantiation*
OUTPUT	*procedural, standard*
OUTPUT_IO	*composite, procedural, standard, visible*
SCALAR	*aggregate, instantiation, visible*
VECTOR	*aggregate, instantiation*

Table 2.4 Example Program Edge Attributes

Dependency	Attributes
COMPUTE→LINEAR_ALGEBRA	*induced, visible*
COMPUTE→MATRIX	*universal*
COMPUTE→SCALAR	*universal*
COMPUTE→VECTOR	*universal*
INPUT→IO	*universal*
LINEAR_ALGEBRA→SCALAR	*inherited, visible*
MAIN→COMPUTE	*restricted, visible*
MAIN→INPUT	*restricted*
MAIN→INPUT_IO	*restricted, induced, visible*
MAIN→OUTPUT	*restricted*
MAIN→OUTPUT_IO	*restricted, induced, visible*
MATRIX→SCALAR	*universal*
MATRIX→VECTOR	*universal*
OUTPUT→IO	*universal*
VECTOR→SCALAR	*universal*

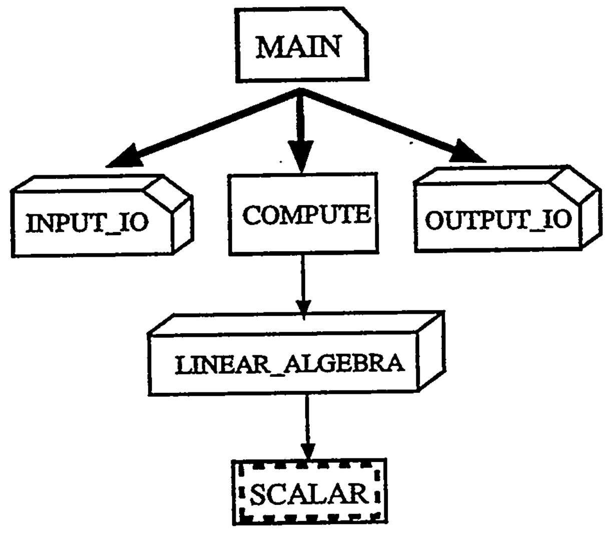


Figure 2.6 Composite layout of Table 1.1

Note that Figure 2.6 actually contains four configurations; the top level configuration and the configuration within each of the three composite nodes. Each of these configurations contains its own visualization space. A "zoom" or expand operation on the

LINEAR_ALGEBRA node, for example, would reveal the composite layout shown in Figure 2.7. □

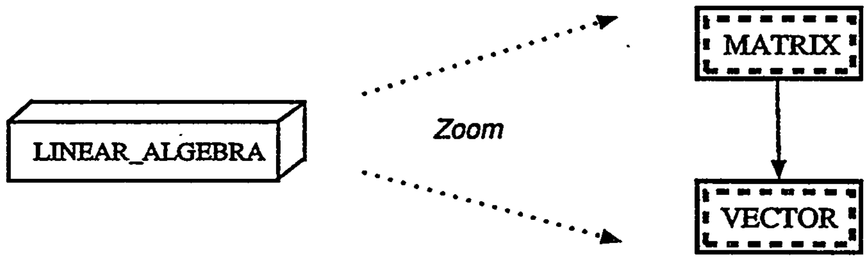


Figure 2.7 Composite configuration, L_3

To differentiate between composite and non-composite layouts, a layout which contains no composite structures will be referred to as a *simple* layout. The diagrams in Figure 2.3 and Figure 2.5 represent simple layouts while Figure 2.6 and Figure 2.7 represent composite layouts.

With the introduction of composite spaces, an additional enhancement to our configuration definition is in order. Note that the nodes INPUT, OUTPUT, and IO, MATRIX, and VECTOR were not marked with the *visible* attribute nor were the edges MATRIX→VECTOR, INPUT→IO, and OUTPUT→IO. Yet, if any of the composite spaces L_2, L_3, or L_4 are examined as was done in Figure 2.7 (i.e. L_3), certain nodes and edges may then become visible while others will become invisible. Consequently, the *visible* node and edge attributes are dependent upon the visualization space.

To capture this enhancement in the configuration framework, a minor modification to the layout definition is needed. Recall that a single layout $L_i \in L$ was defined as $L_i = (S_i, P_i, Q_i)$ where $P_i = \{p_1, p_2, p_3, ..., p_n\}$ and $Q_i = \{q_1, q_2, q_3, ..., q_k\}$. If the restriction is relaxed that P and Q be in one-to-one correspondence with V and E, respectively, we will be able to readily add and remove nodes and their corresponding edges from a visualization spaces without concern for the visible attribute. Node visibility is then dependent upon which layout and visualization space is selected. Visibility of a node v within a particular space can then be tested as follows:

$$\boxed{\text{Given a layout } L_i = (S_i, P_i, Q_i)\text{, does there exist a } p \in P_i \text{ for node } v?}$$

To establish the correspondence between a node and its position in S_i, we could re-define P_i as a set of two-tuples, $P_i \subseteq V \times S_i$. To simplify the presentation below, we will assume that this correspondence is implied with each position vector $p \in P_i$.

Visibility of an edge e in a layout L_i can be tested in a similar manner:

$$\boxed{\text{Given a layout } L_i = (S_i, P_i, Q_i)\text{, does there exist a } q \in Q_i \text{ for edge } e?}$$

While this definition is adequate, it requires that we maintain both P_i and Q_i for every visualization space. We can simplify this process by noting that edge visibility is directly linked to node visibility. An edge will be visible within a particular space if and only if its two nodes are visible. Consequently, our visualization system can determine edge visibility and positioning by simply tracking node positions. While the definition of a layout still holds, we will assume that Q_i can be derived given P_i, V, and E. Similarly, we will assume that the correspondence between an element $q \in Q_i$ and E where $q \subseteq E \times S$ is implied.

Note that this discussion assumes that nodes are the primary objects that are moved throughout visualization spaces and that their corresponding edges follow their movements. An alternative approach might be to perform edge positioning and let nodes be positioned based on their movement. This presentation focuses on the prior assumption since each node represents a fundamental system building block. While an edge is associated with exactly two nodes, a node may be associated with numerous edges. Examination of a single dependency will only yield limited information about its associated two system elements, while examination of a single element and its many dependencies is potentially much more revealing. Furthermore, edge manipulation can be simulated with node manipulation via the ability to simultaneously manipulate both nodes associated with a particular edge at once. The techniques presented in Chapter 3 enable these type of operations.

2.12 Meta-Configurations

While the introduction of nested configurations provides a convenient method of reducing the number of nodes present in a particular configuration, it also potentially introduces another system understanding problem. The most recent configuration definition from the previous section placed no restrictions on the number of configurations nor the depth of configuration nesting that may be applied. Furthermore, the definition placed no limit on the number of configurations a particular node may belong to and even allowed a node to be placed within its own visualization space. As additional composite layouts are defined and nodes inserted throughout these spaces, another complex dependency structure will result. Consequently, it would appear that an entirely new visualization problem has been created.

Fortunately, this new problem is no different than the original dependency problem faced at the onset. A new configuration is defined where each node represents a composite layout in the original configuration and each edge represents a visualization space dependency in the original configuration. The problem of visualizing this new structure can then be equated to our original visualization task, allowing identical modeling techniques to be applied. A configuration derived in this manner is termed a *meta-configuration* of the original configuration. The process by which a meta-configuration is created is referred to as an *export* operation that is applied to the original configuration (See Section 3.7).

As we shall see, the export process defines a mapping from a configuration onto its meta-configuration. There are numerous ways that these mappings can be defined, however. This discussion will focus on two immediately useful mappings; one for examining layout containment structure, the other for examining layout dependency structure.

In order to proceed with these definitions, it is first necessary to formalize the notion of node and layout. A node is said to be *directly* contained within a particular layout if the node itself has been directly positioned within the visualization space associated with the layout. That is, v_i is contained in a layout $L = (S, P, Q)$ if $v_i \rightarrow p_i, p_i \in P$. For example, the layout L_1 in Example 2.3 contains the following six nodes: MAIN, OUTPUT_IO, COMPUTE, INPUT_IO, LINEAR_ALGEBRA, and SCALAR. For notational convenience, node containment within a layout will be indicated using the traditional set inclusion symbol $\in$. For example, $v \in L$ where v is a node and L is a layout signifies that v is positioned in the visualization space S associated with L as indicated by the set P.

This containment concept can be further refined by applying it recursively, examining any composite layouts that are be bound to any of the nodes in the original layout. A node is said to be *indirectly* contained within a layout if the node is not directly contained in the layout, but instead is either directly or indirectly contained in one or more of the composite layouts bound to any of the nodes in the original layout. For example, the nodes INPUT, OUTPUT, IO, MATRIX, and VECTOR are indirectly contained in layout L_1 in Example 2.3. To complete the containment definition, a node is said to be simply *contained* within a layout if it is either directly or indirectly contained in that layout.

Since composite layouts involve binding between nodes and layouts, it is often convenient to refer to both node and layout containment. A node v_i is said to be contained with another node v_j it the node v_i is contained within a layout that has been bound to node v_j. Similarly, a layout L_i is said to be contained with another layout L_j if the layout L_i is bound to a node that is contained within the layout L_j. Finally, a layout L is said to be contained within a node v if the layout L is either bound directly to v or bound to a node contained within v. Hence, the notion of node containment and layout containment will often be used interchangeably with the appropriate node/layout binding structure implied.

Note that under these definitions, there is nothing to preclude either nodes or layouts from being contained within themselves. This can occur by placing a node in a layout and then binding that layout either directly to the node itself or indirectly via intervening node/layout bindings. The meta-configuration process is useful for exploring and revealing these different types of relationships. With a suitable containment definition now in place, its possible to proceed with the meta-configuration generation discussion that makes use of this concept.

Layout Containment Structure

The first meta-configuration mechanism to be discussed is concerned with the structure that can be extracted from a configuration by examining its node/layout containment relationships. Of particular interest is the understanding of how the different layouts within a configuration are contained within each other as a result of node placement and node bindings. This is addressed by mapping the layouts of the original configuration onto new nodes in the meta-configuration. Similarly, instances of layout containment in the original configuration will be mapped onto edges in the meta-configuration.

To formalize this notion, a function or mapping $\phi_{\subset}$ is defined that maps a configuration C onto a new configuration, its meta-configuration, C'. This relationship is expressed by:

$$\phi_{\subset}: C \rightarrow C' \text{ or}$$

$$\phi_{\subset}: (G, L, A, B) \rightarrow (G', L', A', B').$$

For every layout $L_i \in L$ in C, the mapping ϕ assigns a node $v' \in V'$ in C'. That is,

$$\phi_{\subset}: L_i \rightarrow v_i'.$$

Similarly, for every pair of layouts $L_a, L_b \in L$, where L_a contains a node v and $(v, L_b) \in B$, the mapping $\phi_{\subset}$ assigns an edge $e_{ab}' \in E'$ in C' or

$$\phi_{\subset}: (L_a, L_b)_v \rightarrow e_{ab}'.$$

Note that not all layout pairs in C are mapped onto an edge in C'. Only those layouts with connected visualization spaces in C are represented in C'. As a result, an isolated layout in C with no node bindings will map to a node in C' which no other node will be dependent on. Similarly, any layout in C which contains no composite nodes will map onto a node in C' with node dependencies.

To retain visual consistency, the mapping of attributes from $C \rightarrow C'$ will follow similar inheritance rules to those outlined above for nodes and edges. A new node v_i' in C' will inherit its attributes from all nodes that were contained in its layout L_i from C. This method enables the same inheritance rules to be applied to all layouts, regardless of whether or not they were bound to a node in C. Since all new nodes in C' are initially not bound to any new layouts in C, the *composite* attribute will automatically be dropped from any new node in C' to retain visual consistency. Similarly, the *inherited* and *induced* edge attributes must also be dropped. Since all initial edges in C' represent a hierarchical, top-down decomposition as a result of direct layout containment of layouts in C, the new edges in C' will acquire the *restricted* attribute.

In practice there will frequently be at least one (and usually only one) layout which is not bound to any node. Such a layout provides a convenient "top-level" view of the system being visualized. When generating a meta-configuration, we can allow new nodes in C' to acquire their *name* attribute from the node which was bound to the layout in C. Using this approach, new nodes in C' mapped from any of these unbounded layouts in C will have an undefined *name* attribute. For our purposes, an arbitrary, but unique name will be assigned to these types of nodes for this purpose.

Example 2.4 Continuing with from the previous example, we will now proceed to define the meta-configuration for the composite system shown above in Figure 2.6. Recall from Example 2.3 that this system involves four layouts L_1, L_2, L_3, and L_4. The top level layout, L_1, contains three nodes that are each bound to one other layout, resulting in three edges corresponding to $L_1 \rightarrow L_2$, $L_1 \rightarrow L_3$, and $L_1 \rightarrow L_4$. The names INPUT_IO, LINEAR_ALGEBRA, and OUTPUT_IO are acquired from the nodes in L_1 that were bound to the original layouts, L_2, L_3, and L_4, respectively. Since the original layout L_1 was not bound to a node, the *name* attribute for its corresponding node in the meta-configuration is also set to "L_1". The layout containment node and edge information for Figure 2.6 is shown in Table 2.5 and Table 2.6 respectively. The resulting meta-configuration diagram is shown in Figure 2.8. Note that the *procedural* and *aggregate* attributes were retained in the export process for their corresponding nodes.

Table 2.5 Layout Containment Nodes

Element	Attributes
L_1	*aggregate, visible*
INPUT_IO	*procedural, standard, visible*
LINEAR_ALGEBRA	*aggregate, visible*
OUTPUT_IO	*procedural, standard, visible*

Table 2.6 Layout Containment Edges

Dependency	Attributes
L_1→INPUT_IO	*restricted, visible*
L_1→LINEAR_ALGEBRA	*restricted, visible*
L_1→OUTPUT_IO	*restricted, visible*

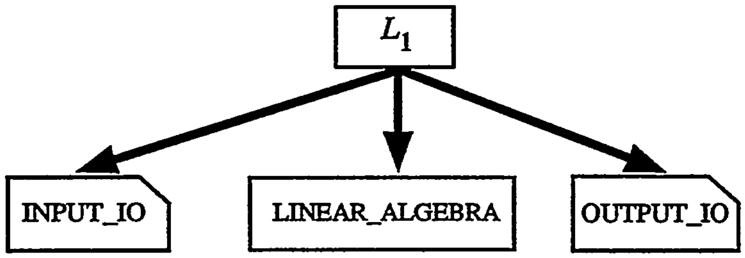


Figure 2.8 Layout containment meta-configuration for Figure 2.6

Layout Dependency Structure

In addition to examining a configuration's layout containment structure, it may also be instructive to examine its layout dependency structure. In contrast, layout dependency structure is concerned with the relationships that exist between layouts as a results of the nodes that were placed in their visualization spaces and not simply with how the layouts have been bound together. Whenever a node is placed within a composite node, the composite node (and its accompanying composite layout) will inherit all of that node's dependencies including those nested at great depth in the layout containment structure. The visualization of this layout dependency structure provides an effective method of summarizing a complex dependency arrangement.

To formalize this notion, we again define a function or mapping $\phi_{\rightarrow}$ that maps a configuration C onto a new configuration, its meta-configuration, C'. This relationship is expressed by:

$$\phi_{\rightarrow}: C \rightarrow C' \text{ or}$$

$$\phi_{\rightarrow}: (G, L, A, B) \rightarrow (G', L', A', B').$$

For every layout $L_i \in L$ in C, the mapping $\phi_{\rightarrow}$ assigns a node $v' \in V'$ in C'. That is,

$$\phi_{\rightarrow}: L_i \rightarrow v_i'.$$

Similarly, for every pair of nodes v_a, v_b where $v_a \in L_i$, $v_b \in L_j$, $(v_a, v_b) \in E$, $v_a \neq v_b$, and $L_i \neq L_j$, the mapping $\phi_{\rightarrow}$ assigns an edge $e_{ab}' \in E'$ in C' or

$$\phi_{\rightarrow}: (v_a, v_b) \rightarrow e_{ab}'.$$

Note that not all edges in C are mapped onto an edge in C'. Only those edges which span visualization spaces in C are represented in C'. Similarly, an isolated layout in C with no dependencies on other layouts will map to an isolated node in C' with no other dependencies on other nodes.

To retain visual consistency as before, the mapping of attributes from $C \rightarrow C'$ will follow again similar inheritance rules as those described above. A new node v_i' in C' will inherit its attributes from all nodes that were contained in its layout L_i from C as previously described. The *composite*, *inherited* and *induced* edge attributes will again be dropped. Unlike our attribute strategy for layout containment, however, the new edges in C' will inherit their from the union of all other edges attributes that spanned the two original layouts in C. The *implied* attribute is applied to any edge which results from a layout being dependent upon any in which it is properly contained. Such a dependency is generally the case and usually implied, thereby justifying the use of this attribute. The *name* attribute strategy used above for layout containment will again be used.

Example 2.5 Returning once more to the composite system shown in Figure 2.6, a layout dependent meta-configuration will now be generated. Recall again from Example 2.3 that this system involves four layouts L_1, L_2, L_3, and L_4. The top level layout, L_1, contains one or mores nodes that are dependent upon at least one other node contained in each of the other three layouts, resulting in three dependencies, $L_1 \rightarrow L_2$, $L_1 \rightarrow L_3$, $L_1 \rightarrow L_4$ (e.g. MAIN→INPUT, COMPUTE→MATRIX, MAIN→OUTPUT. Since both layouts L_2 and L_4 contain the node IO and they both also contain a node dependent upon IO (i.e. OUTPUT, and INPUT, respectively), the dependencies $L_2 \rightarrow L_4$ and $L_4 \rightarrow L_2$ result. Lastly, since the layout L_3 contains the one or more nodes that are dependent upon at least one node in L_1 (e.g. VECTOR→SCALAR), the dependency $L_3 \rightarrow L_1$ is also present. The names INPUT_IO, LINEAR_ALGEBRA, and OUTPUT_IO are acquired from the nodes in L_1 that were bound to the original layouts, L_2, L_3, and L_4, respectively. Since the original layout L_1 was not bound to a node, the *name* attribute for its corresponding node in the meta-configuration is once again set to "L_1." The resulting layout dependency meta-configuration node, edge, and diagram for Figure 2.6 is shown Table 2.7, Table 2.8, and Figure 2.9, respectively. Note the retention of the *procedural, aggregate, restricted* attributes for their corresponding nodes and edges. □

Table 2.7 Layout Dependency Nodes

Element	Attributes
MAIN	*aggregate, visible*
INPUT_IO	*procedural, standard, visible*
LINEAR_ALGEBRA	*aggregate, visible*
OUTPUT_IO	*procedural, standard, visible*

Table 2.8 Layout Dependency Edges

Dependency	Attributes
INPUT_IO→OUTPUT_IO	*universal*
INPUT_IO→L_1	*universal*
LINEAR_ALGEBRA→L_1	*inherited, visible*
L_1→INPUT_IO	*restricted, induced, visible*
L_1→LINEAR_ALGEBRA	*universal*
L_1→OUTPUT_IO	*restricted, induced, visible*
OUTPUT_IO→INPUT_IO	*universal*
OUTPUT_IO→L_1	*universal*

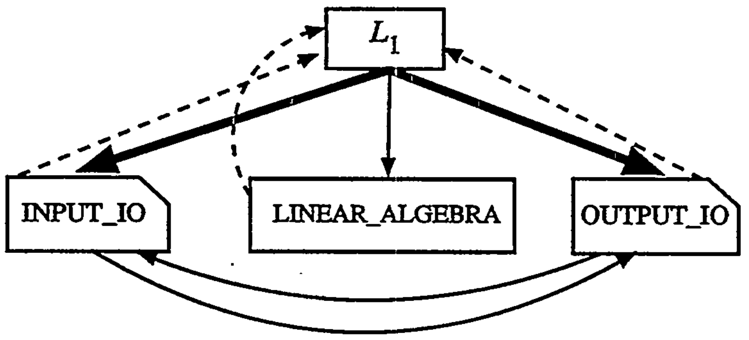


Figure 2.9 Layout dependency meta-configuration for Figure 2.6

In the process of exploring a complex dependency structure, a typical sequence would be involve the creation of a configuration followed by its manipulation and, if necessary, the generation of a meta-configuration. The use of a meta-configuration provides another tool to aid in the understanding of a system's dependencies at another conceptual level. Note that the configuration editing/meta-configuration export cycle can actually be repeated indefinitely. Once a configuration export has been performed using either of the above techniques, the new meta-configuration could then be manipulated, consolidating nodes, creating additional composite spaces, etc.. An additional meta-configuration could then be exported from this resulting configuration and the process repeated.

In actual use, it is unlikely that this process would be useful to repeat beyond a few iterations. With each export operation of a configuration C_i, a new configuration C_j is defined which initially contains only one layout and a set of nodes V_j where $|V_j|$ is equal to the number of layouts in WC_i, (i.e. $|V_j| = |L_i|$). Consequently, the complexity of each subsequent meta-configuration will reduce accordingly. If no modifications are made to a configuration's meta-configuration, a second export will yield a configuration with a single node, representing the single layout/visualization space resulting from the initial export. Repeated exports without further modification would continue to yield a single node configuration. Hence, the configuration shown in Figure 2.10 will be referred to as the *identity* configuration and will always be produced after two iterations of either process if no intervening modifications are made to the configuration.

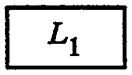


Figure 2.10 The *Identity* configuration.

2.13 Configuration State

An interactive environmental that implements the A-Vu configuration model presented in this chapter is envisioned. To provide such an environment, it will be necessary to examine and manipulate the contents of individual layouts, select items within these layouts, remove them, and position them in other layouts. The dependency and attribute information associated with these nodes must be updated automatically during these operations. To perform these functions, some additional state variables will be required. One approach would be to incorporate these variables into our configuration definition. This approach has the advantage of allowing the state information to be stored and transmitted with the configuration, thereby being accessible in subsequent interactive sessions. However, the utility of retaining this information across multiple sessions is rather limited. Rather than complicate our configuration model, these state variables will be defined separately and considered valid only during the course of a single session.

To begin with, the state variable Λ is introduced to indicate which layout $\Lambda = L_i \in L$ has been selected for current viewing and processing. This variable is useful in many of the cut and paste type operations that will be described in Section 3.2. In order to direct these editing operations, we will also need a mechanism for indicating which nodes certain operations will be applied to. To accomplish this, we will define a set $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, ..., \sigma_l\}$ where $l = |L|$ and $\sigma_i \subseteq V$. The set σ_i contains a node $v \in V$ if and only if the v is both contained in the layout L_i and v has been *selected*. The notion of a *selected* node can be viewed simply as a layout-dependent node attribute that is either true or false. An accompanying visualization system implementation may use this attribute to indicate the *selected* condition via color, highlighting, shadowing, etc. Finally, the notion of a cut buffer X is defined as $X \subseteq V \times (S_1 \cup S_2 \cup S_3 ... \cup S_l)$ where $l = |L|$ with $L_i \in L$ and $L_i = (S_i, P_i, Q_i)$. The set X is used to save nodes and their positions as they are copied, inserted, and removed to/from various layouts. Initially, Λ is undefined, $X = \varnothing$, and $\Sigma = \varnothing$. Whenever a layout L_i is created (deleted), a node selection set σ_i is similarly created (deleted).

In summary, we can define the state of a configuration as $\psi = (\Lambda, \Sigma, X)$. The interactive visualization representation for a system is then (C, ψ). Many of the operations described in the next chapter will make use of this information.

At this stage, the conceptual framework of the A-Vu model has been completed. Some minor enhancements to the model will be made in later chapters to accommodate special implementation considerations, but largely it will remain unchanged. This framework will provide the basis for all future dependency analysis and visualization discussion. The notion of a configuration has emerged as the fundamental construct for representing arbitrary complex system dependency structure. The above discussion, however, did not address how dependency information is caste into a configuration and how the resulting configuration may be manipulated. This is the topic for the next chapter.

3. Configuration Manipulation

With the configuration concept defined, the focus of the discussion now shifts to the development of techniques for creating, modifying, and managing the configuration structures that conform to the A-Vu model. This chapter presents these techniques by progressively developing a series of primitive configuration manipulation operations. Collectively, these operations provide the fundamental building blocks necessary for complex system exploration. To aid discussion, this collection is organized by placing each operation into one of the following seven functional groups:

- Initialization
- Editing
- Selection
- Arrangement
- Reduction
- Compaction
- Archival

The initialization operations in Section 3.1 are concerned with the creation, deletion, and modification of a configuration structure and its associated visualization space. Once a suitable visualization space has been initialized, operations which manipulate the visualization space's contents can then be performed. The editing operations described in Section 3.2 provide a mechanism for creating node/edge structures in the visualization space and for moving nodes throughout this space. The selection operations described in Section 3.3 provide a series of search algorithms for locating and isolating particular subsets of nodes in the configuration. The arrangement operations in Section 3.4 are a series of graph-based algorithms used to generate particular visualization space layouts with the desired properties. The reduction operations in Section 3.5 are used to eliminate certain collections of nodes and their associated edges from the visualization space as a means of reducing the structural and visual complexity of the resulting representation. The compaction operations presented in Section 3.6 are helpful in maintaining visualization space dimensions. Finally, a set of archival operations are presented in Section 3.7 that are used for loading, saving, and restoring configurations once they are defined.

Included with the discussion of each group below is an expository description of what motivates each operation's use. The computational complexity of each operation will also be analyzed to insure suitable performance as specified in the requirements in Section 1.4. Toward the development of an *interactive* A-Vu tool, fast and responsive algorithms are essential. Consequently, the discussion below focuses on algorithms whose computational complexity is at least as good as $\Theta(n^2)$ and $\Theta(k^2)$ where n is the number of nodes in the graph, and k is the number of edges. Algorithms with poorer performance (such as $\Theta(n^3)$ or $\Theta(k^3)$ or worse) would undoubtedly lead to poor interactive performance as the number of elements in the system approaches quantities not unusual in modern system designs (e.g. $\geq$1000).

3.1 Initialization

Under the A-Vu model, the task of visualizing a complex system's dependencies must begin with the preparation of a suitable data structure capable of capturing the system's dependency information, storing its visualization representation, and binding nodes with their composite spaces. Once this initial data structure is created, we may proceed to populate it with dependency information including node/edge definitions and associated attribute information. We must then be prepared to create and delete layouts and bind and unbind layouts to nodes as we proceed to explore and manipulate a systems' dependency structure. The set of operations for performing these initial housekeeping type functions are presented here.

Configuration Initialization

Recalling our configuration definition $C = (G, L, A, B)$, we assume that our initial configuration consists of a null graph with no attributes, no layouts, and no bindings. Hence, our first operation which defines our initial configuration structure is described here.

Operation: INITIALIZE;

Description: The INITIALIZE operation creates a null configuration $C = (G, L, A, B)$ where $G = (\emptyset, \emptyset)$, $L = \emptyset$, $A = (\emptyset, \emptyset)$, and $B = \emptyset$.

Analysis: The operation INITIALIZE is $\Theta(c)$ where c is a constant since it is equivalent to a constant time data structure memory allocation.

Node Creation

Once a null configuration has been created, we can start to populate this configuration beginning with node information. The following operation adds a node to a configuration.

Operation: CREATE_NODE (v);

Description: The operation creates a new node, adds the node to the configuration, and returns a unique identifier for the node. An attribute set for the node must also be defined. Let v be the new node, then CREATE_NODE performs the following sequence:

```
$v$ := new;
$V := V \cup \{v\}$;
$\varpi_v := \emptyset$;
$A_v = A_v \cup \{\varpi_v\}$
```

Analysis: CREATE_NODE is $\Theta(c)$ since it is equivalent to a single memory allocation and two single element set union operations. A single element set union operations, equivalent to a set insertion operation, can be implemented in constant time [Ah'83].

Node Deletion

Just as nodes must be created, so too is a mechanism for node deletion required. Node deletion, however, is more complex due to potential edges with other nodes, possible attribute associations, placement in a visualization space, and participation in a composite binding (to be discussed shortly).

Operation: DELETE_NODE (v);

Description: The DELETE_NODE operation removes a node from a configuration by deleting all of its node attributes, deleting all bindings that it is associated with, removing its node and edge positions from all layouts, deleting all edges that may contain it and any corresponding edge attributes, and deleting the node from the node list. Let v be the node to be deleted, then DELETE_NODE performs the following sequence:

```
$A_v := A_v - \{\varpi_v\}$;
$\forall\, b \in B$ do
    $B := B - (v \times L)$;
end;
$\forall\, L_i \in L, L_i = (S_i, P_i, Q_i)$ do
    $P_i := P_i - \{p_v\}$;
    if $\exists e \in E, p_1 \in P_i, p_2 \in P_i, e = (v_1, v_2)$ |
        $v_1 \rightarrow p_1, v_2 \rightarrow p_2$ then
        $Q_i := Q_i - \{q_e\}$;
    end;
$\forall\, w \in V$ do
    $E := E - (v \times w) - (w \times v)$;
end;
```

```
$\forall\, e \in ((v \times V) \cup (V \times v)) \cap E$ do
    $A_e := A_e - \{\varepsilon_e\}$;
end;
$V := V - \{v\}$;
```

Analysis: The worst-case time complexity of DELETE_NODE is max{ $\Theta(|V|)$, $\Theta(|B|)$, $\Theta(|L| \times \max(|V|, |E|))$, $\Theta(|E|)$ }. The $\Theta(|V|)$ term is introduced as a result of iteration through the sets V and A_v to delete element v and any of its attributes. The term $\Theta(|B|)$ results from the deletion of its bindings. The term $\times \max(|V|, |E|))$ results from deletion of the node and its edges from every layout. The term $\Theta(|E|)$ results from an iteration through the set E in search all edges containing v and the deletion of any associated edge attributes. Since $|B| \le |L|$ is usually $<< |V|$ and $\Theta(|V|) \le \Theta(|L| \times |V|)$ and $\Theta(|E|) \le \Theta(|L| \times |E|)$, we can simplify the above equation, yielding an expected worst-case complexity of $\Theta(|L| \times \max(|V|, |E|)$) Note that this assumes a linear list implementation set implementation. Alternative implementations of set deletion operations can be performed in constant time [Ah'83], enabling the entire operation to be performed in $\Theta(|L|)$.

Edge Creation

Once one or more nodes have been defined within the configuration, edges between the nodes can then be defined. Addition of a new edge could, however, affect one or more of the configuration's layouts. If the new edge involves any nodes that were previously added to one or more layouts defined in the configuration, the edge must also be defined in these layouts. In an actual implementation, this situation may result simply in a notification to the visualization system where a new edge set within the affected layouts are updated based on the new graph description. For completeness, the entire edge creation operation is described here:

<u>Operation</u>: CREATE_EDGE (e, v_1, v_2);

<u>Description</u>: The operation defines a new edge between two nodes v_1 and v_2, adds the edge to the edge set, and then adds the edge to any layouts within the configuration that contains both v_1 and v_2. Let e be the new edge, then CREATE_EDGE performs the following sequence:

```
$e := (v_1, v_2)$ ;
$E := E \cup \{e\}$;
$\varepsilon_e := \emptyset$;
$A_e := A_e \cup \{\varepsilon_e\}$
$\forall L_i \in L, L_i = (S_i, P_i, Q_i)$ do
    if $\exists p_1 \in P_i, p_2 \in P_i \mid v_1 \rightarrow p_1, v_2 \rightarrow p_2$ then
        $Q_i := Q_i \cup \{q_e\}$;
    end;
end;
```

<u>Analysis</u>: In order to determine if the pair of nodes v_1 and v_2 exists within any layout, an $\Theta(|L|) \times \Theta(|V|)$ search through each layout's node position list would be required. Assuming that layout node lists are instead organized for constant time lookup [Ah'83], the time complexity of the entire operation could be reduced to $\Theta(L)$.

Edge Deletion

The sequence of operations required to delete an edge from a configuration is a subset of those described for DELETE_NODE. Only the attributes associated with an edge and the edge itself must be removed from the configuration and all internal layouts.

Operation: DELETE_EDGE (*e*);

Description: The DELETE_EDGE operation removes an edge from a configuration by first deleting all of its attributes and then removing the edges as described in the following sequence:

```
$A_e := A_e - \{\varepsilon_e\}$;
$E := E - e$;
$\forall L_i \in L, L_i = (S_i, P_i, Q_i)$ do
    if $\exists p_1 \in P_i, p_2 \in P_i \mid e = (v_1, v_2)$,
        $v_1 \rightarrow p_1 \wedge v_2 \rightarrow p_2$ then
        $Q_i := Q_i - \{q_e\}$;
    end;
end;
```

Analysis: Assuming again a constant time set lookup and set deletion operations, the DELETE_EDGE operation can be performed in time proportional to the number of layouts, $\Theta(L)$.

Attribute Association

As nodes and edges defined in the configuration, attributes can be associated with them. Recalling the discussion in Section 2.7 and Section 2.9, the range of attributes that can be assigned to both nodes and edges is quite diverse. To simplify our discussion, we will assume that every node and edge attribute can be described via an attribute identifier (such as those given in Section 2.6 and Section 2.8) along with an optional attribute value. The operations for associating attributes to nodes and edges in a configuration are then, respectively:

Operation: SET_NODE_ATTRIBUTE (*v*, *attribute*, [*value*]);

SET_EDGE_ATTRIBUTE (*e*, *attribute*, [*value*]);

Description: These operations add a particular node or edge *attribute* with an optional *value* to the sets $\varpi_v \in A_v$ and $\varepsilon_e \in A_e$, respectively, as described by the following sequences:

$$attribute := value;$$
$$\varpi_v := \varpi_v \cup \{attribute\};$$

and

$$attribute := value;$$
$$\varepsilon_e := \varepsilon_e \cup \{attribute\};$$

Analysis: Involving only set union operations, these operations can also be done in order $\Theta(c)$ [Ah'83].

Attribute Disassociation

It is sometimes necessary to delete a particular attribute from a node or an edge. This may be required, for example, when a node is deleting from inside a composite node, resulting in an attribute change to the composite node, or to a attribute change to any of the edges linking the composite node.

Operation: CLEAR_NODE_ATTRIBUTE (*v, attribute*);

CLEAR_EDGE_ATTRIBUTE (*e, attribute*);

Description: Removes a node (edge) from the node (edge) attribute list, respectively, via the following sequences.

$$A_v := A_v - \{\varpi_v\};$$

and

$$A_e := A_e - \{\varepsilon_e\};$$

Analysis: The time complexity for both operations are $\Theta(c)$ again assuming a constant time set deletion operation.

Layout Creation

With the ability to capture a system's dependency information in place, we are now ready to construct the visualization space that will be used to display this information.

Recalling that $L = \{L_1, L_2, \ldots L_l\}$ where $L_i = (S_i, P_i, Q_i)$, an initial layout will consists of a visualization space of specified dimensions and a null node position set. The editing operations described in the next section will be used to populate this set.

<u>Operation</u>: CREATE_LAYOUT (L_i, (x, y, z));

<u>Description</u>: The operation CREATE_LAYOUT adds an additional layout L_i to the configuration layout set L. The dimensions of the new layout's visualization space are determined by the parameters x, y, z. That is, $S_i = \{ \{0..x\} \times \{0..y\} \times \{0..z\} \}$. If x, y, and z are all integer parameters, then S_i is a discrete space. If x, y, and z are all reals, then S_i is a continuous space. If x, y, and z are a mix of integer and real parameters, then S_i is a hybrid space. The node position set P_i for L_i is initialized to the empty set. The following sequence defines a a layout creation:

```
$S_i$ := space($x$ ,$y$, $z$);
$P_i$ := $\emptyset$
$L_i$ := ($S_i$, $P_i$);
$L$ := $L \cup \{L_i\}$;
```

<u>Analysis</u>: CREATE_LAYOUT is also $\Theta(c)$ since each item in the sequence can be performed in constant time.

Note that as defined, S_i is simply a set of (possibly mixed) reals and/or integer vectors and that an actual implementation would actually only require the boundaries of this space to be registered. A variety of options in managing this space are available. Based on a measure of spatial node density such as $\delta = |V| \div (xyz)$, the dimensions of S_i could either manually or automatically be adjusted to insure adequate room for additional node placement. For discrete spaces, a similar computation could be performed along each row and/or column of each visualization space plane. Since there are generally at most $|V|$ nodes placed in a given space at one time, this computation could readily be performed on demand or, if desired, upon completion of each a node editing operation (Section 3.2). Throughout this chapter, it will be assumed that visualization spaces will automatically expand as necessary to properly contain all nodes within the layout. Compaction, however, must be explicity initiated as described in Section 3.6. A simple heuristic measure for determining suitable space dimensions is given in Section 3.6. A hybrid technique used by the A-Vu system is described in Chapter 7.

Layout Deletion

Deletion of a layout is generally only required in conjunction with the deletion of a composite node. To preserve configuration validity, deletion of a layout can not be performed unless all bindings associated with the layout are also deleted (see DELETE_BINDING below). A layout will frequently only be associated with at most one binding.

Operation: DELETE_LAYOUT (L_i);

Description: This operation first removes all bindings involving L_i then deletes L_i from the layout set. The following sequence performs this operation:

```
$\forall\, v \in V$ do
    $B := B - (v \times L_i)$; end;
$L := L - \{L_i\}$;
```

Analysis: The iteration through the node set V results in a $\Theta(|V|)$ time complexity. Maintaining the bindings in a list would reduces this complexity to $\Theta(|B|)$ via a sequential examination of each binding.

Binding Creation

The final operation in the initialization group is used to create composite visualization spaces within a configuration. Recalling that a binding $b \in B$ where $B \subseteq V \times L$., the CREATE_BINDING operation can only be performed only after the desired sequence of CREATE_NODE and CREATE_LAYOUT operations.

Operation: CREATE_BINDING (b, v_i, L_j);

Description: The CREATE_BINDING operation accepts a node and a layout identifier to create a composite space or binding. The following sequence applies:

$$b := (v, L_j);$$
$$B := B \cup \{b\};$$

Analysis: This operation is again $\Theta(c)$ for same reasons stated above.

Binding Deletion

The deletion of a binding is performed with association of a particular configuration with a node is no longer desired. This is required when the associated layout is to be deleted or associated with a different node, or the node itself is to be deleted.

Operation: DELETE_BINDING (b);

Description: The simple operation removes the binding b from the binding set B using the sequence:

$$B := B - \{b\};$$

Analysis: $\Theta(c)$.

3.2 Editing

With the foundation for creating (and deleting) configuration structures now established, we can proceed to define the operations that are necessary for manipulating configuration items. The most basic of these operations involves the positioning and movement of nodes within a visualization space. The primitive operations defined in this section are modeled after a contemporary interactive text editor, providing collective placement, removal, relocation, and duplication functions.

In a configuration with a single layout, the process of adding and removing nodes is straight forward. The coordinates of the node must simply be added to the layout's node position set and a check for any new visible edges must be performed.. With the introduction of composite layouts, however, the addition or deletion of a single node can dramatically affect both the dependencies and attributes of any node bound to the current layout or any of its parental composite layouts. The use of meta-configurations is useful to help visualize this process. Recalling the meta-configuration in Figure 2.9 from Example 2.5, the addition of a single node in, for example, either OUTPUT_IO or INPUT_IO, can dramatically effect both their edges and their node and edges attributes. Adding a copy of the VECTOR node to the OUTPUT_IO layout would result in the additional edges OUTPUT_IO → LINEAR_ALGEBRA, OUTPUT_IO → SCALAR, LINEAR_ALGEBRA → OUTPUT_IO and COMPUTE → OUTPUT_IO. Hence, whenever a node v is inserted into a composite node's layout, all nodes that were dependent upon v will now be dependent upon the composite node also. Similarly, the composite node and all nodes that contain the composite node will now inherit all of the the nodes that v is dependent upon.

Before defining the actual configuration editing operations, it is useful to first develop an operation that will take care of all the important edge and attribute inheritance operations that are required for layout modification. The RECONNECT operation is therefore defined first.

Layout reconnection

This deceptively simple operation involves some of the most fundamental and important aspects of the A-Vu model. Much of the composite layout, edge, and attribute inheritance structure hinges upon its successful execution. This operation must be performed at the completion of any node insertion or deletion sequences to insure the validity of the layout dependency and attribute information. This same operation will also prove invaluable to both the CUT and PASTE operations described below.

Operation: RECONNECT (L_i);

Description: The RECONNECT operation must examine every layout that contains a node bound to the specified layout L_i. In turn, every layout that contains a node bound to a recently updated layout, must also be updated. A standard queue mechanism is used to record the layouts in the order that they are encountered. The set LAYOUTS is used to keep track of which layouts have been update. The following sequence performs this operation, making use of the UPDATE operation described next:

```
QUEUE $\leftarrow L_i$;
LAYOUTS := $L - L_i$;
while QUEUE $\neq \emptyset$ do
    $l \leftarrow$ QUEUE;
        $\forall v \mid (v, l) \in B$ do
            $\forall m \mid (v, m) \in B$ do
                if $m \in$ LAYOUTS then
                    QUEUE $\leftarrow m$;
                    LAYOUTS := LAYOUTS - $m$;
                end;
            end;
        end;
    UPDATE ($l$);
end;
```

The UPDATE (l) operation is used to actually perform the node, edge and corresponding attribute updates for all composite nodes bound to the specified layout l. This operation is comprised of two parts: update of the composite node's attributes and update of the composite nodes incoming and output edges. The first part of the algorithm examines all nodes in the layout and forms the union ϖ of all attributes including the *composite* attribute. Each of the attributes in ϖ are then applied to all nodes bound to the specified layout.

The second part of the operation is responsible for creating and/or deleting any incoming and output edges and adding and/or removing any edge attributes that may have changed as a result of the previous layout modification operations. This is performed by examining each node v in the configuration and forming the the union ε_{in} of all attributes associated with the edges (v, u) where u is a node contained in the specified layout. The union ε_{out} is similarly generated by examining all outgoing edges (u, v). Next, each edge in the system between the node v and every node w bound to the specified layout is examined. If the edge attribute set ε_{in} is not empty, then the edge (v, w) is created if necessary and its attributes are set and cleared based on the contents of ε_{in}. Similarly, if the edge attribute set ε_{out} is not empty, then the edge (w, v) is created if necessary and its attributes are set accordingly based on the contents of ε_{out}. If either ε_{in} or ε_{out} are empty, then respective edges (v, w) or (w, v) must be deleted.

Part I - Update composite node's attributes:

```
$\varpi$ := {$composite$};
$\forall v \in P$ do
    $\varpi := \varpi \cup \varpi_v$;
end;
$\forall v \mid (v, l) \in B$ do
    $\forall \alpha \in \varpi$ do
    SET_NODE_ATTRIBUTE ($v, \alpha$);
end;
```

Part II - Update composite node's edges & edge attributes:

```
$\forall v \in V$ do
    $\varepsilon_{in} := \{induced\}$;
    $\varepsilon_{out} := \{inherited\}$;
    $\forall u \mid u \rightarrow p, p \in P, u \in V$ do
        if $(v, u) \in E$ then
            $\varepsilon_{in} := \varepsilon_{in} \cup \varepsilon_{(v, u)}$;
        end;
        if $(u, v) \in E$ then
            $\varepsilon_{out} := \varepsilon_{out} \cup \varepsilon_{(u, v)}$;
        end;
    end;

    $\forall w \mid (w, l) \in B$ do
        if $\varepsilon_{in} \neq \varnothing$ then
            if $(v, w) \notin E$ then
                CREATE_EDGE $(v, w)$;
            end;
            $\varepsilon_{in} := \varepsilon_{in} \cup \{ implied \}$;
            $\forall \beta \in \delta$ do
                if $\beta \notin \varepsilon_{in}$ then
                    SET_EDGE_ATTRIBUTE ( $(v, w)$, $\beta$ );
                else
                    CLEAR_EDGE_ATTRIBUTE ( $(v, w)$, $\beta$ );
                end;
            end;
        else
            if $(v, w) \in E$ then
                DELETE_EDGE ( $(v, w)$ );
            end;
        end;

        if $\varepsilon_{out} \neq \varnothing$ then
            if $(w, v) \notin E$ then
                CREATE_EDGE $(w, v)$;
            end;
            $\varepsilon_{out} := \varepsilon_{out} \cup \{ inherited \}$;
            $\forall \beta \in \delta$ do
                if $\beta \notin \varepsilon_{out}$ then
                    SET_EDGE_ATTRIBUTE ( $(w, v)$, $\beta$ );
                else
                    CLEAR_EDGE_ATTRIBUTE ( $(w, v)$, $\beta$ );
                end;
                end;
            else
                if $(w, v) \in E$ then
                    DELETE_EDGE ( $(w, v)$ );
                end;
            end;
        end;
    end;
```

Analysis: Since a layout will generally be bound to at most one node, the computational complexity of the RECONNECT operation is equal to $\Theta(|L|) \times \Theta(\text{UPDATE})$. Examining the UPDATE operation, Part I must iterate once through the entire node set V. Since SET_NODE_ATTRIBUTE can be performed in constant time, Part I is $\Theta(|V|)$. Part II, however, involves two nested iterations through both the node set V and through all of the nodes in a particular layout. In the worst case, a layout set may contain all of the nodes in V. Since CREATE_EDGE, DELETE_EDGE, SET_NODE_ATTRIBUTE, CLEAR_NODE_ATTRIBUTE can all be performed worst case in $\Theta(|L|)$, the worst case complexity for UPDATE is $\Theta(|L| \times |V|^2)$. The worst case complexity for RECONNECT is then $\Theta(|L|^2 \times |V|^2)$. In a typical configuration, however, $|L|$ is usually $<< |V|$. When and if $|L| >> 1$, then $|P_i|$ for any layout L_i will generally be $<< |V|$. In effect, RECONNECT is generally $\Theta(|V|^2)$ unless an unusually large number of layouts are created with numerous copies of the entire node set V placed in each of these layouts.

Node Insertion

In order for a node to be visible, it must first be inserted into a layout so that it may subsequently be positioned in the layout's visualization space. When the node is first inserted, a null or undefined position vector Φ is initially assigned as its value. The assignment of null position vectors reflects a slight enhancement to the configuration definition. This modification allows a layout to appropriately inherit a node's dependencies without the node actually appearing in the layout's visualization space. Looking back to Example 1.2, this ability is of practical importance as seen with the elimination of the IO node from view.

Note that a RECONNECT operation must always follow a series of node insertions to insure configuration consistency. The INSERT_NODE operation is now defined:

Operation: INSERT_NODE (v, L_i);

Description: Assigns the position vector $p_v = \Phi \notin S_i$ for the node v within the layout $L_i = (S_i, P_i, Q_i)$ as defined by the following sequence:

$p_v := \Phi$;
$P_i := P_i \cup \{p_v\}$;

Analysis: Constant time set insertion operation, $\Theta(c)$.

Node removal

As witnessed early on in Example 1.2, there is a need to remove nodes from a layout as well as to add them. From the previous discussion on node addition, removal of a node is also a complex process due to the introduction of composite layout structures. A RECONNECT operation must always follow a series of node removals to insure configuration consistency.

Operation: REMOVE_NODE (v, L_i);

Description: Removes position vector for node v from the layout $L_i = (S_i, P_i, Q_i)$ as defined by the following sequence:

$P_i := P_i - \{p_v\}$;
$p_v := \Phi$;

Analysis: Constant time set deletion operation, $\Theta(c)$.

Node Positioning

Once a node has been placed in a layout, it can then be made visible by assigning a position in the layout's visualization space. The position of the node may subsequently need to be changed as necessary depending upon the configuration operations that are ultimately performed. This simple operation is defined as follows:

Operation: SET_NODE_POSITION (v, L_i, p);

Description: Assigns the position vector $p = (x, y, z) \in S_i$ for the node v within the layout $L_i = (S_i, P_i)$ as defined by the following sequence:

$$P_i := P_i \cup \{p\};$$

Analysis: Constant time set insertion operation, $\Theta(c)$.

Selection

In typical interactive editing operations, it is common practice to be able to select a subset of items and manipulate or operate on them collectively. The set Σ described above is used for this purpose. The SELECT operation is used to add nodes to the individual layout node selection sets $\sigma_1, \sigma_2, \sigma_3, ..., \sigma_l$ as nodes are selected and deselected. These sets will be used to direct numerous operations described in the sections below. The SELECT operations is described as follows:

Operation: SELECT (v, L_i);

Description: Indicates that the node v contained in layout L_i is to be selected. The following sequence defines this operation:

$$\sigma_i := \sigma_i \cup \{v\};$$

Analysis: Constant time set insertion operation, $\Theta(c)$.

Deselection

Just as nodes can be selected, so too must they be deselected. The DESELECT operation is used to remove nodes from the layout node selection sets σ_1, σ_2, σ_3, ..., σ_l. The DESELECT operation is described as follows:

Operation: DESELECT (v, L_i);

Description: Indicates that the node v contained in layout L_i is to be deselected. The following sequence defines this operation:

$$\sigma_i := \sigma_i - \{v\};$$

Analysis: Constant time set deletion operation, $\Theta(c)$.

Cut

It is frequently desirable in interactive editing sequences to a remove a selected list of items because they are no longer needed or so they can be repositioned. The CUT operation is used for this purpose and makes use of the results of the SELECT operation to determine which items are involved. The set X of node, position pairs described above is used to maintain which nodes were removed.

Operation: CUT;

Description: Removes selected nodes from the current layout Λ and any possible edges that may be defined in the layout and save the nodes and their positions in the set X. The following sequence defines the CUT operation:

```
∀ $v \in \sigma_i$ do
    $\chi := (v, p_v)$;
    REMOVE_NODE $(v, \Lambda)$;
    $X := X \cup \{\chi\}$;
end;
$\sigma_i := \emptyset$;
RECONNECT $(\Lambda)$;
```

<u>Analysis</u>: Since REMOVE_NODE is $\Theta(c)$, the worst case complexity of the loop portion of the operation is $\Theta(|V|)$. If the number of layouts in the system is kept small with respect to the number of nodes, the worst case time complexity of the overall operation is $\Theta(|V|^2)$ due to the RECONNECT operation.

Copy

In a typical interactive edit sequence, it is frequently desirable to make a copy of a collection of items so that they may be inserted elsewhere. The COPY operation also makes use of the results of the SELECT operation for this purpose.

<u>Operation</u>: COPY;

<u>Description</u>: Records all selected nodes and their positions in the current layout Λ via the following sequence:

```
$\forall v \in \sigma_i$ do
    $\chi := (v, p_v)$;
    $X := X \cup \{\chi\}$;
end;
```

<u>Analysis</u>: If n is the number of nodes selected in the current layout, the operation is completed in linear time, $\Theta(n)$.

Paste

The PASTE operation is essentially the inverse of the CUT operation. A CUT operation followed by an immediate PASTE operation should leave the state of the configuration unchanged.

<u>Operation</u>: PASTE;

Description: Insert nodes from the cut buffer X into the current layout Λ and adds any new possible edges to the layout. The following sequence defines the PASTE operation:

```
$\forall \chi \in X, \chi = (v, p_v)$ do
    INSERT_NODE ($v$, $\Lambda$);
    SET_NODE_POSITION($v$, $\Lambda$, $p_v$);
    $\sigma_i := \sigma_i \cup \{v\}$;
end
$X := \emptyset$;
RECONNECT ($\Lambda$);
```

Analysis: Since INSERT_NODE and SET_NODE_POSITION are both constant time operations, PASTE is $\Theta(|V|^2)$ worst case due to the complexity of RECONNECT operation.

3.3 Selection

The operations described in this section are used to select from within the current layout Λ, a particular node or a set of nodes that possess desired properties. These properties may concern either node attributes or node relationships as characterized by edges and/or edge attributes. All of the operations in this section have been selected based on the useful role they serve in the exploration of a complex dependency structures. Many of these operations mimic common editing type operations while others capture useful graph theoretic operations.

Note that limiting these operations to the current layout Λ is mainly to simplify the discussion. Recursive versions of each of these operations could be easily adopted by re-applying the operation to the layout bound to every composite node in Λ. The computational complexity of each of these operations would then be increased by at most a factor of $|L|$. The exact increase for a specific configuration is determined by the number of layouts that are contained in the current layout Λ.

Global Selection

At times it may be necessary to select every node in the current layout Λ. This may be required, for example, when all nodes are to be collectively repositioned, modified, copied, or removed.

Operation: SELECT_ALL ;

Description: This operation is performed by simply adding every node directly contained in the Λ to the select set σ_Λ as follows:

```
$\forall v \mid v \rightarrow p, p \in P, v \in V$ do
    SELECT ($v$, $\Lambda$);
end;
```

Analysis: Since $|P| <= |V|$ and SELECT can be performed in constant time, SELECT_ALL is $\Theta(|V|)$.

Global Deselection

Just as it may be necessary to select every node in the current layout Λ, it may also be required to deselect every node in Λ described as follows:

Operation: SELECT_NONE ;

Description: This operation is performed by simply removing every node directly contained in the Λ from the select set σ_Λ as follows:

```
$\forall v \mid v \rightarrow p, p \in P, v \in V$ do
    DESELECT ($v$, $\Lambda$);
end;
```

Analysis: For the same reasons give with SELECT_ALL, SELECT_NONE is also $\Theta(|V|)$.

Note that both SELECT_ALL and SELECT_NONE can actually be performed in constant time via the alternative sequences, respectively:

$$\sigma_\Lambda := V;$$

and

$$\sigma_\Lambda := \varnothing;$$

Inverse

While it is often desired to select nodes with desired properties, it is similarly often desired to select only those nodes that do not possess those properties. This could be accomplished by defining complementary operations for every select operation in this section. Instead, a single inverse selection operation is provided for this purpose.

Operation: SELECT_INVERSE ;

Description: This operation constructs the complement of the set σ_Λ as follows:

```
$\forall v \mid v \rightarrow p, p \in P, v \in V$ do
    if $v \in \sigma_\Lambda$ then
        DESELECT ($v$, $\Lambda$);
    else
        SELECT ($v$, $\Lambda$);
    end;
end;
```

Alternatively state,

$$\sigma_\Lambda := V - \sigma_\Lambda;$$

Analysis: As above, SELECT_INVERSE is $\Theta(|V|)$.

Root Selection

In many systems, particularly software systems, there exists at least one element that may be characterized as the main or root element(s) of the system. In order to understand the structure of these systems, it is often advantageous to identify these elements first. With very large systems, however, the ability to discern these elements is very cumbersome, again requiring automated means. The SELECT_ROOT operation serves this purpose by selecting only those nodes that are not a dependent of another node contained within the current layout Λ.

Operation: SELECT_ROOT ;

Description: The SELECT_ROOT operation examines all node pairs within the current layout Λ and looks for every node v that is not associated with any edge of the form (w, v) within Λ.

```
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P$ do
    ROOT := TRUE;
    $\forall w \mid w \in V, w \rightarrow p_w, p_w \in P$ do
        if $(w, v) \in E$ then
            ROOT := FALSE;
            exit;
        end;
    end;
    if ROOT then
        SELECT $(v, \Lambda)$;
    end;
end;
```

Analysis: Since SELECT_ROOT examines every node pair (v, w) within Λ, it is $\Theta(|V|^2)$. With access to an inverse node adjacency list (i.e. *inv-adj*$(v) = w$ if $(w, v) \in E$), the inner loop can be simplified to a test for incoming edges on v. The existence of such an edge (i.e. *inv-adj*(v) is non-empty) would immediately indicate v is not a root. Hence, the entire operation could be reduced to $\Theta(|V|)$.

Leaf Selection

Just as it is important to identify a system's root elements, so to may it be useful to identify its leaf element. In contrast to a root element, a leaf element is one which has no dependents. Knowledge of leaf elements is helpful in the construction of bottom-up composite structures as they can be collapsed with their parent elements without introducing any additional dependencies. In top-down analysis, leaf elements also represent the finest level of granularity for the system. From an abstraction perspective, leaf elements are at the lowest level with all other elements in the system based upon them. Hence, the ability to locate these fundamental building blocks can be quite useful. With no other dependencies, they are perhaps the simplest conceptual elements of the system even though their implementation could the most complex.

Operation: SELECT_LEAVES ;

Description: The SELECT_LEAVES operation examines all node pairs within the current layout Λ and looks for every node v that is not involved in any edge of the form (v, w) within Λ.

```
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P$ do
    LEAF := TRUE;
    $\forall w \mid w \in V, w \rightarrow p_w, p_w \in P$ do
        if $(v, w) \in E$ then
            LEAF := FALSE;
            exit;
        end;
    end;
    if LEAF then
        SELECT $(v, \Lambda)$;
    end;
end;
```

Analysis: For the same reason given above for SELECT_ROOT, SELECT_LEAVES is also $\Theta(|V|)$ given access to a node adjacency list where a non-empty $adj(v)$ would immediately indicate that v is not a leaf. The operation is $\Theta(|V|^2)$ otherwise.

In practice, it might also helpful to identify those leaf elements which are associated with one and only one dependency. As an aid to structural understanding, these singular elements can often be removed from the layout or "filtered" from view to help simplify the understanding without loss of essential dependency information. Note also from this discussion that a node that is not associated with any edge in the layout is both a root and a leaf element.

Body Selection

Stripped of its leaf and root elements, the remaining interdependent elements represent the most difficult portion of a system to unravel. An operation to extract the body of system is therefore provided.

Operation: SELECT_BODY ;

Description: The SELECT_BODY operation examines all node pairs within the current layout Λ and looks for every node v that is not involved in any edge of the form (w, v) or (w, v) within Λ. Any node that is both a parent and a child of another node is included in the body selection.

```
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P$ do
    PARENT := FALSE;
    CHILD  := FALSE;
    $\forall w \mid w \in V, w \rightarrow p_w, p_w \in P$ do
        if $(v, w) \in E$ then
            PARENT := TRUE;
        end;
        if $(w, v) \in E$ then
            CHILD := TRUE;
        end;
        if PARENT and CHILD then
            SELECT $(v, \Lambda)$;
            exit;
        end;
    end;
end;
```

Analysis: With access to both an adjacency list *adj*(v) and an inverse adjacency list *inv-adj*(v), SELECT_BODY is also $\Theta(|V|)$, $\Theta(|V|^2)$ otherwise.

Note that a SELECT_ROOT, SELECT_BODY, and SELECT_LEAVES operation sequence is identical to a single SELECT_ALL operation. Similarly, a SELECT_ROOT and SELECT_LEAVES sequence is identical to a SELECT_BODY operation followed by a SELECT_INVERSE operation.

Component Selection

A common technique used in the initial stages of many graph layout approaches is to identify the various connected components of a graph. To define this operation, the notion of a graph *path* must first be reviewed. A path from node v to node w in the graph $G = (V, E)$ is a sequence of edges (v_1, v_2), (v_2, v_3), (v_4, v_5), ..., (v_{k-1}, v_k) such that $v_1 = v$, $v_k = w$, and each node $v_i \in V$ where $i \in \{1, 2, 3, \ldots, k\}$ is a distinct node in V. A graph component (i.e. a subgraph of G) is connected then if for every pair of nodes v and w in the component, there exists a path from v to w.

In standard directed graph terminology, the distinction is usually made between *strongly* connected and *weakly* connected depending upon whether or not the direction of each edge is taken into consideration. A directed graph $G = (V, E)$ is weakly connected if, for each pair of nodes v and w, there exists a sequences of nodes $v_1, v_2, v_3, \ldots, v_k$ such that $v_1 = v$, $v_k = w$, and for $i = 1, 2, \ldots, k\text{-}1$ either $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$. In contrast, the graph G is strongly connected if, for each pair v and w, there is a path from v to w.

Within the context of the configuration model, it is desirable to identify the connected components that exist within the current layout Λ. To relate this to our selection discussion, the SELECT_WEAK and SELECT_STRONG operations described below use the selection set σ_Λ to determine which nodes are to be used to generate the desired components. In essence, a union of connected graph components are generated be these operations. If a specific weakly or strongly connected component for a particular node v in Λ is sought, the selection set σ_Λ should be equated to that node. The primary advantage of this approach is that it allows the one or more components to be located simultaneously within a single visualization space using familiar editing type operations.

Operation: SELECT_WEAK ;

Description: The operation sequence described here is based on a variant of the breadth-first search algorithm. Each node in the selection set σ_Λ is loaded into a queue. Nodes are then removed from the queue and examined for edges to other nodes in Λ. This task is readily accomplished through the maintenance of an edge adjacency list as described early on in Example 1.1. Any new nodes discovered are marked as selected and added to the search queue. The sequence terminates when the queue is empty signifying that all reachable nodes have been searched.

```
QUEUE ← $\sigma_\Lambda$;
while QUEUE $\neq \varnothing$ do
    $v$ ← QUEUE;
    $\forall w \mid w \in V,\ w \rightarrow p_w, p_w \in P, ((v, w)$ or $(w, v) \in E)$ do
        if $w \notin \sigma_\Lambda$ then
            QUEUE ← $w$;
            SELECT ($w, \Lambda$);
        end;
    end;
end;
```

Analysis: Since $|\sigma_\Lambda| \leq |V|$, its possible that the every node in the V may need to be searched or the entire edge list E may need to be examined. Hence, SELECT_WEAK is $\Theta(\max(|E|, |V|))$.

Operation: SELECT_STRONG ;

Description: The strongly connected component selection process is considerably more complex than the weakly connected case due to the requirement that a directed path exist between every pair of distinct nodes within the component. The approach taken here is a modified version of the strongly connected component algorithm presented in [Ah'74]. Unlike the Aho-Hopcroft-Ullman (or AHU) algorithm, the SELECT_STRONG operation discussed here must take into consideration the use of selection set σ_Λ and the fact that Λ may not contain all nodes in V. In this approach, only those nodes which are contained in Λ are initially marked. Similarly, only those nodes which are contained in σ_Λ are passed to the SEARCH routine from the outer loop.

```
COUNT := 1;
STACK := $\varnothing$;
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P$ do
    MARKS := MARKS $\cup$ { $v$ };
end;
NODES := $\sigma_\Lambda$;

$\forall v \mid v \in$ NODES do
    if $v \in$ MARKS then
        SEARCH ( $v$ );
    end;
end;
```

The SEARCH (v) portion of the operation is again similar to the AHU algorithm with the exception of the completion section. Rather than simply printing the names of strongly connected component elements, the nodes are saved in a set until the final node in the component has been identified. If any of the nodes in this set are contained in σ_Λ, all of the nodes in this set will be selected in the current layout Λ.

```
MARKS := MARKS - { $v$ };
DFNUM[$v$] := COUNT;
COUNT := COUNT + 1;
LOWLINK[$v$] := DFNUM[$v$];
STACK ← $v$;

$\forall w \mid w \in V, w \rightarrow p_w, p_w \in P, (v, w) \in E$ do
    if $w \in$ MARKS then
        SEARCHC ($w$);
        LOWLINK[$v$] := min(LOWLINK[$v$], LOWLINK[$w$]);
    else
        if DFNUM[$w$] < DFNUM[$v$] and
            $w \in$ STACK then
            LOWLINK[$v$] := min(DFNUM[$w$], LOWLINK[$v$]);
        end;
    end;
end;

if LOWLINK[$v$] = DFNUM[$v$] then
    COMP := $\varnothing$;
    loop
        $w$ ← STACK;
        COMP := COMP $\cup$ { $w$ };
        exit when $v = w$;
    end;

    if COMP $\cap$ NODES $\neq \varnothing$ then
        NODES := NODES - COMP;
        $\forall w \mid w \in$ COMP do
            SELECT ( $w, \Lambda$ );
        end;
    end;
en100
```

<u>Analysis</u>: The modifications made to the basic AHU algorithm were worst case $\Theta(|V|)$. As a direct result of Theorem 5.4 in [Ah'74], SELECT_STRONG is still $\Theta(\max(|V|, |E|))$.

Note that in modern software systems built using structured languages such as Ada, Pascal, and Modula-II, the ability to identify strongly connected components given a selection σ_Λ does not typically yield interesting results. Unless one or more of the nodes specified in the σ_Λ are part of a cyclic graph component contained in Λ, the selection set σ_Λ will remain unchanged at the completion of the operation. A more specialized operation which looks specifically for cyclic components is therefore introduced.

Cyclic Selection

In certain system organizations, there may exists a sequence of element dependencies of the form $m_1 \rightarrow m_2$, $m_2 \rightarrow m_3$, $m_3 \rightarrow m_4$, ... $m_n \rightarrow \mathrm{m}_1$ with $n \geq 1$. While this organization is relatively uncommon in software systems with strong top-down and bottom-up development constructs, they do arise frequently in systems employing concurrent operations or parallel execution. Since cyclic organizations seldom possess an explicit hierarchical dependency structure, layout generation that aids understanding can be quite difficult. The resulting visualizations of non-hierarchical systems must therefore capture these peer-to-peer type architectures, constraining the cyclically dependent components to a particular layer or composite node within the configuration. An operation that identifies these cyclic components, allowing them to be aggregated and separately analyzed, is thus very useful.

The algorithm for locating cyclic components is very similar to that used for determining the strongly connected components of a graph. Unlike the SELECT_STRONG operation which makes use of the selection set σ_Λ, the SELECT_CYCLIC operation is used to find the union of all nodes in Λ which are involved in a cycle of length one or greater. In an acyclic directed graph, each node forms its own strongly connected component. In such a case, the SELECT_CYCLIC operation would leave the selection set σ_Λ unchanged. If the graph does contain a cycle, the SELECT_STRONG operation would only add the nodes involved in the cycle to the selection set if one or more of the nodes where contained in σ_Λ at the start of the cycle. Hence, the SELECT_CYCLIC operation is suited for finding all cycles while the SELECT_STRONG operation is suited for examining cycles that may be associated with one or more specified nodes.

Operation: SELECT_CYCLIC ;

Description: The SELECT_CYCLIC operation is very similar to SELECT_STRONG, sharing much of the same algorithm. Unlike SELECT_STRONG, however, SELECT_CYCLIC is directed at the entire layout contents and not just those nodes contained in σ_Λ. The outer portion of the operation is therefore re-formulated, more closely resembling the AHU algorithm, as follows:

```
COUNT := 1;
STACK := $\varnothing$;
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P$ do
    MARKS := MARKS $\cup$ { $v$ };
end;

while $\exists v \mid v \in$ MARKS do
    SEARCH ( $v$ );
end;
```

The second portion of the SEARCH algorithm must also be modified to select only those connected components that contain two or more nodes. Without this modification, the entire graph would be selected each time since a single node not involved in a cyclic component is itself a strongly connected component. The updated sequence is as follows:

```
if LOWLINK[$v$] = DFNUM[$v$] then
    $w \leftarrow$ STACK;
    if $v \neq w$ then
        loop
            SELECT ( $w$, $\Lambda$ );
            exit when $v = w$;
            $w \leftarrow$ STACK;
        end;
    end;
end;
```

Analysis: With relatively few modifications from the SELECT_STRONG operation, SELECT_CYCLIC is also $\Theta(\max(|V|, |E|))$.

Relationship Selection

In very large system dependency graphs, it is very difficult to examine more than just a few relationships at a time. To gain an understanding of an unfamiliar system, it is often beneficial to start with a few critical elements and explore those relationships which directly apply to these elements. In the later stages of the software life cycle, for example, a system maintainer may be asked to make enhancements to a particular group of software modules. In order to initiate these modifications, it is important that the software maintainer understand how these modifications will affect other modules in the system. A method for identifying these closely related system elements is therefore in order.

Three relationship selection operations are described here. The SELECT_RELATIVE operation is used to select all those nodes which are dependents, either directly or indirectly, of the specified set of nodes, σ_Λ (i.e. all of the children of σ_Λ). Conversely, the SELECT_ABSOLUTE operation is used to select all those nodes that dependent upon the specified set of nodes (i.e. all of the parents of σ_Λ). Lastly, the operation SE-

LECT_REFERENCED selects both the dependent nodes and the nodes dependent upon the selection set σ_Λ (i.e. all of the nodes that are both parents and children of σ_Λ).

Operation: SELECT_RELATIVE ;

Description: As formulated, the SELECT_RELATIVE operation is identical to the SELECT_WEAK operation described above. Both operations use the selection set σ_Λ to initially direct their searches.

Analysis: SELECT_RELATIVE is $\Theta(\max(|E|, |V|)$ as described above.

Operation: SELECT_ABSOLUTE ;

Description: The SELECT_ABSOLUTE operation is essentially the same operation as SELECT_RELATIVE and SELECT_WEAKLY_CONNECTION but traverses graph edges in the reverse direction. The following statement in the SELECT_RELATIVE operation

$$\forall w \mid w \in V, (w, v) \in E, w \rightarrow p_w, p_w \in P \text{ do}$$

replaces the statement

$$\forall w \mid w \in V, (v, w) \in E, w \rightarrow p_w, p_w \in P \text{ do}$$

in the SELECT_ABSOLUTE operation.

Analysis: SELECT_ABSOLUTE is again $\Theta(\max(|E|, |V|)$ as above.

Operation: SELECT_REFERENCED ;

Description: This operation combines both the SELECT_RELATIVE and SELECT_ABSOLUTE operations into one, traversing edges from the selection set σ_Λ in both directions.

<u>Analysis</u>: Since a combined version of the relative and absolute selection operations still requires a single traversal of the node and/or edge sets, SELECT_REFERENCED is also $\Theta(\max(|E|, |V|))$.

Note that a SELECT_RELATIVE followed by a SELECT_ABSOLUTE is not equivalent to a single SELECT_REFERENCED operation as the selection set σ_Λ most likely will have changed after the initial SELECT_RELATIVE operation. A SELECT_BODY operation followed by a SELECT_REFERENCED operation, however, is equivalent to a SELECT_ALL operation. Similarly, a SELECT_ROOT and SELECT_LEAVES sequence followed by a SELECT_REFERENCED operation is also equivalent to a SELECT_ALL operation.

Attribute Selection

The next set of operations focus on the attributes associated with nodes and edges contained in the current layout Λ. Of particular importance is the *name* attribute. As identified in [Wa'81], naming is a crucial factor in modern system design. To aid the design process, a collection of naming standards are often adopted in order to help organize a system's elements and identify their purpose and function. A common operation in complex system analysis is to locate a specific set of elements given their naming characteristics so that their relationships can be carefully examined. The *name* attribute presented in Section 2.6 provides the necessary mechanism for identifying individual nodes. In large systems implementations, the seemingly simple task of locating a single node with the desired *name* attribute value can be a struggle if done manually, requiring sifting through hundreds of nodes. Furthermore, the exact system element names may not be known ahead of time. Consequently, a flexible node find operation is in order. The SELECT_NAME operation presented here serves this purpose.

Operation: SELECT_NAME (*name-expression*);

Description: The SELECT_NAME operation can be performed via a simple linear search of the node *name* attribute list. A typical search-expression implementation would provide wild card type string matching such as the use of the "%" character for single character substitutions and the "*" character for multiple character substitutions.

```
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P$ do
    if MATCH ( NAME($v$), name-expression) then
        SELECT ( $v$, $\Lambda$ );
    end;
end;
```

Analysis: Since the SELECT operation is restricted to the single layer Λ, a simple linear search through an unordered node list V can be performed in at most $\Theta(|V|)$. Assuming that the *name-expression* parameter represents a regular expression and that the lengths of both *name-expression* and all name attributes are bounded by constants s and t, respectively, a suitable $\Theta(\max(s, t))$ parser can be constructed for the MATCH operation using the techniques described in [Ho'79]. Since $\max(s, t)$ is generally $\ll |V|$, SELECT_NAME is $\Theta(|V|)$. This complexity can be reduced to $\Theta(\log |V|)$ via binary search techniques using an ordered node name list instead, although an additional computation investment would have to be made in the CREATE_NODE, INSERT_NODE, or SET_NODE_ATTRIBUTE operations required to build and maintain this list.

A particularly useful enhancement to the SELECT_NAME operation is the ability to apply multiple find expressions that have been previously recorded in an external file. Within a particular software development environment, there are frequently collections of tools that are used repeatedly by different developers across systems. The ability to construct and retain these frequently referenced items for later can greatly ease the burden associated with the initial analysis of an unfamiliar system.

Operation: SELECT_PATTERN (*file_specification*) ;

Description: The SELECT_PATTERN operation is equivalent to a series of FIND operations with multiple search expressions specified via an external input source. The selection set σ_Λ is updated with the union of each FIND operation's results.

Analysis: If m expressions are defined in the specified input source, the result operation will be $\Theta(m \times |V|)$ due to the $\Theta(|V|)$ complexity of the SELECT_NAME operation.

Similar to a SELECT_NAME operation, it is frequently helpful to locate all those nodes or edges in a configuration that possess a specified set of attributes. In software systems, for example, locating all foreign standard nodes would be a common operation performed by a operating system interface programmer while identifying all aggregate packages would be of particular importance to a software librarian. Locating elements linked with a particular edge attribute set would be of similar use. The SELECT_NODE_ATTRIBUTES and SELECT_EDGE_ATTRIBUTES operations provide this functionality.

Operation: SELECT_NODE_ATTRIBUTES (*attributes*) ;

Description: The SELECT_NODE_ATTRIBUTES operation performs a simple linear search through the node list associated with the current layout Λ and selects those nodes that match all of the the specified node attributes as follows:

```
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P$ do
    if $attributes \cap \varpi_v = attributes$ then
        SELECT_NODE ( $v, \Lambda$ );
    end;
end;
```

Analysis: Similar to the SELECT_NAME operation, SELECT_NODE_ATTRIBUTES is $\Theta(|V|)$.

Operation: SELECT_EDGE_ATTRIBUTES (*attributes*) ;

Description: The SELECT_EDGE_ATTRIBUTE operation performs a simple linear search through the edge list associated with the current layout Λ and selects those nodes associated with edges that match all of the the specified edges attributes as follows:

```
$\forall e \mid e = (v, w) \in E, v \rightarrow p_v, p_v \in P, v \rightarrow p_w, p_w \in P$ do
    if $attributes \cap \varepsilon_e = attributes$ then
        SELECT_NODE ( $v, \Lambda$ );
        SELECT_NODE ( $w, \Lambda$ );
    end;
end;
```

Analysis: The SELECT_EDGE_ATTRIBUTES is again similar to the SELECT_NAME operation except that a search through the edge list is now required yielding $\Theta(|E|)$.

Note that both the node and edge attribute selection operations could be further refined to search based on more elaborate node/edge attribute matching criteria such as the regular expression parameter supplied with the FIND operation.

Degree Selection

In many applications, the maximum number of times any element in a system is referenced (referred to as *fan-in*) or the maximum number of references made by any element in a system (commonly referred to as *fan-out*) have often been used as measures of the system's complexity [He'89]. While measures such as these can frequently be misleading [Sh'88], they do provide some interesting insight into the organization of a system. Those elements which exhibit high fan-in or fan-out measures may represent an important centralized resource critical to the understanding of the system. Alternatively, they may represent standard elements that by virtue of the operating environment are required. Text input/output packages, common error handling routines, message logging services, etc. are examples frequently encountered in large software system design. A collection of operations to locate these elements in a complex system organization is therefore very helpful.

The three operations described here examine what is commonly referred to as the *degree* or number of dependencies associated with the node. For the purpose of this discussion, in-degree is the measure of the number of incoming edges or nodes that are dependent upon a particular node. Similarly, the out-degree of a node is the number of out-going edges or nodes that are dependents of a particular node. Finally in-out-degree is a measure of the some of both incoming and outgoing edges. The following operations select the nodes in the current layout L that exhibit the highest values of each of these three measures, respectively:

Operation: SELECT_MAX_IN ;

Description: The SELECT_MAX_IN operation must scan the node list V and examine each dependency associated with the current layout Λ, maintaining a maximum value and a list of elements which this value applies to. At the completion of the sequence, those elements with the highest count are selected.

```
MAX := 0;
DEGREE := $\varnothing$;
$\forall v \mid v \in V$ do
    COUNT := 0;
    $\forall e \mid e = (v, w) \in E, v \rightarrow p_v, p_v \in P, v \rightarrow p_w, p_w \in P$ do
        COUNT := COUNT + 1;
    end;
    if COUNT > MAX then
        DEGREE := { $v$ };
    else if COUNT = MAX then
        DEGREE := DEGREE $\cup$ { $v$ };
    end;
end;
$\forall v \mid v \in$ DEGREE do
    SELECT_NODE ( $v$, $\Lambda$ );
end;
```

Analysis: Since the sequence must possibly iterate through the configuration's entire node list and/or edge adjacency list, SELECT_IN is $\Theta(\max(|V|, |E|))$.

Operation: SELECT_MAX_OUT ;

Description: SELECT_MAX_OUT is identical to SELECT_MAX_IN with the exception of the following statements:

```
$\forall e \mid e = (w, v) \in E, v \rightarrow p_v, p_v \in P, v \rightarrow p_w, p_w \in P$ do
    COUNT := COUNT + 1;
end;
```

Analysis: $\Theta(\max(|V|, |E|))$.

Operation: SELECT_MAX_IN_OUT ;

Description: SELECT_MAX_IN_OUT is identical to SELECT_MAX_IN with the exception of the following statements:

```
$\forall e \mid e = (v, w) \in E, v \rightarrow p_v, p_v \in P, v \rightarrow p_w, p_w \in P$ do
```

```
        COUNT := COUNT + 1;
    end;
    $\forall e \mid e = (w, v) \in E, v \rightarrow p_v, p_v \in P, v \rightarrow p_w, p_w \in P$ do
        COUNT := COUNT + 1;
    end;
```

Analysis: $\Theta(\max(|V|, |E|))$.

A natural refinement to this set of operations would be the ability to locate nodes which possess a specific value for in-degree, out-degree, or in-out-degree. Note that all nodes selected by the SELECT_ROOT operation have an in-degree of 0 and that all nodes selected by the SELECT_LEAVES operations have an out-degree also of 0.

3.4 Arrangement

This section examines numerous operations for generating configurations with specific properties. Using the editing operations from Section 3.2 and the selection operations from Section 3.3, a user could manipulate a configuration manually, moving nodes by hand hoping to obtain a reasonable organization by trial and error. Such a process becomes grossly inadequate, however, as the number of nodes to be examined climbs much beyond a few dozen. Fortunately, there are number of arrangement operations that have been proven to be quite useful in graph layout applications. When used in unison with manual manipulation techniques, a powerful system set of configuration browsing services emerges. These services provide a user with a great deal of flexibility when exploring complex structures. A survey of the more popular operations and their uses is given here.

Default Arrangement

At the beginning of an examination of an unfamiliar system, a user will typically have little knowledge of the system's internal organization. Yet, in order to unravel its structure, an initial starting point must first be defined. Often the only structural knowledge initially available about a system may be just a list of its elements. By identifying these components, some sort of configuration can be initially generated, providing at the very least, a coarse layout which can then be substantially improved. The ARRANGE_DEFAULT operation is provided for this purpose.

<u>Operation:</u> ARRANGE_DEFAULT ;

<u>Description:</u> The ARRANGE_DEFAULT generates a planar layout for all elements that have been inserted (via the INSERT_NODE operation) into the current layout Λ. This operations assume a simple rectangular type grid for the position of each element. The z coordinate of the visualization space is assumed to be constant. The horizontal and vertical spacing between each node, δ_x and δ_y, respectively, is likewise considered constant. The following sequence defines the ARRANGE_DEFAULT operation:

```
$x := 0;$
$y := 0;$
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P_\Lambda$ do
    SET_NODE_POSITION ($v, \Lambda, (x, y)$ );
    $x := x + \delta_x;$
    If $x > \text{width}(S_\Lambda)$ then
        $x := 0;$
        $y := y + \delta_y;$
    end;
end;
```

<u>Analysis:</u> Since this sequence involves only a single iteration through the node set V and SET_NODE_POSITION can be performed in constant time, ARRANGE_DEFAULT is $\Theta(|V|)$.

Hierarchical Arrangement

Hierarchical relationships are very common in systems. Such relationships are characterized by a arrangement in order of rank, grade, level, etc. In modern software systems, an abstraction hierarchy can be derived by examining the object class or type structure and determining how they are used to synthesize new, higher level or higher rank classes or types. This relationship is easily captured within the directed graph framework. A hierarchical relationship between two elements a and b are represented via the dependency a $\rightarrow$ b and its corresponding edge (a, b) in the graph G.

Because of the prevalence of hierarchical structures, an operation which takes an arbitrary configuration and imposes a layout which reveals this structure is in order. This can be accomplished by arranging all nodes such that given any pair of nodes (a, b), if a is dependent upon b, then a will be arranged hierarchically to b. A precise definition of hierarchical arrangement will be given in Section 5.3, but for now it will suffice to say that (a, b) is hierarchically arranged if a appears "above" b in the layout.

Note that not all systems possess a hierarchical arrangement. Any system that contains a strongly connected component with more than one node (i.e. contains a cycle) can not be readily hierarchically arranged since $a_1 \rightarrow a_2$, $a_2 \rightarrow a_3$, $a_3 \rightarrow a_4$, ..., $a_k \rightarrow a_1$. In order to generate a hierarchical layout, therefore, the layout must first be in reduced form. That is, the layout must not contain any strongly connected component with more than one node. A technique for reducing cyclic structures will be presented in Section 3.5. The ARRANGE_HIERARCHICAL operation is defined as follows:

Operation: ARRANGE_HIERARCHICAL ;

Description: The ARRANGE_HIERARCHICAL operation takes an arbitrary layout and repositions nodes until a strict hierarchical arrangement is imposed between all pairs of dependent nodes. The operation first verifies that the layout has been reduced via the function REDUCED which can be easily constructed from the SELECT_STRONG operation defined above in Section 3.3. The algorithm then starts at the "top" of the visualization space and moves nodes "downward" by a constant distance c_y until a hierarchy is imposed between all dependent pairs. The function $\max_y$ returns a node whose y component of its position vector is greater than or equal to the y component all other nodes in Λ. The following sequence describes this operation:

```
if REDUCED($\Lambda$) then
    $H := \emptyset$;
    $\forall v \mid v \in V, v \rightarrow p_v, p_v \in P_\Lambda$ do
        $H := H \cup \{ (v, p_v) \}$;
    end;

    while $H \neq \emptyset$ do
        $(v, p_v) := \max_y(H)$;
        $\forall (w, p_w) \mid (w, p_w) \in H$ do
            if $(w, v) \in E$ then
                if $p_w(y) \geq p_v(y)$ then
                    $p_w(y) := p_v(y) - c_y$.
                    SET_NODE_POSITION ($v, \Lambda, p_w$);
                end;
            end;
        end;
    end;
end;
```

Analysis: As discussed in Section 3.3, the REDUCED function can be performed in $\Theta(\max(|V|, |E|)$. In the worst case where the layout is hierarchially arranged, but up-side-down, every pair of nodes in Λ may have to be examined. Hence, ARRANGE_HIERARCHICAL is $\Theta(|V|^2)$.

Because of the utility of hierarchical arrangements, a special purpose operation for generating a hierarchical arrangement without regard to original layout is quite useful. The ARRANGE_DEPENDENT operation performs this task and may be considered an alternative to the ARRANGE_DEFAULT operation for non-cyclic layouts. ARRANGE_DEPENDENT is defined as follows

Operation: ARRANGE_DEPENDENT ;

Description: The ARRANGE_DEPENDENT operation must again verify that the layout is in reduced form via the REDUCED function to insure that the sequence will terminate. The operation first builds a set D of all nodes contained in Λ. It then proceeds to find nodes which are not dependents of any other node in D. These nodes are added to a queue so that they may be subsequently positioned at the appropriate location in the visualization space. Once positioned, these elements are removed from D and the x and y coordinates are reset for the next iteration. The z coordinate of each node is unchanged. The process repeats until no remaining nodes are in D.

```
$y := \max_y(S_\Lambda)$;
if REDUCED($\Lambda$) then
    $D := \varnothing$;
    $\forall v \mid v \in V, v \rightarrow p_v, p_v \in P_\Lambda$ do
        $D := D \cup \{ v \}$;
    end;

    while $D \neq \varnothing$ do
        $\forall v \mid v \in D$ do
            if $\forall w \mid w \in D, (w, v) \notin E$ then
                QUEUE $\leftarrow v$;
            end;
        end;

        $x := 0$;
        while QUEUE $\neq \varnothing$ do
```

```
                $v \leftarrow$ QUEUE;
                SET_NODE_POSITION ($v$, $\Lambda$, ($x$, $y$, $p_v(z)$) );
                $x := x + c_x$;
                $D := D - \{ v \}$;
            end;
            $y := y - c_y$;
        end;
    end;
```

Analysis: Similar to ARRANGE_HIERARCHICAL, ARRANGE_DEPENDENT requires examining each pair of nodes in Λ, resulting in $(|V|^2)$.

Layered Arrangement

Another very powerful concept for organizing complex systems is the use of layering, described in detail in [IS'82]. The layering concept involves the decomposition of a system into a series of discrete levels of abstract where each level utilizes the resources and services provided by a well-defined interface to the the layer beneath, to provide access to an integrated set of resources and potentially more powerful services to the layer directly above, again through a well-defined interface. The use of layering can be used as a model to guide a system's design as well as its implementation.

In a strictly layered system, all elements are dependent only on other elements residing at the same level in the system or on elements contained within the layer directly below. Conversely, all elements are dependents only of other elements residing at the same level or of elements contained within the layer directly above. While strict layering methods are not a prerequisite for good system organization, general adherence can provide significant aid to clearer understanding of very complex structures similar to other conventions and standards.

To capture the layering concept within the configuration framework defined here, space within the visualization space associated with each layout need only be designated for this purpose. This can be accomplished by defining a series of discrete position vector ranges for each level of the layering structure. Alternatively, a simple calculation with a position vector can be used to determine its level. While a more formal description will be given in Section 5.3, for now it will suffice that given a coordinate y, its layer number can be determined by an integer divide with a constant, y **div** c. The following operation is used to generate a strictly layered organization given an arbitrary layout:

<u>Operation:</u> ARRANGE_LAYERED ;

<u>Description:</u> The ARRANGE_LAYERED operation scans the visualization space, working from the highest levels (y >> 0) down to the lowest level (y ≅ 0). The nodes at each level are examined. Every node dependent upon any node contained at the current level being scanned is moved to this level. Similarly, every node that is a dependent of any node at the current scan level that is not contained within this level or the level directly below, is moved to the level directly below. The process repeats for each level until the lowest level (y **div** $c = 0$) is completed.

```
$\forall l \mid l \in \{ 1 .. (\max_y(S_\Lambda)$ div $c) \}$ in reverse do
    $\forall v \mid v \in V, v \rightarrow p_v, p_v \in P_\Lambda$ do
        if $(p_v(y)$ div $c) = l$ then
            QUEUE $\leftarrow v$;
        end;
    end;

    while QUEUE $\neq \emptyset$ do
        $v \leftarrow$ QUEUE;
        $\forall w \mid w \in V, w \rightarrow p_w, p_w \in P_\Lambda$ do
        if $(w, v) \in E$ then
            if $(p_w(y)$ div $c) < l$ then
                $p := (p_w(x), p_v(y), p_w(z))$;
                SET_NODE_POSITION $(w, \Lambda, p)$;
                QUEUE $\leftarrow w$;
            end;
            else if $(v, w) \in E$ then
                if $(p_w(y)$ div $c) < l-1$ then
                    $p := (p_w(x), p_v(y) - c, p_w(z))$;
                    SET_NODE_POSITION $(w, \Lambda, p)$;
                end;
            end;
        end;
    end;
```

Analysis: As with the two previous operations, ARRANGE_LAYERED is again $\Theta(|V|^2)$ since it must examine every combination of node pairs (v, w) contained in the Λ.

Breadth-First Arrangement

When first examining an unfamiliar system or set of subsystem elements, it is frequently instructive to first locate a root level set of elements (i.e. nodes with no parents), determine their dependents, and examine those dependents, then in turn determine the dependents of those elements, examine their dependents, etc., repeating the process until all elements in the system have been examined. This process essentially mimics a breadth-first search of the system's dependency graph.

Due to the important relationships the breath-first search may reveal, it is convenient to be able to specify any subset of system elements as a starting point. The selection set σ_Λ can again be used for this purpose. By allowing any set of nodes in Λ to be used as the starting point for the search, it is possible that not all nodes in Λ will be examined. For example, the parents of any nodes that are not children of any of the other nodes in the search will never be examined. As a further aid to understanding, these nodes can be temporarily eliminated from the visualization space so as not to obscure the results of the search. This can be easily accomplished by setting the position vector of these nodes to an undefined value, Φ. Borrowing from the layering concept, the vertical position of each node in the resulting layout is used to reflect the depth of the node in the search. The ARRANGE_BREADTH search is now defined.

Operation: ARRANGE_BREADTH ;

Description: The ARRANGE_BREADTH operation begins by building a queue of all selected nodes, σ_{Λ}, and a set of all nodes in Λ. The operation then removes each of the nodes in the queue, one by one examining each node v and adding any of v's dependents that have not already been positioned to the queue. At the completion of each examination, v is repositioned at its new x, y position, v is removed from B, and the x coordinate is updated for the next node. Once the original set of nodes are examined, the y coordinate is set to the next lower level and the process is repeated until no remaining nodes are left in the queue. The final part of the sets the position of all remaining nodes in B to undefined.

```
$y := \max_y(S_{\Lambda})$;
QUEUE $\leftarrow \sigma_{\Lambda}$;
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P_{\Lambda}$ do
    $B := B \cup \{ v \}$;
end;

$x := 0$;
while QUEUE $\neq \emptyset$ do
    $\forall i$ in $\{1..\text{length(QUEUE)}\}$ do
        $v \leftarrow$ QUEUE;
        $\forall w \mid w \in B$ do
            if $(v, w) \in E$ then
                QUEUE $\leftarrow v$;
                $B := B - \{ v \}$;
            end;
        end;
        SET_NODE_POSITION $(w, \Lambda, (x, y, p_w(z)))$;
        $x := x + c_x$
    end;
    $y := y - c_y$;
```

```
end;

∀$v$ | $v \in B$ do
    SET_NODE_POSITION($v$, $\Lambda$, $\Phi$);
end;
```

Analysis: Using the adjacent list implementation for representing edges, ARRANGE_BREADTH requires a single pass through each element in the node list and each of its edges, resulting in $\Theta(\max(|V|, |E|)$.

Depth-First Arrangement

The other popular graph traversal technique useful in complex system exploration involves the use of depth-first search. The ARRANGE_DEPTH operation is used to generate a layout based on the results of this search. In contrast to the breadth-first operation, the ARRANGE_DEPTH operation is useful in revealing the nesting structure of a system. The position of an element in the resulting layout can be used to gauge the length of the dependency path of the element from the root elements that were specified. This information is particularly useful in determining how far removed an element is from its parent elements which can serve as an indicator of their relative organizational importance to the parent elements.

Although similar in concept, the implementation of ARRANGE_DEPTH is slightly more complex than ARRANGE_BREADTH since layout can no longer be performed by working from the top to the bottom of the visualization space. The ARRANGE_DEPTH operation instead must work "across" the visualization space, recording the maximum horizontal position of each node placed at each level.

Operation: ARRANGE_DEPTH ;

Description: The main body of the ARRANGE_DEPTH operation consists of three parts as shown below. The first part constructs a set D of all elements currently positioned in the Λ. The second part performs the actual depth-first arrangement of the Λ using the recursive procedure PLACE described below. All nodes not positioned at the completion of this part of the sequence are then set to undefined by the third part which scans the remaining nodes in D.

```
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P_\Lambda$ do
    $D := D \cup \{ v \}$;
end;

$\forall v \mid v \in \sigma_\Lambda$ do
    PLACE($v$, $\max_y(S_\Lambda)$);
end;

$\forall v \mid v \in D$ do
    SET_NODE_POSITION($v$, $\Lambda$, $\Phi$);
end;
```

The PLACE operation accepts two parameters, v and y, corresponding to a node and a y-coordinate in the visualization space S_Λ, respectively. PLACE set the position of the node v based on the current level y. All of v's dependents are then examined. If any dependent w has not been placed (i.e. w is a member of D), then w is passed to PLACE to be positioned at the next lower level.

```
$D := D - \{ v \}$;
SET_NODE_POSITION ($v$, $\Lambda$, $(p_v(x), y, p_v(z))$ );
$\forall w \mid w \in D$ do
    If $(v, w) \in E$
        PLACE ($w$, $y - c_y$);
    end;
end;
```

Analysis: As discussed above, the basic depth-first search algorithm is $\Theta(\max(|V|, |E|))$. Again assuming an implementation with access to the node adjacency list $adj(v)$, the PLACE operation will examine each node and edge once. Since SET_NODE_POSITION is $\Theta(c)$, ARRANGE_DEPTH is also $\Theta(\max(|V|, |E|))$.

Horizontal Arrangement

Most of the arrangement operations up to now have paid little attention to the x-coordinate or horizontal position of each node. In fact, at the completion of several of these operations, many of the nodes may actually overlap. This overlap must be addressed to prevent compounding the visual understanding problem. Described below are some simple horizontal positioning operations that have proven useful. Because of the relative simplicity of each operation, a detailed description of each operation will be excluded from the presentation. Since each operation involves a number of linear sweeps across each level of the visualization space, proportional to the number of layers in the visualization, their time complexity will generally be $\Theta(|V|)$ or $\Theta(|E|)$. The analysis of each operation will also be excluded.

Operation: ARRANGE_UNIFORM ([δ]);

Description: The ARRANGE_UNIFORM scans each level of the visualization space starting at the "left-hand" side (i.e. $x = 0$) of the visualization space and moving to the "right-hand" side (i.e. $x >> 0$) adding each node to a queue. Starting again at $x = 0$, the nodes are then removed from the queue in order and repositioned, insuring uniform spacing in between each node. The optional parameter δ is used to specify the distance to be allocated between each node. A suitable default minimal distance would be applied by an actual implementation.

Operation: ARRANGE_CENTERED ([δ]);

Description: The ARRANGE_CENTERED operation is similar to ARRANGE_UNIFORM accept that as nodes are removed from the queue, they are instead alternatingly positioned at a uniformly increasing distance from a vertical centerline $x = \max_x(S_\Lambda)$ in the visualization space. The optional parameter δ is again used to specify the distance to be allocated between each node.

Operation: ARRANGE_LATERAL ([δ]);

Description: One of the difficulties faced with uniformly spaced layouts is that an edge between two nodes on different levels with one or more intervening levels may exactly coincide with another edge whenever edges are drawn strictly as straight lines. The special purpose operation ARRANGE_LATERAL is used to partially compensate for this condition. ARRANGE_LATERAL examines the dependent nodes of each node and whenever it finds a pair that are separated by one or more intervening layers, moves the dependent node either left or right by the amount δ. While ARRANGE_LATERAL may not always yield an optimal result, it is particularly useful in interactive applications where high-performance is required Since ARRANGE_LATERAL must examine each edge it time complexity is $\Theta(\max(|V|,|E|))$.

Operation: ARRANGE_ADJUSTED ([δ]);

Description: This special purpose operation performs a simple useful heuristic arrangement operation involving leaf nodes with only one parent.

ARRANGE_ADJUSTED insures that, if possible, such nodes are positioned directly beneath their parent. This can be accomplished by first positioning all single parent nodes along each level and then uniformly filling in with constant spacing δ all remaining nodes along the level. Since two or more single-parent nodes may have the same parent, it is not always possible to accommodate all such nodes. Since ARRANGE_ADJUSTED must examine each node edge, it is also $\Theta(\max(|V|,|E|))$. This operation is primarily used in at the completion of an optimization sequence (discussed in Chapter 6) to make fine adjustments to the layout that are of sufficiently low energy that the optimization algorithm may not have detected them.

Note that the set of arrangement operations outlined in this section are by no means exhaustive. While the operations presented here address many common dependency visualization issues, there remain numerous questions concerning optimality of the resulting layouts and whether or not they capture the layout criteria of interest. Chapter 5 will lay the framework for making this determination.

3.5 Reduction

The concept of composite layouts was introduced in Section 2.11 as a powerful mechanism for collapsing portions of complex dependency graphs while simultaneously tracking and maintaining the dependency information associated with each composite element. Element consolidation is a tremendous tool for organizing complex structures and controlling visual complexity. Of particular interest is the ability to reduce the number of visible edges by collapsing multiple nodes with numerous interdependencies into a single composite layout. The utility of this operation can be easily demonstrated by re-examining our sample system used throughout the previous examples.

Re-drawing our example system using a centered, dependent arrangement with the node and edge attribute information as provided above, a resulting layout without the use of composite reduction is shown in Figure 3.1.

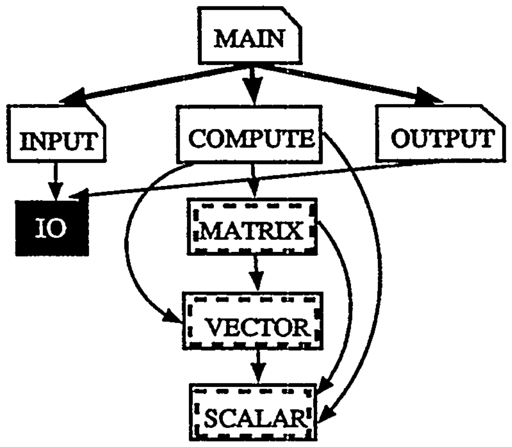


Figure 3.1 Example system without composite reduction

If the element IO is reduced with INPUT and OUTPUT, forming the composites INPUT_IO and OUTPUT_IO, respectively, and the elements MATRIX, VECTOR, and SCALAR are reduced to the single composite element, LINEAR_ALGEGRA, a new resulting layout is shown in Figure 3.2. Note that a total of seven edges have been eliminated from view, but without loss of important information. These dependencies can still be found by examining the contents of INPUT_IO, OUTPUT_IO, and LINEAR_ALGEBRA.

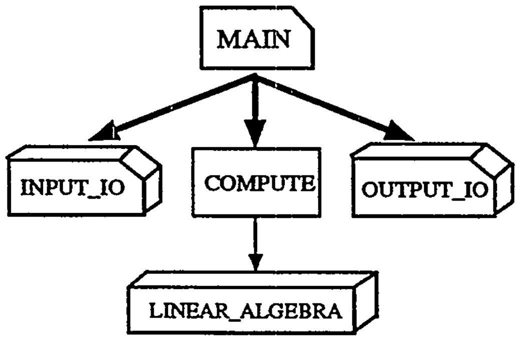


Figure 3.2 Example system with composite reduction

This section presents several useful operations for performing this reduction process. Each operation makes use of either the selection set σ_Λ or the node/edge attribute sets. A typical editing session or configuration manipulation sequence might involve a series of arrangement operations followed by a set of selection operations, followed by a reduction operation. As a result of the operation, one or more new nodes will be generated within the configuration. The attributes of these new nodes and their corresponding edges are assigned via the inheritance rules outlined in Section 2.6 and Section 2.8, respectively, or via manual execution of the the SET_NODE_ATTRIBUTE and SET_EDGE_ATTRIBUTE operations.

Selection

Perhaps one of the most useful reduction operations is the ability to consolidate those elements specified by the selection set σ_Λ. This mechanism allows nodes to be specified both by manual selection (i.e. SELECT_NODE) or via any of the selection operations presented in Section 3.3. All specified nodes are consolidated into a single new node. The name attribute of this new node is left undefined at the completion of this operation and should be set with the SET_NODE_ATTRIBUTE operation. The REDUCE_SELECTED operation is defined as follows:

Operation: REDUCE_SELECTED ;

Description: The REDUCE_SELECTION operation makes extensive use of the operations previously defined. It first creates a new node v and inserts v into the current layout Λ. It then creates a new layout l and binds v to l. Using the editing operations from Section 3.2, all of the nodes in σ_Λ are cut from Λ and pasted into the new layout l. The selection set σ_Λ is reset to v upon completion.

```
CREATE_NODE( $v$ );
SET_NODE_ATTRIBUTE( $v$, $composite$ );
INSERT_NODE ($v$, $\Lambda$);
CREATE_LAYOUT( $l$, ($c_x$, $c_y$, $c_z$) );
CREATE_BINDING( $b$, $v$, $l$);
CUT;
temp := $\Lambda$;
$\Lambda$ := $l$;
PASTE;
ARRANGE_DEFAULT;
$\Lambda$ := temp;
SELECT_NODE( $v$ );
```

Analysis: Since CUT and PASTE are $\Theta(|V| \times |L|)$, ARRANGE_DEFAULT is $\Theta(|V|)$, and all other operations are constant time, REDUCED_SELECTED is $\Theta(|V| \times |L|)$.

Component Reduction

At the start of any complex system analysis session, it is instructive to first reduce the system into its weakly and strongly connected components. If a system contains more than one weakly connected component, the ability to separate multiple components can significantly reduce the complexity of the organizational task by breaking it down into several smaller, independent visualization problems. While this is a wise strategy, their is frequently little gained since even very large systems such as shown in Figure 1.3 may contain only one weakly connected component.

Using typical configuration control mechanisms, common in modern software engineering environments, it is usually possible to separately maintain descriptions of isolated systems. A system which contains n weakly connected components can generally be regarded as n separate systems and which can be maintained independently since there exists no interdependencies. Nevertheless, it is still possible for two or more system descriptions to coexist either unwillingly or unkowningly. A mechanism for reducing these systems into separate components is therefore still in order. The REDUCE_WEAK operation provides this functionality.

Operation: REDUCE_WEAK ;

Description: The operation REDUCE_WEAK is a variant of the SELECT_WEAK operation described above in Section 3.3. While SELECT_WEAK used the selection set σ_Λ, REDUCE_WEAK works with the entire layout, decomposing it into individual components only if more than one component exists within the current layout Λ. REDUCE_WEAK begins by building a set W of all nodes in the current layout Λ. The algorithm looks for an element v in W and uses it to initialize a search queue. The algorithm then removes elements from the queue looking for both parents and children that have not yet been examined. Whenever an unexamined element is encountered, it is added to the search queue, removed from W, and marked as selected. When the search queue is empty, a weakly connected component has been found. If the connect component is the same as the entire layout, the algorithm terminates, otherwise the component is collapsed to a single composite node. The process repeats until all nodes in the set W have been searched.

```
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P_\Lambda$ do
    $W := W \cup \{ v \}$;
end;

count := $|W|$;
while $\exists v \mid v \in W$ do
    SELECT_NONE;
    QUEUE $\leftarrow v$;
    while QUEUE $\neq \emptyset$ do
        $v \leftarrow$ QUEUE;
        SELECT_NODE ($v$, $\Lambda$);
        $\forall w \mid w \in W$, (($v$, $w$) $\in E$ or ($w$, $v$) $\in E$ ) do
            if $w \notin \sigma_\Lambda$ then
                QUEUE $\leftarrow w$;
                $W := W - \{ w \}$;
                SELECT_NODE ($w$, $\Lambda$);
            end;
        end;
    end;
    if count $\neq |\sigma_\Lambda|$ then
        REDUCE_SELECTED;
    else
        exit;
    end;
end;
```

Analysis: As a variant to the weakly-connected component algorithm, the basic sequence is $\Theta(\max(|E|, |V|))$. However, the REDUCE_SELECTED operation may have to be executed as many as $|V|$ times. REDUCE_WEAK is therefore $\Theta(|V|^2)$.

Reduction of a layout into its strongly connected components is useful as it provides a mean for isolating cyclic portions of the layout that contain no inherent hierarchical structure. The ability to collapses these portions of the layout into a composite layout would allow the resulting layout to be treated hierarchically without concern over cyclic dependencies. The cyclic components could then be examined independently within another layout. Operations such as SELECT_MAX_IN and SELECT_MAX_OUT are useful in these instances for locating pivotal elements that potentially introduce the cyclic behavior. As expected the REDUCE_STRONG operation is similar in nature to the SELECT_STRONG operation except that strongly connected components are searched for over the entire layout rather than just from select set σ_Λ.

Operation: REDUCE_STRONG ;

Description: The REDUCE_STRONG is nearly identical to SELECT_CYCLIC except that a REDUCE_SELECTED operation is performed at the completion of each selected component identification. For operational convenience, only those strongly connected components that contain two or more nodes are collapsed. An updated Part II of the SEARCH procedure is listed here:

```
if LOWLINK[$v$] = DFNUM[$v$] then
    $w \leftarrow$ STACK;
    if $v \neq w$ then
        loop
            SELECT ( $w, \Lambda$ );
            exit when $v = w$;
            $w \leftarrow$ STACK;
        end;
        REDUCE_SELECTED;
        SELECT_NONE;
    end;
end;
```

Analysis: REDUCE_STRONG is also $\Theta(|V|^2)$ as a result of the $\Theta(|V| \times |L|)$ time complexity introduced by the REDUCE_SELECTED operation.

Node Attribute Reduction

The use of node attributes was originally introduced in Section 2.6 as an aid to visual understanding. As seen with the selection operations in Section 3.3 and to be explored further with regards to configuration metrics in Section 5.4, attributes can serve many other purposes, both in the manipulation and in the evaluation of a configurations. A general purpose reduction operation that consolidates based on the matching of a particular set of attributes could prove quite useful. The abilit to collect, for example, all *foreign*, *standard* system elements into a single composite node so that the system's external interfaces can be examined is often of frequent interest. To provide this functionality, a simple node attribute reduction operation based on SELECT_NODE_ATTRIBUTES can be defined as follows:

Operation: REDUCE_ATTRIBUTES (*attributes*) ;

Description: The operation REDUCE_ATTRIBUTES accepts a parameter *attributes* $\subseteq \varsigma$ and is comprised of the following operation sequence:

```
SELECT_NONE;
SELECT_NODE_ATTRIBUTES ( attributes );
REDUCE_SELECTED;
```

Analysis: Since SELECT_NONE and SELECT_NODE_ATTRIBUTES are both $\Theta(|V|)$ and REDUCE_SELECTED is $\Theta(|V| \times |L|)$, REDUCE_ATTRIBUTES is also $\Theta(|V| \times |L|)$.

While the REDUCE_ATTRIBUTES operation serves many useful purposes, it is often desirable to consolidates nodes of similar attributes into groups of composite nodes rather than into just one single node. Of specific interest is the use of the *name* attribute.

In large system design, naming conventions are frequently used to help identify groups of closely related components. In Ada for example, compilation unit specifications and their bodies (i.e. implementations) possess the same unique name. Consequently, it is advisable that these elements be consolidated into the same composite node. If applied globally, the visual complexity of the entire system could be reduced substantially, consolidating as many as $|V| \div 2$ nodes. Similarly in Ada, subunits resulting from applications of top-down decomposition design share the same name prefix.

The elements COMPUTE, INPUT, and OUTPUT from the examples above can be used to illustrate this point. The complete name specifications of these elements would be MAIN.COMPUTE, MAIN.INPUT, and MAIN.OUTPUT, respectively. Hence, reducing these elements into the same composite node would again considerably reduce overall visual complexity. Application of this concept to Figure 3.2 yields the very simple system layout shown Figure 3.3.

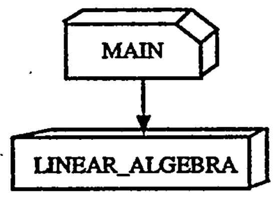


Figure 3.3 Reduced version of example program

Obviously, this type of sequence could be performed manually using operations that have already been introduced. This, however, assumes that the names of the system elements to be reduced are known ahead of time. The ability to automatically apply this concept on large system configurations with numerous naming groups, all of which may be unknown prior to the operation, could significantly reduce visual complexity. The REDUCE_NAMES operation is introduced for this purpose.

Operation: REDUCE_NAMES ;

Description: The REDUCE_NAMES operation again makes use of many of the previously defined operations. REDUCE_NAMES iterates through a layout's node list examining the *name* attributes of all node pairs. All nodes with suitably matching name criteria (as defined by the MATCH function) are selected. At the completion of each pass through the outer loop, the selected items are reduced into a single node if more than one item was selected. The *name* attribute of the new composite node is assigned based on the *name* attribute of the internal nodes. The process repeats until all node pairs have been examined.

```
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P_\Lambda$ do
```

```
        SELECT_NONE;
        $\forall w \mid w \in V, w \rightarrow p_w, p_w \in P_\Lambda$ do
            if MATCH( NAME($v$), NAME($w$) ) then
                SELECT_NODE($w$);
            end;
        end;
        if $|\sigma_\Lambda| > 1$ then
            REDUCE_SELECTED;
            SET_NODE_ATTRIBUTE( $v_{new}$, $name$, NAME($v$) );
        end;
    end;
```

Analysis: Since REDUCE_NAMES must examine all node pairs within the layout Λ, a total of $\Theta(|V|^2)$ examinations is required. In the worst case, a total of $|V| \div 2$ applications of the REDUCE_SELECTED operation may have to be performed. As a result, a total of $|V|^2 \times |L|/2$ operations will be required. Since $|V|^2 >> |L|/2$, REDUCE_NAMES is $\Theta(|V|^2)$.

Edge Attribute Reduction

Just as it is useful to perform node consolidation within layouts based on node attributes, so to may it be useful to employ edge attribute reduction techniques. The utility of a general purpose reduction operation which consolidates an entire set of nodes associated with a particular edge attribute into a single composite node is probably of limited value since edge attribute similarity in large configurations seldom implies a direct relationship. However, the ability to reduce a node directly linked to several dependents via an edge with a particular attribute has many more applications. Two cases concerning the *implied* and *restricted* attributes are immediately apparent.

Recall from Section 2.8 that the *implied* attribute was used to indicate tight coupling between system elements where the existence of one element immediately implied the existence of the other. In these instances, it is convenient to consolidate both nodes into a single node to reduce overall system visualization complexity. Similarly, the restricted attribute was used to represent the dependency that exists between a system element and its subelements in hierarchical and top-down decompositional system architectures. Here too it would be beneficial to consolidate an element and all of its subelements into a single composite element for the same reasons. The REDUCE_IMPLIED and REDUCE_RESTRICTED operations are provided for this purpose.

Operation: REDUCE_IMPLIED ;

Description: The REDUCE_IMPLIED operation scans all node edges contained within Λ and consolidates all nodes associated with edges that possess the *implied* attribute. Consolidation is performed by selecting a node and all of its dependents that are linked via an implied edge executing the REDUCE_SELECTED operation. The process repeats until all node/edges in Λ have been examined.

```
$\forall v \mid v \in V, v \rightarrow p_v, p_v \in P_\Lambda$ do
    SELECT_NONE;
    SELECT_NODE($v$);
    $\forall w \mid w \in V, w \rightarrow p_w, p_w \in P_\Lambda, (v, w) \in E$ do
        if $implied \in \varepsilon_{(v, w)}$ then
            SELECT_NODE($w$);
        end;
    end;
    if $|\sigma_\Lambda| > 1$ then
        REDUCE_SELECTED;
    end;
end;
```

Analysis: REDUCE_IMPLIED must iterate through all $|V|$ nodes performing a REDUCE_SELECTED operation on as many as half of the nodes. Since REDUCE_SELECTED is $\Theta(|V| \times |L|)$ and $|V|^2 >> |L|$, REDUCE_IMPLIED is $\Theta(|V|^2)$.

Operation: REDUCE_RESTRICTED ;

Description: The REDUCE_RESTRICTED operation is identical to REDUCE_IMPLIED with the exception of the statement

If *implied* $\in \mathcal{E}_{(v,\,w)}$ then

which is changed to

If *restricted* $\in \mathcal{E}_{(v,\,w)}$ then

Analysis: As above, $\Theta(|V|^2)$.

3.6 Compaction

When working with large configurations, nodes can easily become widely dispersed across the visualization making the entire configuration difficult to view in its entirety. Consequently, it frequently is useful to be able to "compact" a configuration along one of the axes so that nodes are located in close proximity. The ARRANGE_UNIFORM operation presented in Section 3.4 accomplished this along the *x*-axis. This operation can easily be reformulated to the *y*-axis and *z*-axis, giving rise to the following three operations:

Operation: COMPACT_X
COMPACT_Y
COMPACT_Z

Description: Arranges nodes uniformly along the *x*, *y*, and *z* axes, respectively.

Analysis: Identical to ARRANGE_UNIFORM, $\Theta(|V|)$.

A related operation is to compact the visualization space itself, rather than its contents. As defined, a visualization space is set of possible node position values. Compaction of a visualization space is used to reduce of the number of positions possible for nodes.

Operation: COMPACT_VOLUME

Description: COMPACT_VOLUME reduces the current visualization space S_Λ to the smallest volume capable of holding the current layout's contents. Essentially, COMPACT_VOLUME iterates through the current layout to determine the maximum x, y, and z coordinate of every node and uses these values as new dimensional limits for S_Λ.

Analysis: $\Theta(|V|)$.

The COMPACT_VOLUME operation is particularly useful during the optimization process presented in Chapter 6. Used prior to a depth-first and breadth-first arrangement operation and applied to a layout's root node, provides a simple-heursitic for determining suitable visualization space limits. The maximum height and width of the visualizaiton space at the completion of these two operations is generally adequate. The following operation sequence provides this functionality:

```
ARRANGE_DEFAULT;
COMPACT_VOLUME;
SELECT_ROOT;
ARRANGE_DEPTH;
ARRANGE_BREADTH;
```

3.7 Archival

The final set of operations to be discussed are primarily concerned with configuration housekeeping type functions that provide a necessary foundation for an effective interactive environment. These operations allow configuration specifications to be originally loaded from an arbitrary system, manipulated, and then stored for later access. Included in this discussion are also the export operations for generating meta-configurations.

As an aid to both defining and saving configuration descriptions, a definition language referred to here as ADL (an acronym for A-Vu Definition Language) is referenced. This language corresponds closely to the configuration model presented in Chapter 2, but incorporates a syntactical structure that makes it easier to specify, read, maintain, and parse than the formal description. A complete description of ADL is presented in Appendix A.

The ability to maintain configurations in this manner has tremendous advantages. First, a complex system can often be organized and viewed in many different perspectives. As mentioned in Chapter 1, the type of user, their particular expertise, and their specific interest will often mandate that they view a particular system from a different perspective. The ability to create and save different configurations of the same system allows these users to explore many different aspects of the system, providing a mechanism for documenting their understanding and for exchanging their perspectives with others.

Second, just as a text document can not always be prepared in one sitting, it may also be difficult to assemble a desirable configuration for a complex system in a short period of time. Similar to text, a configuration may require further editing and modification as additional information about the system is obtained or clarified. A configuration storage mechanism is useful for maintaining and updating system views that may continue to evolve over time.

Lastly, the use of a readable/writeable definition language allows the tools developed here to process dependency structures from many different types of systems via the use of an appropriate translator. The configuration definition is sufficiently flexible that it can be used to easily represent virtually any directed graph type description.

The operations presented in this section were again modeled after a typical interactive text editor, allowing configurations to be created, saved, and closed in external file storage. A typical configuration examination session begins with an OPEN operation where an initial ADL description of a system is read and recast in an appropriate configuration data structure as determined by the implementation. Alternatively, the LOAD operation can be used to extract the dependency information directly from a system environment. The A-Vu system described in Chapter 7 implements this operation for large Ada program libraries. Once an initial configuration is generated, the user can proceed to manipulate the configuration using the techniques presented throughout this discussion. A copy of the configuration can be stored at any time via a SAVE operation. Similarly, a previously saved copy can be recalled. At the completion of a session, the resulting configuration can then be closed, relinquishing any data structures that may have been allocated by the implementation. A brief summary of the basic operations used for these purposes is presented here. As all of these operations make use of standard language parsing and generation techniques, their performance is proportional to the number of nodes, edges, layouts, and bindings defined within the configuration. That is, each operation is essentially $\Theta(\max(|V|, |E|, |B|, |L|)$.

<u>Operation</u>: OPEN (*file_specification*) ;

<u>Description</u>: The OPEN operation reads an ADL file, parses its contents, and constructs a corresponding configuration data structure for use by the visualization system. In an actual implementation, the OPEN operation would be initiated by a user to examine either a newly defined configuration or a configuration that was saved from a past session.

<u>Operation</u>: SAVE (*file_specification*) ;

<u>Description</u>: The SAVE operation is used to store the current contents of a configuration in a file for later access via the OPEN operation. The OPEN and SAVE operations provide a mechanism for maintaining numerous views of the same dependency structure.

<u>Operation</u>: LOAD (*file_specification*) ;

<u>Description</u>: The LOAD operation provides an alternate method of initially defining a configuration structure. The LOAD operation is intended to be used within the context of a particular system environment for directly extracting element and element dependency information from a system. The A-Vu system described in Chapter 7 for example, uses this operation to extract the appropriate module information from large Ada software systems by parsing its program library listing. This operation is provided for convenience in working in this environment and eliminates the need for an intermediate ADL translator.

<u>Operation</u>: CLOSE ;

<u>Description</u>: The CLOSE operation is complementary to the OPEN operation. When access to a particular dependency structure is no longer of interest, the CLOSE operation is used to dispose of its configuration structure and prepare the visualization system for the next system to be examined. A typical implementation would provide verification that the current configuration being examined (if any)

has been saved and would prompt (if necessary) for a save operation.

The remaining two operations are provided specifically for the purpose of generating meta-configurations as described in Section 2.12. Recall that meta-configurations are used to describe either a configuration's layout containment structure or a configuration's layout dependency structure. The two export operations described below performs these functions. These operations are particular useful when working with very large configurations where the composite layout structure becomes increasingly complex with the introduction of each new composite space.

Operation: EXPORT_CONTAINMENT (*file_specification*) ;

Description: The EXPORT_CONTAINMENT operation creates a meta-configuration description of the current configuration's layout containment structure. This meta-configuration is written to a file in ADL format so that it may subsequently be accessed via an OPEN operation.

Operation: EXPORT_DEPENDENCY (*file_specification*) ;

Description: The EXPORT_DEPENDENCY operation is similar to the EXPORT_CONTAINMENT export operation except that the meta-configuration it creates describes the current configuration's layout dependency structure rather than its containment structure. This meta-configuration is again written to a file in ADL format.

While this concludes a comprehensive survey of important configuration manipulation operations, it is by no means exhaustive. The operations described above address the most frequented issues in complex structure analysis and provide a solid foundation with which to build other future operations. As configurations are explored, sequences of repetitive operations will emerge and additional dependency analysis questions will arise. This information can then be used to formulate new configuration operations as part of a continuous refinement process.

Although the utility of these operations can be easily shown in practice, the ability to measure their effectiveness is still lacking at this point in the discussion. At the completion of each operation, an analysis of the operation's performance was discussed. This analysis described only the operation's computational performance and did not capture how well the application of the operation contributed to increased system understanding. Hence, a quantitative measure for evaluating a system's complexity is now needed. This issue will now be addressed in the next chapter.

4. Configuration Viewing

The success of the manipulation operations presented in Chapter 3 is closely linked to the effectiveness of the methods for displaying the information that these operations produce. Dependency information display must be treated as an integral part of the configuration generation process. This chapter examines various issues associated with the generation of visual representations of A-Vu model configurations.

Several methods for displaying configuations depending upon their internal properties are presented in Section 4.1. When viewing an arbitrary configuration, it is useful to be able to examine the configuration from a variety of perspectives. Section 4.2 contains a collection of viewing parameter operations that provide this functionality. Section 4.3 defines a set of operations for navigating throughout nested configuration structures. Prior to displaying a configuration, it is also useful to first eliminate or reduce any unnecessary information. Section 4.4 describes a *filtering* process which serves this purpose.

4.1 Configuration Display

Once a configuration has been constructed, a visual representation of its contents must next be generated. While the A-Vu model mandates no specific technique, different methods lend themselves to different types of visualization spaces associated with each configuration layout. After a configuration has been created and an initial arrangement has been made, the visual representation that is generated must be suitable for the particular display environment. As various manipulation operations are subsequently applied to the configuration and elements within the configuration are moved throughout various visualization spaces, the visual representation of the configuration's current state must be updated accordingly. In an interactive environment, this update process must be continually applied while maintaining adequate user performance. Several operations are identified in this section that support these functions.

The A-Vu model treats a complex graph as a series of nested three-dimensional spaces. Most display environments, however, are two-dimensional in nature. Consequently, a mapping must be established from conceptual visualization space onto the physical display space. Recall from Section 2.4 that a visualization space may be discrete, hybrid, or continuous. Several different mapping schemes applicable to these different types of spaces are useful for display purposes.

Planar

In traditional graph layout approaches, complex graphs are decomposed into a series of subgraphs. Each of these subgraphs are then managed separately in planar two-dimensional form. Discrete and hybrid visualization spaces are a natural match for this type of approach. Each discrete plane in the visualization space is used to hold one of these subgraphs. A layout can then be displayed by examining planar cross-sections of its visualization space. If $p = (p_x, p_y, p_z)$ is the position of node v in layout L, and z is the discrete plane being viewed along the z-axis, then v will be mapped to the display if $p_z = z$. This technique is a natural extension of standard graph layout methods. With discrete visualization spaces, planar views are perhaps the most convenient and intuitive. Planar views are useful in examining the layered or hierarchically arranged layouts.

Composite

With hybrid and discrete visualization spaces, nodes can become widely dispersed across multiple planes. It is convenient at times to be able to project the positions of all these nodes onto a single two-dimensional plane to examine the density and distribution of nodes in the visualization space. If $p = (p_x, p_y, p_z)$ is the position of node v in layout L, and the visualization space is being viewed along the z-axis, then will v will be mapped onto a position (p_x, p_y) in two-dimensional plane regardless of the value of p_z. This operation is particular useful with hybrid spaces where nodes in various subgraphs can be freely placed within each plane, yet a single composite graph can be rapidly constructed.

Spatial

Since visualization spaces in the A-Vu are inherently three-dimensional, the third viewing scheme is to display the entire contents of the space using three-dimensional viewing technique that employs view perspective transformation, polygon clipping, and hidden surface removal [Ha'83]. Continuous visualization spaces are best displayed using spatial viewing since they exhibit no inherent planar structure. This scheme could be further extended to include color rendering, shading, light source illumination, etc. as described in [Ro'85]. Tools to interactively examine, enlarge, and rotate the visualization space representation are currently available through numerous software and hardware packages.

While a three-dimensional display environment would appear most beneficial due to A-Vu underlying visualization space structure, only certain dependency structure are ideally suited to this environment. Cone trees are an excellent example [Ro'91]. Unfortunately, arbitrary software system dependency structures rarely conform to natural three-dimensional objects in this manner. Consequently, full spatial rendering could actually compound rather than reduce visual complexity due to the increased amount of information to be presented. Nevertheless, spatial viewing provides a powerful visualization alternative to standard planar projection.

Refresh

One a configuration is generated and a display scheme has been selected, the visual representation of each layout within the configuration can be presented on a graphical display for examination by the user. Whenever a new layout within the configuration is selected as the current layout Λ, whenever Λ is modified, or whenever new viewing parameters are selected, a new visual display of this information must be generated and presented to the user. The REFRESH operation performs this function.

Operation:	REFRESH ;
Description:	This operations generates the actual graphical representation of the current layout Λ and presents this information to the user on a display screen. The details of this process are dependent upon the particular software visualization packages and methods employed. Suggested representations for nodes and edges based on their attributes where presented in Chapter 2. REFRESH performs this operation by scanning through the configuration data structures, examining the nodes and edges contained in the current layout along with their attributes. Information is displayed as directed by the viewing parameters described in Section 4.2.
Analysis:	Since this operation must scan every node and edge in the current layout to generate a display, REFRESH is $\Theta(\max\{|V|, |E|\})$.

4.2 Viewing Parameters

In order to apply the viewing schemes presented in the previous section, the display system must be properly directed as to which portion of the visualization space should be examined, how, and from what perspective. The operations below are typical of any three-dimensional viewing system with added support for discrete planar viewing. To aid in their definition, an additional variable will be added the configuration status definition.

Recall from Section 2.13 that the current state of configuration was defined by $\psi = (\Lambda, \Sigma, X)$. This definition will be extended to include current viewing parameters. Configuration state is redefined as $\psi = (\Lambda, \Sigma, X, \Xi)$ where Λ, Σ, X, are defined as before and where Ξ defines the collection of viewing parameters. Let $\Xi = (\varphi, \delta, \rho, \pi, \sigma, \tau, o)$. The variable φ is referred to as the viewing scheme, δ is referred to as the viewing direction, ρ as the viewing reference, π as the viewing perspective, σ as the scaling parameters, τ as the translation parameters, θ as the viewing orientation, and μ as the view reflection as described respectively below.

Scheme

As described in Section 4.1, three different view schemes can be used to examine the visualization spaces contained within a configuration. Let $\vartheta \in$ {*planar*, *composite*, *spatial*} represent the current viewing scheme as outlined above. The following three operations are use to select the type of viewing scheme to be applied during a REFRESH operation:

Operation: SET_PLANAR ;

Description: Selects planar viewing as the current display scheme.

$\varphi :=$ *planar*;

Operation: SET_COMPOSITE ;

Description: Selects composite viewing as the current display scheme.

$\varphi :=$ *composite*;

Operation: SET_SPATIAL ;

Description: Selects spatial viewing as the current display scheme.

$\varphi := spatial;$

Analysis: Since no operations are performed on any nodes or edge SET_PLANAR, SET_COMPOSITE, SET_SPATIAL are constant time, $\Theta(c)$.

Direction

Since visualization spaces are three-dimensional, they may be viewed from many different angles. With discrete visualization spaces, the most natural planar views occur along the x, y, and z axes. Hence, it is reasonable to be able to view such spaces from three basic directions. With hybrid spaces, planes are typically oriented along, say, the z-axis (front). Consequently, planar viewing along the x-axis (side) any y-axis (top) is of limited value since nodes seldom fall into natural planes along these directions. Composite viewing, therefore, is more appropriate for the side and top directions. With continuous spaces, nodes many not naturally reside within any plane. As a result, a viewing direction independent of the node arrangement is required.

The viewing direction parameter δ is best described as a vector in S. Operations for selecting viewing direction can then be defined in terms of δ as follows:

Operation: VIEW_FRONT ;

Description: $\delta := (0, 0, -1);$

Operation: VIEW_SIDE ;

Description: $\delta := (-1, 0, 0);$

Operation: VIEW_TOP ;

Description: $\delta := (0, -1, 0)$;

Operation: SET_DIRECTION (x, y, z) ;

Description: $\delta := (x, y, z)$;

Analysis: Constant time, $\Theta(c)$.

Reference

In order to select a particular plane within the visualization space to be viewed, a reference point ρ within the space ($\rho \in S$) must be first selected. The reference point ρ can also be used as center point for three-dimensional perspective projections in association with spatial viewing schemes. The SET_REFERENCE operation performs this function.

Operation: SET_REFERENCE (x, y, z) ;

Description: Sets the reference point ρ to the position vector (x, y, z).

$\rho := (x, y, z)$;

Analysis: Constant time, $\Theta(c)$.

For discrete and hybrid spaces, it is useful during interactive sessions to be able to move the reference point forward or backward a single plane at a time or move to a specific plane. The direction of movement or axis of plane selection can be controlled by the viewing direction parameter δ as follows:

Operation: VIEW_NEXT ;

Description: VIEW_NEXT subtracts a unit vector from the viewing reference point ρ along the viewing direction δ. This operation is performed by the statement:

$\rho := \rho - (\delta/\|\delta\|)^2;$

Operation: VIEW_PREVIOUS ;

Description: VIEW_NEXT adds a unit vector to the viewing reference point ρ along the viewing direction δ. This operation is performed by:

$\rho := \rho + (\delta/\|\delta\|)^2;$

Operation: SET_PLANE (p) ;

Description: VIEW_PLANE replaces the planar component of the viewing reference point ρ along the viewing direction δ with p. This operation is performed by:

$\rho := \rho - \rho\ (\delta/\|\delta\|)^2 + p\ (\delta/\|\delta\|)^2;$

Analysis: VIEW_NEXT, VIEW_PREVIOUS, and SET_PLANE are all constant time operations, $\Theta(c)$.

Perspective

With planar and composite view schemes, a parallel projection of the visualization space contents onto a planar viewing display is assumed. Under spatial viewing, a center of projection, the perspective point $\pi \in S$, can be specified. Unlike the parallel project schemes, a perspective projection provides the viewer with an indication of depth or distance. Items in the visualization space appear smaller the further they are away from π. The transformations that defined this projection are again defined in [Ha'83]. Each point in the visualization space would transformed accordingly during a REFRESH operation.

Operation: SET_PERSPECTIVE (x, y, z) ;

Description: Sets the center of perspective π to the specified position.

$\pi := (x, y, z);$

Analysis: Constant time, $\Theta(c)$.

Translation

When examining large configurations, it is often difficult to display the entire contents of layout on a single display screen at one time. Consequently, it is useful to be able to "pan" or "scroll" across across the visualization space, examining selected portions as desired. It p is an arbitrary point within the visualization space, then a transformation operation applied to p can be defined. Applying the transformation to all points within the visualization space during the execution of the REFRESH operation will produce the desired translation. This transformation can be defined in matrix form using homogeneous coordinates (i.e. 4×4 matrix) as

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix}$$

where T_x, T_y, and T_z define the distance to be moved in the x, y, and z directions, respectively. The viewing parameter τ is used by the REFRESH operation to perform this transformation.

Operation: SET_TRANSLATION (T_x, T_y, T_z) ;

Description: Sets the transformation parameters, τ.

$\tau := (T_x, T_y, T_z);$

Analysis: Constant time, $\Theta(c)$.

Scaling

With large configurations, it is also useful to be able to enlarge or reduce a layout's visual representation in order to make more effective use of the limited display screen space or to better discern configuration detail. This function can again be defined by a matrix transformation that is applied to each position p within the visualization space during the execution of the REFRESH operation. This transformation is defined as

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where S_x, S_y, and S_z define the scaling factors along the x, y, and z axes, respectively. The viewing parameter σ is used by the REFRESH operation to perform this transformation.

Operation:	SET_SCALING (S_x, S_y, S_z) ;
Description:	Sets the scaling parameters, σ.
	$\sigma := (S_x, S_y, S_z)$;
Analysis:	Constant time, $\Theta(c)$.

Orientation

With any of the three viewing schemes, its is often convenient to be able to rotate the visualization space with respect to the viewing display. This operation allows the visualization space contents to be better oriented along a horizontal or vertical axis for improved visual comprehension. The viewing parameter $\theta = (\theta_x, \theta_y, \theta_z)$ is used to described the angles of rotation along each of the three axes passing through the viewing reference point π and parallel to the visualization space origin.

Rotation about π is accomplished by first translating the visualization space from π to the origin using the translation transformation as described above and then performing the three axis rotations. The transformations for θ_x, θ_y, and θ_z are defined, respectively, as follows:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & \sin\theta_x & 0 \\ 0 & -\sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta_y & 0 & -\sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta_z & \sin\theta_z & 0 & 0 \\ -\sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For discrete spaces, θ_x, θ_y, and θ_z are limited to $\pm i\ 90°$ where $i = 0, 1, 2, \ldots$. A similar restriction exists for hybrid spaces when viewing the visualization space from either the top and side or whenever δ is not perpendicular to each plane within the space. The following operation sets the orientation parameters:

<u>Operation</u>: SET_ORIENTATION ($\theta_x, \theta_y, \theta_z$) ;

<u>Description</u>: Sets orientation parameters, θ.

$\theta := (\theta_x, \theta_y, \theta_z);$

<u>Analysis</u>: Constant time, $\Theta(c)$.

Reflection

The final viewing parameter to be presented concerns symmetry. The linkage between symmetry, visual simplicity, and visual comprehension has long since been established [We'52]. Operations to help explore the symmetrical aspects of complex structures is therefore very beneficial. While the mathematics of symmetry are commonly equated to group theory, only one of the most basic forms is discussed here. Numerous others are presented in [Ar'88].

The most common forms of symmetry exhibited within systems involves reflection symmetry otherwise know as bilateral or mirror symmetry. The viewing parameter $\mu = (\mu_{xy}, \mu_{yz}, \mu_{zx})$ describes mirror-type reflections across the x-y, y-z, and z-x planes, respectively. These reflections are defined as a set of transformations that are applied to each point p in the visualization space during the REFRESH operation. The transformations for μ_{xy}, μ_{yz}, and μ_{zx} are defined, respectively, as follows:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

These three transformations can be used as generators to create all eight bilateral symmetry transformations of the visualization space including the identify transformation. The following operation sets the reflection parameters:

Operation: SET_REFLECTION (μ_x, μ_y, μ_z) ;

Description: Sets reflection parameters, μ.

$\mu := (\mu_x, \mu_y, \mu_z);$

Analysis: Constant time, $\Theta(c)$.

4.3 Composites

With the creation of composite spaces, a method for navigating through a configuration's node/layout binding structure is necessary. Given a particular layout, if a node within the layout is selected that is bound to another layout, that layout will be selected as the current layout. A method for remembering the order in which layouts are visited is required in order to return along the same path. By restricting nodes to be bound to no more than one layout, a simple first-in, last-out type navigation scheme can be used as follows:

Operation: ZOOM_IN (v);

Description: The ZOOM_IN operation determines if the node v is bound to a layout l, and if so, pushes the current layout Λ onto a stack Γ and selects l as the new current layout. The following sequence performs this operation:

```
if $\exists l \in L \mid (v, l) \in B$ then
    $\Gamma \leftarrow \Lambda$;
    $L := l$;
end;
```

Since it is possible that a configuration may contain a cyclic binding structure, it is helpful to only allow the new layout l to be selected if it has already not been visited (i.e. exists on the stack Γ). The following refinement achieves this effect:

```
if $\exists l \in L \mid (v, l) \in B$ and $l \notin \Gamma$ then
```

<u>Analysis</u>: Since the binding list must be searched to determined if the node v is bound to any layout l, ZOOM_IN is $\Theta(|B|)$. With the restriction of only one layout allow per node, $|B| = |L|$.

<u>Operation</u>: ZOOM_OUT ;

<u>Description</u>: The ZOOM_OUT operation returns to the most recent layout visited, if any, reversing the effect of the most recent ZOOM_IN operation. The following sequence defines ZOOM_OUT:

```
if $\Gamma \neq \varnothing$ then
    $\Lambda \leftarrow \Gamma$;
end;
```

<u>Analysis</u>: Constant time, $\Theta(c)$.

4.4 Filtering

Using any of the metrics identified in Chapter 5, the visual complexity of a system configuration can be directly correlated to the amount of information that must be displayed to accurately depict the system. If the complexity of a configuration can not be sufficiently reduced by rearranging node positions, it makes sense that the alternative is to work towards reducing the volume of dependency information that must be portrayed. Obviously, this can not be done arbitrarily as important detail may be lost in the process.

The CUT operation presented in Section 3.2 coupled with the selection operations in Section 3.3 provides an obvious method for reducing nodes and their associated edges. Similarly, the use of composite configurations introduced in Section 2.11 provides effective means of consolidating related nodes and likewise consolidating their edges. Both of these techniques employ node reduction with edge reduction occurring also as a consequence. Once these techniques have been exhausted or the composite node structure of a system is not readily apparent, several useful techniques for directly reducing the number of edges within a layout can still be applied.

Within the A-Vu framework these edge reduction techniques are referred to as *filtering*. In actual interactive implementation, it is useful that these operations be performed by the visualization system at the completion of any modification to the current layout. The redrawing of the layout on a display screen is referred to below as a refresh operation. To sustain interactive service, it is important that refresh performance be maintained regardless of the filtering option selected. Several filtering methods are presented here.

Length

After executing a series of arrangement operations, it is possible that two loosely coupled, but dependent nodes may be positioned a considerable distance apart from each other. As an aid to examining only those nodes in close proximity, an edge length filtering option can be provided. This filtering operation can be easily performed by the visualization system during refresh operation by computing the distance between each dependent node pair and displaying only those edges whose lengths are less than the specified threshold.

Level

In systems that exhibit a high degree of layering (e.g. as measured by J_L), there is regularly little concern regarding the dependencies that occur within a particular level of the layering hierarchy. This relationship frequently occurs when peer entities cooperate on a set of tasks or share a mutual set of resources that are not accessible from outside the layer. In these instances, the edges between such nodes can be "turned off" or filtered from view. Since the layer of each node can be determined in constant time by examining the node's position, layer filtering can be performed with each display refresh in time proportional to the number of edges, $|E|$.

Transitive

Recall from Example 1.2 that a dependency of the form $A \rightarrow C$ is transitive if there exists a sequence of dependencies $A \rightarrow B_1$, $B_1 \rightarrow B_2$, ..., $B_{n-1} \rightarrow B_n$, and $B_n \rightarrow C$ for $n \geq 1$. The dependencies COMPUTE→SCALAR, COMPUTE→VECTOR, and MATRIX→VECTOR from the above examples illustrate this relationship. In system which exhibits a high degree of hierarchy (as defined by J_H), transitive edges result in low layering measures. Strictly hierarchical systems can be modified to obtain full layering compliance by replacing the transitive edges with a series of intermediate nodes positioned within each intervening layer. This practice is seldom instituted, however, as it introduces inefficiency with added volumetric complexity.

Transitive edges from a node v can be readily identified using a depth-first search. The length of each path or the number of interconnecting edges from v to another node w is recorded during the search. If there exists a path e from v to w of length one, and one or more other paths of length greater than one also from v to w, then e is transitive and would be turned off by the filtering algorithm. This process can be repeated for every node in the layout to identify all transitive edges.

Due to the $\Theta(|V|^2)$ nature of the transitive reduction process, the visualization system would suffer considerable performance loss if it were required to determine edge transitivity with each display refresh. Consequently, edge transitivity and edge filtering would typically be perfomed on demand. A new *transitive* edge attribute could be maintained by the edge transitivity algorithm and used to instruct the visualization system during display refresh operations to retain adequate interactive performance.

Reverse

Unless a configuration layout has been arranged according to strictly hierarchy, the layout may often contain reverse edges. A reverse edge is the result of dependency of a node position at a lower level in the layout upon a node positioned at a higher level in the layout. These edges are easily removed by the visualization system during refresh operations by simply examining the vertical position of each node and ignoring all such edges.

Attributed

It is possible that at times, only edges with certain attributes will be of concern. When examining the top-down decomposition of system, for example, only edges which possess the immediate attribute may be of interested. Conversely, it may be desirable at times to eliminate edges which possess certain edges. This can easily be accomplished by maintaining an edge attribute mask or set. During a refresh operation, the visualization system would simply compute the intersection of the attribute mask with the attribute set associated with each edge. If the results of the intersection yields the empty set, the edge would be filtered from view.

Having established methods for defining, manipulating, and viewing complex dependency structures, it is now important to be able to determine the effectiveness of the visual representations generated for these structures. An effective representation is one in which the desired organizational properties of the system are readily conveyed to an observer. To accomplish this, some type of quantitative measure must first be put in place. This issue will be addressed in the next chapter.

5. Configuration Evaluation

A quantitative mechanism for objectively evaluating dependency structure complexity is developed in this chapter. This evaluation process is established below by defining a set of suitable objective or "energy" functions which operate on configurations. Configuration evaluation is at the core of the optimization scheme that will be presented in the next chapter.

This discussion on configuration evaluation is divided into four parts. An overview of the energy function concept is given first in Section 5.1. Next, several popular energy function components that are based primarily on established complexity metrics are reviewed in Section 5.2. The discussion in Section 5.3 explores evaluation functions that are based solely on the directed graph information associated with a configuration (i.e. a configuration's node/edge arrangement). The remaining discussion in Section 5.4 examines objective functions that also have access to a configuration's semantic information (i.e. its node/edge attributes). Each of the last three sections contain a compilation of respective energy function types.

5.1 Configuration Energy Functions

Let C be a configuration. An energy function for C is then defined as $J(C)$ which yields a scalar complexity value. A high value of $J(C)$ is equated to a "complex" configuration while a low value of $J(C)$ is equated to a "simple" configuration. Consequently, the notion of configuration complexity or configuration simplicity can be defined in quantitative terms.

In practice, there are many factors that contribute to system complexity (or simplicity). To address this issue, the function $J(C)$ will be composed from a set of component functions J_1, J_2, J_3, ... J_n. Each of these individual functions will return a value for a particular aspect of the configuration. These functions form an energy function suite that enables the user to customize the layout process. Since some of these components may be considered more important to the user than others, a weight is associated with each *J(C)* component.

Note that some function components operate as *complexity* metrics and are desired to be minimized while the remaining components act as *simplicity* metrics and are desired to be maximized. Both component types are easily accommodated by assigning negative weights to the simplicity components so that they too may be minimized. The final objective function used by the A-Vu model is the weighted sum of each function component that is selected by the user. The resulting hybrid objective function is then defined by:

$$J(C) = \sum_i w_i J_i$$

The efficiency of the iterative improvement scheme presented in Chapter 6 is dependent on the computational complexity of components J_i. In order for the algorithms associated with these functions to perform well, configuration evaluation must be performed quickly. Consequently, the search for energy function in the next three sections will be again limited to functions whose algorithmic computational complexity is less than or equal to $\Theta(n^2)$ or $\Theta(k^2)$ where n is the number of nodes and k is the number of edges in the configuration. This restriction still offers considerable freedom in algorithm selection, but without undue performance degradation except for vary large values of n and k.

5.2 Metric-Based Evaluation

Effective methods for determining the complexity of a system's design and implementation are highly sought after. The ability to accurately predict a system's complexity early in its development life cycle would potentially reduce or eliminate the generation of undesirable components as well as reduce resulting implementation and long-term maintenance costs. To this end, a variety of metrics are now in common use with varying degrees of utility and success [He'89]. Several of these metrics are directly applicable to the A-Vu model and are presented here within the context of the configuration framework.

Counting - J_n, J_k

The first set of metrics involve simply a count of a particular system property. Two obvious metrics of immediate interest when dealing with an unfamiliar system are the total number of elements J_n and the total number of dependencies J_k defined, respectively, as follows:

$$J_n = |V|$$

$$J_k = |E|$$

Additional examples include the summation of sets of node and edge attributes. In software systems, a common metric in this class includes the popular "lines-of-code" metric. As simple as these metrics may appear, strong arguments can be made regarding their effectiveness [CM'91]. While counting metrics provide very limited information regarding a system's organization, they are effective in gauging the relative magnitude of the visualization problem. Consequently, they are typically the first metrics to be calculated.

<u>Analysis:</u> J_n and J_k are obviously $\Theta(|V|)$ and $\Theta(|E|)$, respectively, but generally can be determined in constant time $\Theta(c)$ depending upon the internal implementation of the configuration data structure.

Volumetric - J_V

Similar to the counting metrics are an analogous set of metrics that examine a system's *volume* or *displacement*. These metrics determine the amount of design space that is required to properly contain the system's description. In terms of software systems, this can be readily equated to the number of bits of memory or bytes of file storage that are necessary to contain the entire program. Within the context of the A-Vu model, the volume of a system J_V will be equated to the amount of visualization space that is necessary to display the entire system. J_V can be calculated as follows:

$$J_V = \sum_{i=1}^{|L|} \| S_i \|$$

where L is the set of layouts in C.

In simple terms, J_V is simply the sum of the length × width × height of all visualization spaces contained in a configuration. J_V differs from the counting metrics in that its value will change as nodes are rearranged in the visualization space. The number of nodes occupying a specified volume of visualization space and the number of edges occurring in this volume leads to the following respective definitions of node and edge density:

$$J_{V_n} = \frac{J_n}{J_V}$$

$$J_{V_k} = \frac{J_k}{J_V}$$

Analysis: The dimensions of each visualization space S_i in C are assumed to be finite and readily accessible in an actual implementation. This is readily accomplished by recording the limits of the visualization space each time a node is placed or repositioned. J_V is therefore assumed to be $\Theta(c)$. Since J_n and J_k are respectively, $\Theta(|V|)$ and $\Theta(|E|)$, both of the density measures are correspondingly the same.

While a default arrangement may yield a lower density value, a higher density layout will generally be required to obtain a better understanding of the system via, (e.g. a dependent or hierarchical arrangement). Care must be exercised when using these metrics as they provide a relative measure of complexity between different systems. Application of the metric on different configurations of the same system may be misleading since higher density configurations may often be much more difficult to understand.

Another important set of important volumetric measures appears in Halstead's software science [Ha'77]. These metrics are based on the following definitions:

n_1:	Number of unique system operands.
n_2:	Number of unique system operators.
N_1:	Total number of system operands.
N_2:	Total number of system operators.

Halstead's metric N is then a count of the total number of elements in the system, i.e. $N = N_1 + N_2$. The volume metric V is then defined as

$$V = N \log_2(n)$$

Halstead goes on to define an effort (E) and a difficulty (D) defined as

$$D = (n_2/2) \times (N_2/n_2)$$

$$E = V \times D$$

This method is readily incorporated in the A-Vu model by treating nodes as operands and edges as operators. Similarly, node attributes can be used to distinguish unique operands and edges attributes can be used to distinguish unique operators. While currently not incorporated into the A-Vu prototype system described in Chapter 7, these measures offer an alternative methodology for assessing system volumetric complexity.

Cyclomatic Complexity - J_M

Another immensely popular metric involves the use of MaCabe's cyclomatic complexity measure. Like Halstead's measures, this metric similarly asserts that system complexity is strongly related to certain of its measurable properties. In contrast to Halstead's method, MaCabe's measure is directly concerned with the graph structure of a system. This structure is identical in nature to an undirected equivalent of the graph $G = (V, E)$ within the A-Vu configuration definition. Within this context and assuming a connected graph, cyclomatic complexity is defined as

$$J_M = |E| - |V| + 2$$

While the utility of cyclomatic complexity has been debated [Sh'88], it offers a reasonable upper limit to the (e.g. $J_M \leq 10$) that should be maintained within systems. Because of its widespread use, cyclomatic complexity is incorporated into the A-Vu metric suite.

<u>Analysis</u>: J_M is obviously $\Theta(\max(|V|,|E|))$.

Connectivity - J_I

The last set of metrics to be discussed in this section are more closely involved with the dependencies between system components rather than strictly counts of certain system properties. A quick re-examination of Figure 1.3 is worthy of a reminder that these dependencies are the primary contributor to system complexity. Hence, consideration of this information is often of greater concern.

The information flow metric developed by Henry and Kafura [He'84] captures the severity or magnitude of system component interdependencies by calculating the fan-in and fan-out characteristics of each node as previously discussed in the degree selection portion of Section 3.3. Recall that fan-in is the total number of incoming dependencies on an element and fan-out is the number of outgoing dependencies on a system element. Defining fan-in as f_{in} and fanout as f_{out}, Henry and Kafura's information flow metric for a single node v can be stated as

$$I(v) = (f_{\text{in}}(v) \times f_{\text{out}}(v))^2$$

The total complexity of the configuration can then be computed by summing all of the complexities of each node as follows:

$$J_{IF} = \sum_{v \in V} I(v)$$

As with Henry and Kafura's information flow idea, a hybrid of this metric can be defined which captures the notion of internal node complexity. This extended concept maps naturally onto the composite layout structure already defined. While the outer layout of a complex configuration could appear simplistic, the configuration's internal layouts could be quite heavily interwoven. Internal or composite complexity can be incorporated by modifying the node information flow complexity definition as follows:

$$I(v) = I_i(v) \times (f_{\text{in}}(v) \times f_{\text{out}}(v))^2$$

The internal complexity of a node $I_i(v)$ is recursively defined as

$$I_i(v) = \sum_{(v,L) \in B,\, w \in L} I(w)$$

where w represents a node properly contained in L.

Analysis: Since the calculation of J_{IF} involves the examination of edge edge on edge node, J_{IF} is also $\Theta(\max(|V|,|E|))$.

Note that the above discussion computed the metrics J_n, J_k, J_M, etc. over the entire configuration. An alternative approach could easily be formulated that limits the scope of each metric to the node/edge structure of a particular layout (e.g. Λ). In practice, application of the metric in this manner has proven to be more versatile than a single computation over the entire configuration. Hence, it is assumed that each of the above metrics can be used either as $J_\phi(C)$ or $J_\phi(L)$, where ϕ is the particular metric of interest, C is an entire configuration and L is a single layout. Since a single layout could potentially contain all of the nodes in the configuration, the analysis of these layout-based metrics remain unchanged. This dual formulation will be assumed throughout the remainder of the chapter.

It is also important to note that each of the above metrics were based strictly on a system's graph representation. Although these metrics provide some invaluable information in assessing overall complexity, their use in generating understandable layouts is limited. None of the above metrics take into consideration the positions of nodes relative to each other. As nodes are repositioned in the visualization space, few changes, if any, will be recorded. With the exception of the density measure, the only way to affect a change is by either adding or deleting nodes to or from an existing layout or configuration. This static nature severely limits their use in visual layout evaluation. This will be rectified in the next section by introducing evaluation functions that the relationships between graph elements and the visualization space in which they reside.

5.3 Graph-Based Evaluation

The notion of quantifying visual complexity using graph-based objective function measurements was inspired by [He'89]. The objective functions presented in this section are useful as they collectively incorporate the layout criteria commonly used in conventional graph layout. Examples of traditional graph layout considerations are minimization of line crossings, hierarchical arrangement, total space (area or volume) occupied by the diagram, symmetry of the diagram, etc. Each of these considerations will be recast in objective function form.

The following is an initial compilation of graph-based objective functions:

Distance - J_D

Between every two vertices v_i and v_j, where $v_j \in Adj(v_i)$, we can define a line segment l_{ij} in the visualization space, denoting the graph edge from v_i to v_j as shown in Figure 5.1.

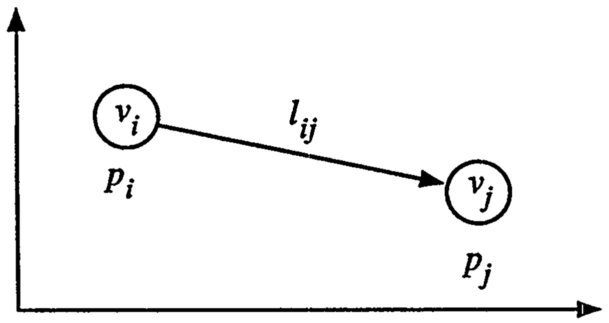


Figure 5.1 Distance between two points, p_i and p_j

The Euclidean distance D_{ij} between any two vertices v_i and v_j (assuming a 3-D space) is defined as

$$D_{ij} = \|l_{ij}\| = \|p_i - p_j\|$$

where p_i and p_j are the positions vectors of v_i and v_j, respectively.

If

$$E_{ij} = \begin{cases} 1, & \text{when } v_j \in Adj(v_i); \\ 0, & \text{otherwise.} \end{cases}$$

then the distance metric J_D can be defined as

$$J_D = \sum_{ij} E_{ij} \| p_i - p_j \| = \sum_{ij} E_{ij} D_{ij}$$

J_D is a measure of the total distance between all dependent node pairs in the graph or the total length of all edges of the graph. This function is effective in contracting a graph and captures, in some sense, the compactness of a program's representation in the visualization space.

<u>Analysis</u>: To determine the total distance of all edges, the function J_D must iterrate through the entire edge set E to obtain the positions of each dependent node pair. As a result, J_D is $\Theta(|E|)$.

Proximity - J_P

The use of J_D alone does not always assure that connected nodes will tend to lie within close proximity of each other. This can be observed in the following case involving three nodes v_i, v_j, and v_k as shown Figure 5.2.

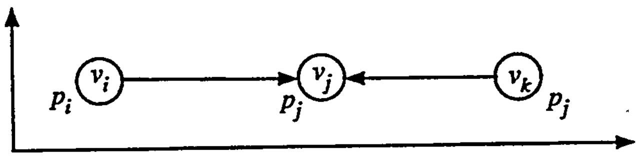


Figure 5.2 Design trade off between distance and proximity

Suppose a graph configuration has been sufficiently transformed to the point where it is unlikely that the positions of v_i and v_k will change due to their minimum objective function relationship with other nodes in the graph. Minimizing J_D would allow v_j to be positioned at any point between v_i and v_k since $\|v_i - v_j\| + \|v_j - v_k\|$ is a constant. This situation can be resolved by incorporating the notion of proximity into the objective function. This new term is used to cluster as many dependent nodes as possible. Let

$$P_{ij} = \begin{cases} -D_{ij}, & \text{if } D_{ij} \leq d; \\ 0, & \text{otherwise.} \end{cases}$$

where d is a constant designating the maximum allowable distance between two dependent nodes. The proximity measure between all dependent nodes in the graph is then

$$J_P = \sum_{ij} E_{ij} P_{ij}$$

As a simplicity metric, J_P is intended to be maximized and hence assigned a negative weight.

<u>Analysis</u>: J_P must again iterrate through the entire edge set and is therefore also $\Theta(|E|)$.

Edge Crossings - J_E

As was evident in Figure 1.3, edge crossings are a major contributor to a graph's visual complexity. Efficient algorithms now exist to determine if an arbitrary graph is planar [Ho'74] (i.e. can be drawn in a plane without any edges overlapping) and, if so, generate a planar embedding [Ch'79, Ja'88]. Unfortunately, large graphs, such as those underlying software systems, are seldom planar. Furthermore, seeking a planar organization of a graph is not necessarily useful due to the loss of spatial relationships between elements that results from the planarization process. Minimization of edge crossings, however, is still a very desirable feature, particularly when used in conjunction with other objective function components. Although the edge crossing problem is *NP*-hard [Ea'86], a near-optimal configuration can generally be found using either simulated annealing or genetic algorithms as proposed here or other heuristic methods [Wa'77]. The number of edge crossings J_E in a graph configuration can be computed as follows:

Let v_r, v_s, v_t, and v_u be four nodes in a graph where $v_s \in Adj(v_r)$ and $v_u \in Adj(v_t)$. We can assign to each node position vectors p_r, p_s, p_t, and p_u, respectively, as before. The edge between v_r and v_s can then be defined by the equation

$$l_{rs} = p_r \alpha + p_s (1 - \alpha) \text{ where } 0 \le \alpha \le 1$$

and the edge between v_t and v_u by the equation

$$l_{tu} = p_t \beta + p_u (1 - \beta) \text{ where } 0 \le \beta \le 1.$$

The two edges (v_r, v_s) and (v_t, v_u) cross if l_{rs} and l_{tu} intersect as shown in Figure 5.3. In order for l_{rs} and l_{tu} to intersect, there must exist a point p_c which lies on both l_{rs} and l_{tu}. The point p_c must satisfy the following equation:

$$p_r \alpha + p_s (1 - \alpha) = p_t \beta + p_u (1 - \beta)$$

where $0 \le \alpha \le 1$ and $0 \le \beta \le 1$.

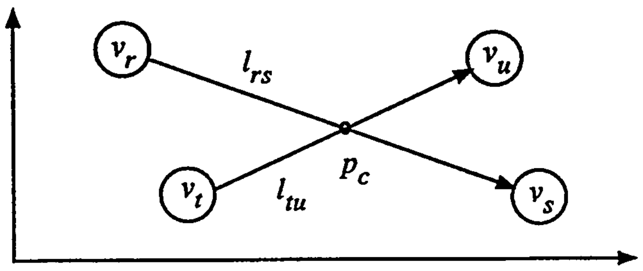


Figure 5.3 Crossing of two graph edges

To prevent the case where two edges such as (v_r, v_s) and (v_r, v_t) which share a common endpoint (node), from being counted as a crossing, we further restrict α and β to $0 < \alpha < 1$ and $0 < \beta < 1$ (i.e. $\alpha, \beta \ne 0$ and $\alpha, \beta \ne 1$). Rearranging for α and β yields

$$(p_r - p_s)\,\alpha + (p_u - p_t)\,\beta = p_u - p_s$$

Further discussion can be simplified if we assume a two-dimensional visualization space (i.e. $|S| = 2$). (Note: the notion of a three-dimensional edge crossing is not particularly meaningful in this application.) Since the position vectors contain both x and y components, the above equation can be rewritten in matrix form. Let

$$A = \begin{pmatrix} (p_r - p_s)_x & (p_u - p_t)_x \\ (p_r - p_s)_y & (p_u - p_t)_y \end{pmatrix},$$

$$B = \begin{pmatrix} (p_u - p_s)_x \\ (p_t - p_s)_y \end{pmatrix}, \text{ and } \mathrm{x} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

We then have the equivalent equation

$$A\,\mathrm{x} = B.$$

If $\det(A) = 0$ then the two line segments l_{rs} and l_{tu} can not possibly intersect. If $\det(A) \neq 0$, then the parameters α and β can be computed from

$$\mathrm{x} = A^{-1}\,B$$

Upon solving this equation, if $0 < \alpha < 1$ and $0 < \beta < 1$, then the two line segments l_{rs} and l_{tu} must intersect and likewise the two edges (v_r, v_s) and (v_t, v_u) must cross.

For every four vertices v_r, v_s, v_t, v_u, the parameters α and β can be computed using the above procedure.

Let

$$T_{rstu} = \begin{cases} 1, & \text{if } 0 < \alpha < 1 \text{ and } 0 < \beta < 1; \\ 0, & \text{otherwise.} \end{cases}$$

and

$$R_{rstu} = \begin{cases} T_{rstu}, & \text{if } \det(A) \neq 0; \\ 0, & \text{otherwise.} \end{cases}$$

then the total number of edge crossings in a configuration can then be computed as follows:

$$J_E = \sum_{rs}\sum_{tu} E_{rs} E_{tu} R_{rstu}$$

The terms E_{rs} and E_{tu} are non-zero when there exist edges (v_r, v_s) and (v_t, v_u), respectively. Similarly, the term R_{rstu} is non-zero whenever the edges (v_r, v_s) and (v_t, v_u) cross.

Note that as stated, this procedure will not consider two overlapping collinear line segments an edge crossings. The definition of R_{rstu} could, however, be suitably modified to check for this condition when $\det(A) = 0$. Alternatively, an additional objective function component could be defined which specifically checks for this condition.

Analysis: Since minimization of edge crossings is known to be *NP*-complete [EA'86], it is not suprising that the computation of edges crossings J_E is an expensive operation. As formulated, J_E requires examining of each *pair* of edges in the layout. This involves a double nested iteration through the edge set E, resulting in $\Theta(|E|^2)$. Several edge presorting heuristics have been developed, however, to help address the complexity of the underlying minimization problem [Ga'92].

Segmentation - J_B

It is often desired to divide a graph into two subgraphs such that the number of connections between the two subgraphs is minimal. This is analogous to placing the modules in two levels of a software hierarchy, partitioning modules into two separate subsystems, or assigning tasks to two processors trying to minimize interprocessor communication cost. In order to formulate the objective function for our minimization algorithms, let us first define a binary variable B_i associated with the nodes of the graph. When $B_i = 1$, the corresponding node belongs to one subgroup or level, while nodes with $B_i = -1$ belong to the other level. Now, an expression for the total number of connections between the two levels can be written as

$$N_c = \frac{1}{4}\sum_{i,j} E_{ij}(B_i - B_j)^2$$

The factor 1/4 can be explained by noting that the term on the right side makes a contribution of 4 to the total sum whenever there is an edge between i and j (i.e., $E_{ij} = 1$) and the two nodes belong to different planes (i.e., $B_i = - B_j$). The above can be rewritten as

$$N_c = \frac{1}{2}N_0 - \sum_{i,j} E_{ij}B_iB_j$$

where

$$N_0 = \sum_{ij} E_{ij}$$

is the total number of connections in the graph. Note if the graph is to be equally partitioned into two planes (assuming an even number of nodes), we can introduce a quantity

$$K_c = \sum_i B_i$$

which represents the difference in the number of nodes in the two groups. For equipartitioning of the nodes, this quantity must vanish for a valid solution. Thus, for an optimum bisection of the nodes into two groups, the quantity

$$J_B = N_c + \lambda K_c^{\,2}$$

for $\lambda \geq 0$, is minimized.

Analysis: As above J_B is $\Theta(|E|)$.

Hierarchy - J_H

The existence of hierarchical relationships within systems is very common. The ability to visualize these important dependencies therefore requires special consideration [Ca'80, Su'81]. With the exception of cyclic dependencies, a hierarchical relationship can be established whenever a dependency exists between two system elements. This relationship has been depicted in line graph form as an arrow from one node to the other. Implicit in this ordering is the notion that one element (or node) belongs at a higher "level" in a hierarchy. A visual representation of a hierarchical configuration is generated by locating higher level elements at higher elevations in the visualization space. To establish which portion of the visualization space corresponds to higher and lower elevations, we define a hierarchical basis vector h. The scalar value representing the level Y of a node's position p_i can then be computed by:

$$Y(p_i) = (p_i \cdot h) \text{ div } h_{layer}$$

where h_{layer} is a constant defining the height of the layer. Typically $h = (0, 1, 0)$, reflecting our desire for higher level elements to appear towards the top of the screen while lower level elements appear towards the bottom. If

$$H_{ij} = \begin{cases} 1, & Y(p_i) > Y(p_j); \\ 0, & \text{otherwise.} \end{cases}$$

then the total number of hiearchical relationships that a particular configuration contains is defined as:

$$J_H = \sum_{ij} E_{ij} H_{ij}$$

Analysis: As above, $\Theta(|E|)$.

Layering - J_L

Systems are frequently organized using layering concepts. A strictly layered software system is one in which all elements are dependent only on other elements at the same level or on other elements at the level directly below. A layering quantity L_{ij} can be computed for every pair of nodes v_i and v_j in a graph as follows:

$$L_{ij} = \begin{cases} 1, & \text{if } 0 \leq Y(p_i) - Y(p_j) \leq 1\,; \\ 0, & \text{otherwise.} \end{cases}$$

The number of layered relationships that exist in the graph configuration is then

$$J_L = \sum_{ij} E_{ij} L_{ij}$$

Analysis: As above, $\Theta(|E|)$.

Reflectivity Component - J_R

Aesthetic concerns are an important factor in graph layout. One of the most obvious characteristics of "nice" graph layout involves the use of symmetry. Symmetry appears in graphs in three rudimentary forms: reflection, translation, and rotation. The most common of these forms is reflection symmetry which exists whenever a graph possesses one or more "mirror" reflection planes. The reflectivity of a graph can be determined by counting the number of edges that reflect onto another graph edge along a specified reflection plane.

Let $P_R = (P, N)$ designate the reflection plane in S where P is an arbitrary point in the plane and N is a normal vector to that plane. Two edges (v_i, v_j) and (v_k, v_l) reflect onto each other if there exists a transformation Γ_R about P_R such that position vector $p_i \cong p_k \Gamma_R$ and $p_j \cong p_l \Gamma_R$. The transformation Γ_R can be generated by translating the point P to the origin, rotating about the x-axis until N lies in the xz plane, rotating the space about the y-axis until N lies along the x-axis, performing the mirror reflection, and then performing the inverse rotations and inverse translation. These steps are described below and illustrated in Figure 5.4.

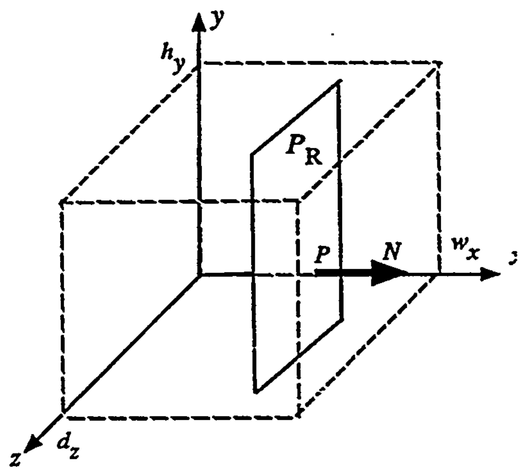


Figure 5.4 Mirror reflection about P_R.

Using homogeneous coordinates and standard matrix transformation notation [e.g. Ha'83], the initial translation to move the point P to the origin is

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -P_x & -P_y & -P_z & 1 \end{pmatrix}$$

The inverse translation which will move P back to it original location after the rotations and reflection have been completed is

$$T^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ P_x & P_y & P_z & 1 \end{pmatrix}$$

The next step in the process is a rotation θ along the x axis. Let $L_{yz} = (P_y^{\,2} + P_z^{\,2})^{½}$, then $\theta = \sin^{-1} P_y/L_{yz} = \cos^{-1} P_z/L_{yz}$. This resulting x-axis rotation transformation is defined as

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & P_z/L_{yz} & P_y/L_{yz} & 0 \\ 0 & -P_y/L_{yz} & P_z/L_{yz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The inverse rotation transformation is then

$$R_x^{\,-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & P_z/L_{yz} & -P_y/L_{yz} & 0 \\ 0 & P_y/L_{yz} & P_z/L_{yz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The normal N now lies in the xz plane. A rotation ϕ about the y axis must now be performed to align N with the x axis. Let $L = (P_x^{\,2} + P_y^{\,2} + P_z^{\,2})^{½}$, then $\phi = \sin^{-1} V_x/L = \cos^{-1} L_{yz}/L$. This rotation and its inverse are defined as:

$$R_y = \begin{pmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} L_{yz}/L & 0 & -P_x/L & 0 \\ 0 & 1 & 0 & 0 \\ P_x/L & 0 & L_{yz}/L & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y^{\,-1} = \begin{pmatrix} L_{yz}/L & 0 & P_x/L & 0 \\ 0 & 1 & 0 & 0 \\ -P_x/L & 0 & L_{yz}/L & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

With P_R now aligned with the *yz* plane, we are finally in a position to perform the mirror reflection. The reflection transformation is

$$M_{yz} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The final transformation Γ_R is then given by the product of the above transformations.

$$\Gamma_R = T R_x R_y M_{yz} R_y^{-1} R_x^{-1} T^{-1}$$

The reflectivity of a graph configuration can now be defined. Let

$$R_{ik} = \begin{cases} 1, & \text{if } \|p_i - p_k \Gamma_R\| < \delta \\ 0, & \text{otherwise.} \end{cases}$$

where δ is an arbitrarily small distance. The total reflectivity of a configuration is then

$$J_R = \frac{1}{2} \sum_{ij} \sum_{kl} E_{ij} E_{kl} R_{ik} R_{jl}$$

The above computation can often be simplified as we are frequently interested only in the reflectivity about a plane parallel to the *yz* plane. If the dimensions of S are $w_x \times h_y \times d_z$, a suitable choice for $P_R = (P, N)$ would be ($(w_x/2, 0, 0)$, $(1, 0, 0)$) as was used in Figure 5.4. Since N already lies along the x axis, $\theta = \phi = 0$. Γ_R can be reduced to $\Gamma_{yz} = T M_{yz} T^{-1}$ or

$$\Gamma_{yz} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ w_x & 0 & 0 & 1 \end{pmatrix}$$

The transformation Γ_{yz} thus maps a point (x, y, z) onto $(w_x - x, y, z)$.

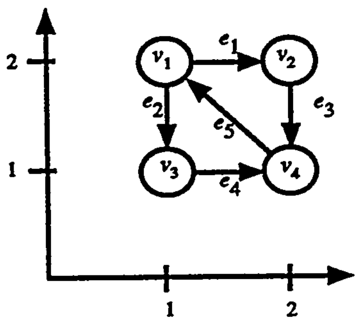
<u>Analysis:</u> As formulated, J_R would appear to be $\Theta(|E|^2)$ as it involves an examination of each edge pair similar to J_E. The complexity of J_R can be readily reduced, however, when using discrete and hybrid

visualization spaces. By maintaining an associative memory data structure indexed by visualization space coordinates, a simple algorithm requiring only a single iteration through the edge set can be used. The existance of a reflective edge can be determined by looking up the edge's two reflected endpoints in the associative memory. A reflected edge exists whenever two dependent nodes exist at the reflected coordinates in the visualization space. Using this method, the complexity of J_R can be reduced to $\Theta(|E|)$.

<u>Example 5.1</u> Suppose we wish to generate a visual representation for a system with elements $M = \{ m_1, m_2, m_3, m_4 \}$ and dependencies $D = \{ (m_1, m_2), (m_1, m_3), (m_2, m_4), (m_3, m_4), (m_4, m_1) \}$. Based on this information, we can construct a graph $G_1 = (V, E)$ with $V = \{v_1, v_2, v_3, v_4\}$, and $E = \{e_1, e_2, e_3, e_4, e_5\}$ where $v_i \leftrightarrow m_i$ and $e_1 \leftrightarrow (m_1, m_2)$, $e_2 \leftrightarrow (m_1, m_3)$, $e_3 \leftrightarrow (m_2, m_4)$, $e_4 \leftrightarrow (m_3, m_4)$, *and* $e_5 \leftrightarrow (m_4, m_1)$. Correspondingly, $e_1 = (v_1, v_2)$, $e_2 = (v_1, v_3)$, $e_3 = (v_2, v_4)$, $e_4 = (v_3, v_4)$, and $e_5 = (v_4, v_1)$. Let the visualization space for this system be $S_1 = I^2$, the set of two-dimensional integer vectors. That is, system elements can only occupy positions in a 2-D space whose coordinates are integers. We next a associate a position vector p_i with each node v_i, yielding the position vector set $P_1 = \{p_1, p_2, p_3, p_4\}$. The layout of this system can now be defined as $L_1 = (S_1, P_1)$. Assuming that the attribute set for this system is $A_1 = (\emptyset, \emptyset)$ (i.e. all nodes and edges possess only the *universal* attribute by default), the resulting configuration is $C_1 = (G_1, L_1, A_1)$. If we let $p_1 = (1, 2)$, $p_2 = (2, 2)$, $p_3 = (1, 2)$, and $p_4 = (2, 1)$, the initial visual representation for the system is shown in Figure 5.5.

Suppose we are interested only in the distance, hierarchy, and reflectivity aspects of this configuration. By examining Figure 5.5, we can see that $J_D = 4 + 2^{1/2}$. Letting $h =$

(0, 1, 0), we have $J_H = 2$. If the dimensions of S_1 are $w_x = 3 \times h_y \geq 2$, then the reflectivity J_R about $P_R = ((3/2, 0), (1, 0))$ is 1 (i.e. e_2 reflects onto e_3). By setting $w_D = 1$ and $w_H = w_R = -1$, the weights for J_D, J_H, and J_R, respectively, and all other weights equal to zero, the resulting value for $J(C_1)$ is $1 + 2^{1/2}$. □



v	$Adj(v)$
v_1	v_2, v_3
v_2	v_4
v_3	v_4
v_4	v_1

Figure 5.5 Configuration C_1

Example 5.2 Suppose the visualization space is instead selected as $S_2 = \Re^2$, the set of two-dimensional real vectors. That is, the elements can now be placed anywhere in the 2-D space. With this freedom, we can define a new configuration $C_2 = (G_2, L_2, A_2)$ where $G_2 = G_1$, $L_2 = (S_2, P_2)$, and $A_2 = A_1$. P_2 is once again a set of four vectors, as in Example 5.1, except the p_i vectors are now defined by

$$
\begin{aligned}
p_1 &= \left(\tfrac{3}{2}, 1 + \tfrac{\sqrt{3}}{2}\right) \\
p_2 &= (1, 1), \\
p_3 &= (2, 1), \text{ and} \\
p_4 &= \left(\tfrac{3}{2}, 1 + \tfrac{1}{2\sqrt{3}}\right).
\end{aligned}
$$

Configuration C_2 is shown in Figure 5.6. If we are now only interested in, for example, the proximity, layering, and reflectivity aspects of the configuration, we set $w_P = w_L = w_R = -1$ and all other weights equal to zero. By examining Figure 5.6 we see that $J_P = 5$, $J_L = 2$, and $J_R = 2$ (assuming P_R from Example 5.1 and the proximity distance constant $d = 1$). The resulting value for J is then -9. Note that if we use these same weights and re-evaluate the configuration C_1 in Figure 5.5 we again obtain a J value of -9 ($J_P = 4$, $J_L = 4$ and $J_R = 1$). Although configuration C_1 exhibits a higher degree of layering than configuration C_2, it is not as aesthetically appealing as configuration C_2 due to C_2's higher degree of reflectivity. A user may therefore wish to reconsider the weight set that was selected and perhaps weight J_R more heavily than J_L. Conversely, if revealing layer structure is of greater concern, J_R could be assigned a lesser weight than J_L, insuring a higher probability of finding configurations more closely resembling C_1. □

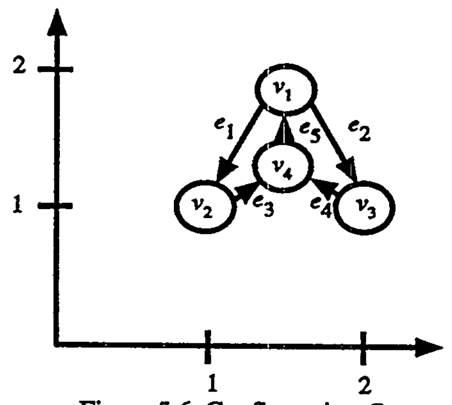


Figure 5.6 Configuration C_2.

5.4 Attributed-Based Evaluation

In the previous section we examined objective function components that were restricted to node/edge graph information. In this section we will relax this restriction and examine a set of functions that also consider the attributes associated with each node and edge. By using the semantic information associated with each system element and element dependency, this enhancement will enable us to pursue more meaningful layouts due to the additional layout criteria we can apply.

Listed below is an initial compilation of attributed-based objective functions. Since all of these functions can be implemented with a single scan of the edge set E, the time complexity of each is $\Theta(|E|)$.

Coupling - J_C

In typical system design, it is common for certain system elements to be more tightly coupled than others. In Section 2.5, the *restricted* attribute was introduced to help capture this tighter element dependency relationship. To aid understanding, a mechanism for indicating this strict dependence is desirable. Restricted edges, therefore, will be arranged vertically (i.e. parallel to the hierarchy vector h defined in the previous section) whenever possible.

Let

$$M_{ij} = \begin{cases} 1, & \text{if IMPLIED(} (v_i, v_j) \text{);} \\ 0, & \text{otherwise.} \end{cases}$$

and let

$$X_{ij} = \begin{cases} 1, & \text{if } h \times (p_i - p_j) = 0; \\ 0, & \text{otherwise.} \end{cases}$$

The term M_{ij} is nonzero for any edge (v_i, v_j) which possesses the *implied* attribute. The term X_{ij} is nonzero if the vector connecting the two distinct positions p_i and p_j for nodes v_i and v_j, respectively, is parallel to h.

$$J_C = \sum_{ij} E_{ij} M_{ij} X_{ij}$$

The function J_C is a measure of implied vertical edges. Since it requires maximization, it is assigned a negative weight. When combined with J_H, it can be used to insure that implied edges are hierarchically arranged. Similarly, when combined with J_L, it can be used to insure that implied edges are also layered as is shown in Figure 5.7.

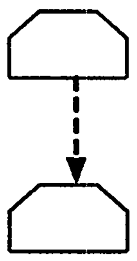


Figure 5.7 Implied edge layout

Information Hiding - J_i

Information hiding is one of the most powerful software engineering principles currently in use. Under this principle, system elements are only allowed access to objects which are required to implement that element's function. Similarly, the internal behavior and implementation of these objects are concealed. The existence of other objects that are not needed by the element is also hidden.

A popular mechanism for implementing information hiding is to divide system elements into two parts; their specification and their implementation. Recall from Section 2.4 that a node may possess the *specification* or *implementation* attributes. When generating a layout for these two elements, it is generally customary that the specification node by placed in close proximity (recall the definition P_{ij}) to the implementation node.

Let

$$I_{ij} = \begin{cases} 1, & \text{if SPECIFICATION}(v_i) \wedge \\ & \quad \text{IMPLEMENTATION}(v_j)\,; \\ 0, & \text{otherwise.} \end{cases}$$

Given two arbitrary nodes v_i and v_j, S_{ij} returns true (nonzero) whenever v_i is a specification and v_j is an implementation. The resulting formula to be maximized is as follows:

$$J_i = \sum_{ij} E_{ij} I_{ij} P_{ij}$$

Note that the use of J_I is an improvement over J_P alone as J_I attempts to minimize only the distances between nodes of a specific type, allowing other nodes to be more appropriately placed according to other optimization criteria. The component J_I can again be combined with both J_H and J_L to insure that implementation nodes are positioned directly beneath their specification nodes as in Figure 5.8.

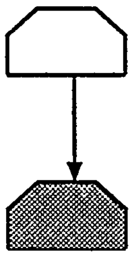


Figure 5.8 Specification - implementation layout

(c) *Reusability* - J_U

Another popular principle employed in modern system designs is the ability to reuse system elements. Programming languages such as Ada provide an explicit mechanism for parameterization of system elements. These generic elements must be instantiated with types, procedures, objects, etc. in order for the element to actually be used. Because of the dependency between an instantiation and its generic template, instantiated elements are customarily drawn above their generic templates. As a result of the reusability construct, frequently more than one instantiation will be depend upon the same generic template. Multiple instantiations of the same element should appear in close proximity of each other.

Let

$$U_{ij} = \begin{cases} 1, & \text{if INSTANTIATION}(v_i) \wedge \\ & \quad \text{GENERIC}(v_j)\ ; \\ 0, & \text{otherwise.} \end{cases}$$

The function U_{ij} returns true only when node v_i possesses the *instantiation* attribute and the node v_j possesses the *generic* attribute. The resulting formula to be maximized is:

$$J_U = \sum_{ij} E_{ij} U_{ij} P_{ij}$$

J_G can again be used with the components J_H and J_L to also insure a hierarchical and layered layout. An optimal reusable element configuration is shown in Figure 5.9.

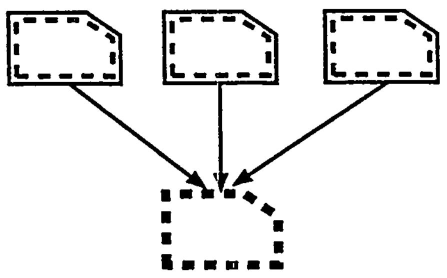


Figure 5.9 Instantiation - generic layout

(d) Top-Down Decomposition - J_T

Top-down decomposition or step-wise refinement is a common system design methodology that makes use of abstraction hierarchy. As was demonstrated with the J_H function in the previous section, we can use the hierarchical relationship implied by a dependency structure to construct a top-down decompositional arrangement. In section 2.4, the *restricted* attribute was proposed to capture this relationship. Since use of the *restricted* attribute implies a strict hierarchical dependence, a element's subelements should appear in close proximity of each other whenever possible.

Let

$$T_{ij} = \begin{cases} 1, & \text{if RESTRICTED(} (v_i, v_j) \text{);} \\ 0, & \text{otherwise.} \end{cases}$$

The function T_{IJ} returns true only when edge (v_i, v_j) possesses the *restricted* attribute. The resulting function to be maximized is

$$J_T = \sum_{ij} E_{ij} T_{ij} P_{ij}$$

As with the three previous cases, J_T can be again combined with J_H and J_L to enforce hierarchy and layering. Figure 5.10 contains a corresponding layout.

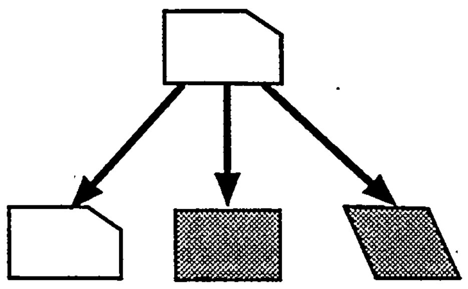


Figure 5.10 Decomposition layout

(e) Foreign Interfaces - J_F

In large system designs, there frequently exist interfaces to other external systems such as operating systems, resource servers, other cooperating processes, etc. These interfaces are commonly encapsulated within a system element to reduce foreign dependencies and enhance portability. In many software designs, these interfaces may be viewed as the lowest level of an abstraction hierarchy. Hence, it may be desirable to always constrain these elements to the "bottom" of the visualization space. The hierarchy vector *h* will once again be used to establish this direction.

Let

$$F_j = \begin{cases} 1, & \text{if FOREIGN}(v_i); \\ 0, & \text{otherwise.} \end{cases}$$

and let

$$B_{ij} = \begin{cases} 1, & \text{if } h \cdot pi > h \cdot pj\,; \\ 0, & \text{otherwise.} \end{cases}$$

The function F_j is used to determine if the node v_j represents a foreign interface. The function B_{ij} determines if the node v_i is located at a higher diagram position along the hierarchy vector h than node v_j. The function for optimal foreign interface placement is then:

$$J_F = \sum_{ij} E_{ij} F_i B_{ij}$$

Figure 5.11 contains an optimal foreign interface layout.

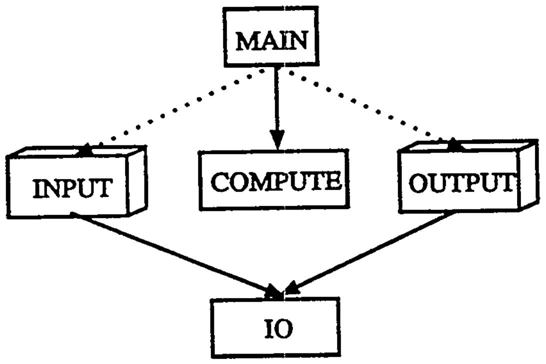


Figure 5.11 Foreign interface layout

(f) Similarity - J_S

Another organizational technique that can be employed is the grouping of similar elements. It may often be desirable to position nodes with similar attributes within close proximity of each other.

Let

$$S_{ij} = \begin{cases} 1, & \text{if } \varpi_i = \varpi_j; \\ 0, & \text{otherwise.} \end{cases}$$

That is, A_{ij} is non-zero whenever the attribute set ϖ_i for node v_i is equal to the attribute set ϖ_j for node v_j. Our similarity measure can then be defined as

$$J_S = \sum_{ij} E_{ij} S_{ij} A_{ij}$$

where A_{ij} is a distance or proximity measure such as D_{ij} or P_{ij}. Note that if we wish to constrain our similarity search to nodes, say, within a single layer, we can add the additional term L_{ij}. That is

$$J_{S_2} = \sum_{ij} E_{ij} S_{ij} A_{ij} L_{ij}$$

Alternatively, we can limit our search to hierarchical similarities by instead adding the term H_{ij} to our basic J_S definition as follows:

$$J_{S_3} = \sum_{ij} E_{ij} S_{ij} A_{ij} H_{ij}$$

A hierarchically arranged system with high similarity is shown in Figure 5.12 below.

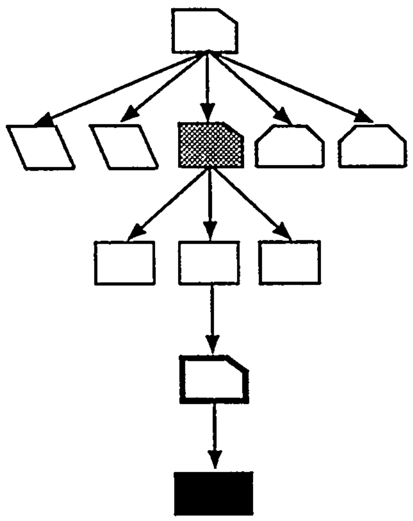


Figure 5.12 Hierarchical, similarity arrangement

The ability to compose additional functions in this manner leads to a general form of objective function definition. If α_{ij}, β_{ij}, χ_{ij}, ... are all functions of nodes v_i and v_j, then a composite objective function J_Λ can be generated as follows:

$$J_{S_3} = \sum_{ij} E_{ij} S_{ij} A_{ij} H_{ij}$$

With the exception of the edge crossing and reflectivity functions, all of the above objective functions were of this general form.

For example, suppose we desire a function J_x which will minimize the distance between specification nodes and implementation nodes and constrain them to the same layer of the visualization space. The following function achieves this purpose:

$$J_x = \sum_{ij} E_{ij} D_{ij} I_{ij} L_{ij}$$

With the above three families of energy functions now defined, a diverse collection of configuration evaluation tools are now available that provide a rich quantitative measure of a configuration's visual complexity. By combining these functions and adjusting their weights as described in Section 5.1, a single scalar complexity value can be generated for any arbitrary configuration. This aggregate function will now be used as the basis of an automated scheme for finding an "optimal" configuration with a desired set of characteristics.

6. Configuration Optimization

With establishment of the A-Vu configuration model, operations for constructing and manipulating configurations under this model, and techniques for evaluating the results of these manipulation sequences, attention can now shift to the integration of these methods. In this chapter, an automated process for generating simplified, aesthetically-pleasing configurations of complex dependency structures will be developed. An overview of the optimization process is presented in Section 6.1. The details of the optimization algorithms are presented in Section 6.2. The configuration generation techniques used during optimization are described in Section 6.3. An automated sequencing technique which joins all of these methods together is described in Section 6.4.

6.1 Process Overview

The configuration optimization process is equated to the optimal placement of nodes in a composite visualization space representation coupled with an appropriate configuration viewing strategy. This integrated process involves the following three distinct elements:

1) Optimization Paradigm

2) Node Placement

3) Sequencing

In order to generate a configuration of minimal complexity, a suitable optimization paradigm must first be adopted. Due to the complex nature of the underlying multi-variable minimization problem, it is common to select a strategy that has proven successful in natural systems. The strategy developed below uses an annealing "metaphor" as a guide. Given a suitable optimization paradigm, the second element of this process involves the creation of new configurations and the modification of existing configurations to achieve an optimal state. While the optimization paradigm provides direction during the simplification process, the placement techniques are required for preparation of the configuration sets to be evaluated.

The last element to be discussed involves the development of a single unified method of automatically applying or *sequencing* all of these tools and techniques. Given an arbitrary dependency structure, this mechanism is used to construct an initial configuration and perform a sequence of desired operations on this configuration without intervention. The original configuration is systematically transformed to achieve the desired result. Each of these elements comprising the configuration automation process are discussed in their respective sections below.

6.2 Optimization Paradigm

Configuration optimization is tantamount to the formulation of a suitable set of algorithms, heuristics, and complexity measures for processing and evaluating configuration structures. This involves the selection of a function J which, when applied to a configuration C of G, returns a scalar value characterizing the complexity of C. Under the A-Vu model, this involves a search for a configuration C that results in a minimum value for $J(C)$. The optimization task at hand can then be stated as follows:

Given a system with a dependency graph $G = (V, E)$ and attributes A, find a configuration $C = (G, L, A, B)$ as defined above such that $J(C)$ is minimized.

Within the context of a particular graph G and a single layout L and its associated visualization space S, this problem reduces to a search for an optimal set of node position vectors P. The solution space to this problem, however, involves many parameters.

If the number of nodes is $n = |V|$, and S is an N-dimensional space, then nN scalar values which minimize J must be found. Given a graph G with n vertices and a visualization space S with m unique locations for positioning the vertices, there are

$$\binom{m}{n} n! = \frac{m!}{(m-n)!}$$

different possible combinations for vertex positioning. Generally, m will be greater than n if S is a discrete space and infinite if S is a continuous or hybrid space. Regardless, there is an enormous number of choices for P. It is obviously not practical to exhaustively search the entire space in order to determine which configuration of dependency structure appears most useful. A better search strategy must be employed. Fortunately, several satisfactory strategies which yield near-optimal results have been developed in recent years. These strategies can be grouped into two broad categories, heuristic methods and metaphoric methods, each discussed below. The A-Vu approach uses a hybrid of these methods as will become apparent throughout the discussion.

Heuristic Methods

Perhaps the most widely used strategy for generating optimal graph layouts involves the use of specialized heuristic algorithms for organizing and positioning nodes within a two-dimensional plane. As outlined early in Chapter 2, this process typically consists of four basic steps which convert a cyclic graph to acyclic, decompose the acyclic graph into distinct layers, order nodes along each layer to minimize edge crossings and edge lengths, and then fine-tune the positions of each node to enforce aesthetic appeal favoring symmetry and balance. These particular methods have been deployed with considerable success offering numerous performance advantages (e.g. [Ga'93, Ea'90]). Many of the operations described in Chapter 3 conform to this class.

The primary drawback of the heuristic approach, however, is that these methods employ fixed layout criteria that can not be readily adjusted or tuned for a specific system. Consequently, the layout generated for a system may be inappropriate or only capture and convey one particular aspect of the system. In addition, these methods are often not intuitive, involving a series of complex algorithms for rank assignment and node ordering.

Metaphoric Methods

An alternative to discrete heuristic methods is to base the algorithms on an approximation or simplified simulation of natural, physical systems. The physical system mimicked serves as metaphor which guides the design and implementation of the method. Conceptually, this approach offers the advantage of being more intuitive assuming the process associated with the physical system is reasonably well understood. The most popular examples of these methods include spring-based models [Ka'89, Ea'84], force-directed placement [Fr'91], simulated annealing [Da'89, Ki'83], and genetic algorithms [Ko'91, Go'90, Gr'85]. Note that these methods in themselves can also be considered heuristic, but their link to the physical world is used here to set them apart.

The advantages and disadvantages of each of these techniques are mixed. The spring-based and force-directed techniques offer reasonable performance, but are again based on fixed layout criteria that may not be applicable to the problem at hand. While simulated annealing and genetic algorithm techniques offer greater adaptability to varying criteria, care must be exercised in their deployment. The performance of both techniques can be extremely slow.

Fortunately, the simulated annealing mechanism is very flexible and adaptable. Heuristic methods can be easily incorporated and the entire process can be readily tuned. Although traditionally considered inappropriate for interactive use, some simple modifications to the basic simulated annealing algorithm make it an ideal candidate for use in the A-Vu model. The ability to control the optimization process, adjust optimization criteria, and determine the amount of computational time to be invested offer several unique advantages.

While not to be dismissed, genetic algorithms are cumbersome as the primary optimization strategy. This technique converges to an optimal solution along several simultaneous paths. As a result, genetic algorithms formulated in their traditional sense for this application can not be easily configured, interrupted, and restarted without considerable expense. A variant of the basic genetic algorithm, however, will be adopted in Section 6.3 as a more intelligent means for generating configuration node placements. Continued discussion of their use will be postponed to that time.

The A-Vu model assumes a hybrid of the above techniques can be used to generate configurations. The integration of these techniques will be described in Section 6.4. The actual optimization strategy used by A-Vu depends heavily on the simulated-annealing model because of its adaptability and is best characterized as *iterative improvement.*

The simplest form of iterative improvement starts with an arbitrary initial placement of elements and selectively perturbs an element in its visualization space. If the perturbation results in an improvement of the desired evaluation function, the new configuration is accepted. The process is repeated until no further improvements are deemed possible. A modification to this scheme is *repeated iterative improvement* where the above procedure is essentially repeated several times with different, randomly selected, initial conditions in order to avoid the possibility of getting stuck in a local optimum.

The A-Vu design incorporates a variant of the simulated annealing algorithm, providing a repeated iterative improvement process. This design gradually transforms a complex configuration into a more simplified version. Unlike other optimization strategies, this transformation process can be carefully controlled through parameters associated with the annealing algorithm. A description of this algorithm is presented here.

Simulated Annealing

The method of simulated annealing is inspired by the statistical mechanics of gradual cooling (*annealing*) in condensed matter [Me'53] and has been applied successfully in a variety of applications including VLSI cell placement [Sh'91], chip floorplanning [Ru'89], and directed graph layout [Da'89]. This iterative heuristic technique seeks a near optimum (say "minimum") solution by randomly perturbing an initial configuration (viz., a set of parameters to the energy function *J*) and accepting all moves that result in a reduction in the value of *J*. New sets of generated parameters continue to be accepted as long as they result in decreasing *J*. To prevent this process from getting "trapped" in a local minimum, a parameter set which produces a higher value of *J* is occasionally accepted with a probability that decreases with an increase in *J*. In many implementations of this method, the acceptance probability is given by $e^{-\Delta J/T}$, where ΔJ is the increase in *J* and *T* is called the *temperature*, a term borrowed from statistical mechanics. Initially, the temperature of the system is set sufficiently high so that most configurations are accepted.

With each iteration, the temperature of the system is reduced according to a predetermined *cooling schedule*. As the system gradually begins to *cool*, fewer and fewer high-*J* (or high *energy*, in statistical mechanics parlance) configurations are accepted. At very low *T*, the probability of accepting a move to a much higher *J* configuration becomes very small. Eventually, the parameter system stabilizes and the process is terminated according to rules which define a sufficiently *frozen* (or optimized) condition. An outline of the algorithm is as follows:

```
$m$  : constant INTEGER   := *moves per iteration*;
$\alpha$  : constant FLOAT     := *cooling rate*;
$T$  : constant FLOAT     := *initial temperature*;

$C$      : CONFIGURATION    := *start configuration*;
$C_{Old}$ : CONFIGURATION;   -- old configuration
$J$      : FLOAT            := $\infty$;
$J_{Old}$ : FLOAT;           -- previous energy value
$\Delta E$     : FLOAT;           -- change in energy


loop
    for $i$ in $1..m$ loop
        $C_{Old}$ := $C$;
        $J_{Old}$ := $J$;
        $C$       := GENERATE($C_{Old}$);
        $J$       := $J(C)$;
        $\Delta E$      := $J - J_{Old}$;
        if $\Delta E \geq 0$ then
            if RANDOM# $\geq e^{-\Delta E/T}$ then
                -- *reject configuration*
                $C := C_{Old}$;
            end if;
        end if;
    end loop;
    $T := a * T$;
    exit when COOLED($C, T$);
end loop;
```

Analysis: Careful examination of the above algorithm reveals that the complexity of the inner loop is determined primarily by the node placement function GENERATE(C_{Old}) and the configuration evaluation function $J(C)$. The GENERATE function will be discussed in Section 6.3, but is limited to $\Theta(|V|)$. The complexity of $J(C)$ is determined by the particular evaluation function selected.

Assuming the functions presented in Chapter 5, $J(C)$ is typically $\Theta(|V|)$ or $\Theta(|E|)$, and worst case $\Theta(|E^2|)$. The remainder of the algorithm is controlled by algorithm parameters and the cooling conditions. A performance analysis covering different parameter values is presented in Chapter 8.

Although the above algorithm is very simple in concept, there is a great deal of flexibility in its implementation. From the discussion in Chapter 5, the evaluation function $J(C)$ can be as simple as a constant-time computation or as elaborate and complex as a series of equations. Similarly, the GENERATE function can be a simple $\Theta(|V|)$ random position generator or a complex heuristic. Lacking any special insight into an ideal configuration evaluator or an ideal configuration generator, a balance between $J(C)$ and GENERATE is probably advised. The details of several possible implementations of this function are discussed in the next section.

One minor improvement to the algorithm is to save the "best" (i.e. lowest energy) configuration reached during its execution. This refinement insures that the algorithm yields the lowest energy solution it encountered during its search, avoiding the situation where the algorithm explores several low energy configurations, but terminates at a higher energy local minimum. This situation can occur when, by chance, a higher energy configuration is accepted at a sufficiently low temperature that prevents it from returning to the lower energy region of the solution space.

6.3 Placement Techniques

Proper node placement is certainly the most crucial step in any graph layout problem. For very small systems, node placement can often be done by hand using intuition and a series of trial-and-error attempts. The editing functions from Section 3.2 would be useful for this purpose. As the size of the system increases beyond a dozen or more nodes, this technique rapidly becomes unfeasible. Several strategies for performing this placement operation are presented here for complex systems. The evaluation functions presented in Chapter 5 act as the guide for determining the desirability of each solution space sample.

Algorithmic Placement

A significant and obvious refinement to manual trial-and-error placement method is to apply specific layout algorithms to the configuration to obtain a desired result. The arrangement operations presented in Section 3.4 were provided for this purpose. These operations offer considerable performance advantages over all other schemes in some instances. Unfortunately, efficient algorithms are not always available. Depending upon the layout criteria selected, the only known algorithms for many layout problems are *NP*-hard. While suitable heuristics may exist for some cases, this approach is limited by the suite of algorithms available and the layout criteria they implement. In very large system problems, proper layout criteria is often not known and can frequently conflict. This issue has lead to the need for an alternative set of placement techniques that can be more readily incorporated into the above optimization strategy.

The simulated-annealing scheme is employed when the proper position of each node in a layout can no longer be determined by any know polynomial time algorithm or any reasonable heuristic. Should such a method exist or were to be discovered, it could then be incorporated into the suite of operations outlined in Chapter 3 and invoked accordingly. Lacking the knowledge and availability of such an algorithm, one must again relegate to probing the $m!/(m-n)!$ solution space in search of a desirable layout.

Random Placement

One of the simplest probing schemes that can be adopted involves the use of random placement. Given a configuration C which contains an existing layout L and its associated visualization space S and a position set P, each node contained in L is assigned a new value in P by randomly selecting a new position from S. A new configuration built in this manner would be returned by the GENERATE function for evaluation by $J(C)$. Should the result of J(C) indicates an improvement or is within an acceptable range as determined by the probability function $e^{-\Delta J/T}$, the new configuration will be accepted. If not, the configuration will be discarded. As outlined in the algorithm, the process repeats until the configuration is sufficiently "frozen" as indicated by COOLED(C, T).

This placement scheme bears resemblance to traditional Monte-Carlo techniques. Its primary strength is that it examines samples that are well distributed throughout the solution space and will not become trapped within a local minimum. Its primary weakness, however, is that it does not readily converge on *any* local minimum and depends strictly on random chance.

Random Displacement

Rather than continually evaluating the completely new, random configurations, the first refinement to be incorporated is to limit the movement of each node by some maximum displacement constant value δ. If the selected value for δ is large, the technique is identical to random placement. At increasingly smaller values of δ, however, examination of the solution space becomes successively localized, better enabling a local minimum to be found. Because of this tendency towards localization, the probability of examining areas of the solution space which exhibit the lowest possible energy values is somewhat smaller.

Random Controlled Displacement

To compensate for the limitations of random displacement, the next refinement is to allow the displacement value to change with temperature, (i.e. $\delta(T)$). At high temperatures, the value of δ would be set sufficiently large to enable nodes to move freely throughout the visualization space. As the temperature decreases, the value of δ is proportionally reduced, enabling the algorithm to better focus on a minimum and devote less time to evaluating configurations which have less certainty of reaching at least a local minimum.

Constrained Random Placement

Developing the controlled displacement idea further, the next enhancement is to impose additional constraints on node movements such as purposefully limiting them to a specific areas of the visualization space. This technique further reduces the number of potentially useless configurations that need to be evaluated. A common constraint of this type would be restrict nodes to a particular plane or linear segment of the visualization space as, for example, determined by one of the arrangement algorithms in Section 3.4. This refinement can be made by treating $\delta(T)$ as a displacement vector rather than as a scalar value.

Analysis: Since each of the above techniques compute each node's new position based on its existing position, a single iteration through the set V to compute P is required. Each of these techniques are therefore $\Theta(|V|)$.

While each of the above variations of random placement offer varying degrees of certainty in adequately sampling the solution space, they are each plagued with the unfortunate outcome that a potentially enormous range of useless samples must be evaluated in order to converge on a successful low energy configuration. While each successful configuration carries with it some of its past history, the strengths and weakness of each configuration generated are hard to discern. Ideally, one would like to retain those aspects of a configuration that lend the most towards achieving a low energy state. Similarly, one would like to discard those aspects that contribute high energy evaluations. The genetic algorithm mechanism identified above offers an attractive solution in this regard.

Genetic Placement

Genetic algorithms (GAs) are search algorithms based on the mechanics of natural selection and natural genetics. There usefulness in system layout has been established in [Gr'85, Ko'91]. Unlike traditional search methods, GAs work with a coding of a parameter set and not the parameters themselves. GAs require the parameter set of an optimization problem to be coded as finite length (typically binary) strings. The mechanics of a typical genetic algorithm involve a series of steps: random number generation, string copying, and partial string exchanges. GAs also use an objective (or energy or cost) function to evaluate the *fitness* of a particular parameter set. Unlike simulated annealing, however, GAs search from a *population* of points and not a single point.

The basic form of a genetic algorithm consists of three main parts: 1) reproduction, 2) crossover, and 3) mutation. At the start, an initial population of strings (coded parameter sets) is generated (typically at random). Reproduction is the process of replicating each of the strings in a population. A simple reproduction scheme assigns a weight to each string according to its fitness, the objective function J. Strings to be replicated are then randomly selected from the population using a weighted probability. Strings which are associated with higher fitness are replicated with a higher probability.

After the reproduction phase, each replicated string is randomly paired with another string in the population. Each pair of strings then undergo crossover. The crossover process involves the selection of a random substring from one of the strings. This substring is then swapped with the substring in the corresponding position in the other string of the pair. For example, given two strings α_1 and α_2 each of length l, an integer position k where $1 \leq k \leq l\text{-}1$ is selected at random. Two new strings are created by swapping the substrings $\alpha_1[k+1 .. l]$ and $\alpha_2[k+1 .. l]$. The new set of strings then become members of the next generation.

The reproduction-crossover cycle is repeated for a number of generations. The string which yielded the highest fitness function value $J(C)$ at the end of the final generation is chosen as the solution. To prevent the reproduction-crossover process from accidentally eliminating a potentially useful solution, a mutation operation is introduced. An occasional alteration is made to the value at a random string position, effectively guarding against irrecoverable loss of important material. This process effectively guards against getting stuck in a local minima by exploring other areas away from the current region of the solution space.

Within the A-Vu framework, the use of genetic algorithms appears ideally suited to the generation of the node position set P. Recall the basic optimization algorithm presented in Section 6.2. Under this scenario, the position of all nodes contained in a layout $L = (P, Q, S)$ are encoded as a binary string, α_L. Using genetic placement, the GENERATE function would first construct a binary string encoding for the current layout in C using the function $\Psi(C)$. $\Psi(C)$ returns a binary string encoding α_Λ of the position set P in the current layout Λ.

Let Π be the set of binary strings representing the population and let $\pi = |\Pi|$. An initial population is formed by randomly mutating α_Λ π-1 times yielding $\Pi = \{\alpha_\Lambda, \alpha_1,$

$\alpha_2, \ldots \alpha_{\pi-1}\}$. This initial population then undergoes the reproduction and crossover cycle c times as described above. At the completion of this process, the contents of the population P are examined. Using the inverse encoding $\Psi^{-1}(\alpha_i)$, the element in Π which yields the lowest value for $J(\Psi^{-1}(\alpha_i))$ is returned. The GENERATE(C) function for genetic placement is summarized as follows:

```
$\alpha_\Lambda := \Psi(C);$
$\Pi := \{\alpha_\Lambda\};$
for $i$ in $1..\pi\text{-}1$ loop
    $\Pi := \Pi \cup \{$ MUTATE($\alpha_\Lambda$) $\};$
end loop;

for $i$ in $1..c$ loop
    REPRODUCTION($\Pi$);
    CROSSOVER($\Pi$);
end loop;

min := $\infty$;
for $i$ in $1..\pi$ loop
    FITNESS := $J(\Psi^{-1}(\alpha_i))$;
    if FITNESS < min then
        min := FITNESS;
        $C_{min} := \Psi^{-1}(\alpha_i)$
    end if;
end loop

return ( $C_{min}$ );
```

Analysis: The performance of the genetic placement version of the GENERATE(C) function is determined by the function J(C). From the discussion in Chapter 5, $\Theta(J(C)) \leq \Theta(\max\{|V|^2, |E|^2\})$. The constants c (generation count) and π (population size) can be used to adjust outer loop performance.

As formulated, this placement approach offers considerable flexibility. In combination with the simulated annealing algorithm, a broad range of custom optimizations are available. With a small annealing iteration count ($m \approx 1$) and at low temperatures ($T \approx$

0), the optimization sequence will exhibit primarily genetic algorithm type behavior. Conversely, with a small population set ($\pi \approx 1$) and a small generation count ($c \approx 1$), the optimization sequence will exhibit primarily simulated annealing type behavior.

While little has been said regarding the MUTATE function, its interesting to note that the same choices available to the simulated annealing algorithm concerning placement apply to mutation. The MUTATE function could simply be a random bit modifier working directly with the binary string encodings. This is traditionally what is employed in genetic algorithm implementations. Alternatively, MUTATE could work with the position set P applying random placement, random displacement, random constrained displacement, etc. In this manner, both the simulated annealing and genetic algorithm processes have been effectively unified into a single optimization strategy. This strategy can be readily tailored to take advantage of the strengths of either optimization approach. The performance of this method is discussed in Chapter 8.

6.4 Configuration Sequencing

At this stage, all of the fundamental components necessary for constructing, manipulating, evaluating, and optimizing configurations have been established. Each of the items discussed up to this point can be invoked interactively, providing a diverse range of powerful tools for manual exploration of complex dependency structures. The prototype implementation presented in Chapter 7 illustrates this point. What remains, however, is the final method for integrating these tools so that the entire process can be automated.

While the tools described above offer significant flexibility that is sufficient for many system understanding scenarios, the ability to capture a series of configuration manipulation and optimization operations and be able reapply this same series a later time or on another system offers considerable time savings. A series of operations of this type is referred to as a configuration *sequence*.

A configuration sequence is simply a list of instructions that are to be applied to configuration. The set of operations described in Chapter 3, the application of various viewing options and parameters as discussed in Chapter 4, the selection of evaluation functions and their corresponding weights as described in Chapter 5, and the initiation and adjustment of the optimization process just presented in Sections 6.2 and Section 6.3 should all be capable of inclusion in a sequence.

To provide this functionality, a simple language known as the A-Vu Sequence Language (ASL) is defined. This language is essentially an itemized list of all possible operations and associated parameters that can be initiated by a full A-Vu implementation. The complete list of instructions is contained Appendix B. This list carefully mimics the majority of the operations presented throughout this dissertation.

Configuration sequences offer several additional advantages beyond simply automating a sequence of steps. For any given system, numerous sequences which reveal different aspects of the system can be applied. The ability to maintain multiple sequences is comparable to maintaining multiple views, but with less concern over modifications. If a system is modified, any of its sequences can be reapplied and, if necessary, updated accordingly without having to re-explore the system's entire dependency structure. Sequences can be captured and exchanged amongst users, providing an aid to both documentation and visualization. In addition, sequences also provide a general purpose machine interface to the graph layout tools of the A-Vu environment. Finally, execution of a sequence produces a natural animation of the system's graph layout process.

Example 6.1 Recall the initial example program represented in Figure 1.1. Using the ASL language, the layout shown in Figure 6.1 is generated via the following simple sequence:

```
load example
arrange default
find IO
cut
arrange dependent
arrange centered
weights symmetrical.wht
schedule default.sch
optimize
option length 1
option filter
```

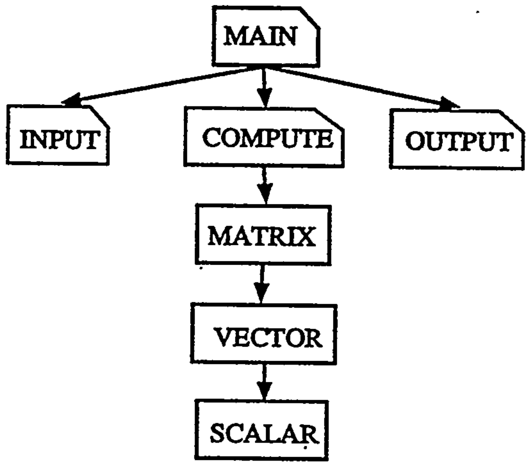


Figure 6.1 Results of example sequence

□

A dramatic demonstration of the utility of the configuration sequencing tools presented in this chapter is to apply this approach to the system represented early on in Figure 1.3. This demonstration will be postponed briefly until Chapter 8 to allow the A-Vu system to first be presented. The full sequence that will be used to transform Figure 1.3, however, is contained in Appendix C. The discussion now turns to the integration and interactive deployment of all A-Vu framework elements.

7. The A-Vu System

In order to validate the effectiveness of the A-Vu model and its associated visualization methods, a prototype software tool has been developed. Known simply as A-Vu, this computer program allows a user to interactively manipulate, analyze, and explore complex dependency structures using the techniques and operations discussed in the previous chapters. The A-Vu tool is presented in this chapter. A comparison of this tool with the model as presented in Chapter 2 is given in Section 7.1. A description of how this tool is organized and its information flow is presented in Section 7.2. Finally, a description of the operational user interface is presented in Section 7.3.

7.1 Model Compliance

This A-Vu system closely conforms to the model presented in Chapter 2. Dependency structures are represented using directed graphs and stored internally in both node adjacency and edge list data structures in support of constant and linear time set operations. As defined by the model, A-Vu allows subgraphs of the system's dependency structure to be stored in multiple layouts. A-Vu imposes no inherent limit on the number of nodes, edges, or layouts that can be represented. The edge coordinate set Q within each layout is generated automatically based on the locations of each node and is currently restricted to single line segments. Spline linkage is a natural implementation extension.

At present, only discrete visualization spaces are supported. A hybrid visualization space can be simulated, however, with a discrete space by increasing its dimensions and applying an appropriate scaling factor. Since the number of locations that can be occupied in a bounded discrete space is finite, the A-Vu maintains a cache for each visualization space. This cache enables constant time node look-ups for high-performance user interaction. The boundaries of this cache are maintained by the node placement operations and expanded automatically when a node is placed outside the space's current limits. Automatic visualization space compaction is purposefully not implemented since it would unnecessarily constrain the node placement algorithm used during optimization. Visualization space compaction can be initiated manually by the user or via an ASL command during the execution of an automated sequence.

Entire configurations are maintained internally in a data structure closely resembling the configuration definition $C = (G, L, A, B)$. All of the node attributes presented in Section 2.6 and edge attributes presented in Section 2.8 are support by the current A-Vu implementation. The visual representations of these attributes are implemented as presented in these sections. Currently, none of the auxiliary node information presented in Section 2.7 and Section 2.9 has been implemented. The current configuration data structures have been designed to allow these items as future enhancements. The ability to access, examine, and edit a node's source code via on-screen selection has, however, been demonstrated.

Perhaps the most powerful feature of the A-Vu system, unlike all other graph manipulation systems, is its support of composite layouts. The A-Vu system implements the full composite node/layout binding structure presented in Section 2.11. A single node or a collection of nodes and their entire composite structure can be freely copied, cut, and pasted in any of the visualization spaces within the configuration. Multiple copies of a node may even be placed in a single visualization space if desired. The attributes of each node, all parent nodes, and all intervening nodes and edges are automatically updated as defined by each editing operation. The meta-configurations outlined in Section 2.12 are supported as a result of the export operations and the ADL language interface described in Appendix B. Configuration state information is also maintained in a manner similar to that presented in Section 2.13 and updated in Chapter 4.

7.2 A-Vu Organization

A-Vu is an interactive tool that allows a user to manipulate complex dependency structures and immediately view the results of their actions. The ability to obtain quick results allows the user to explore the structure from many different perspectives in order to gain a satisfactory understanding of its intricacies. A block diagram of the tool's operation is shown in Figure 7.1.

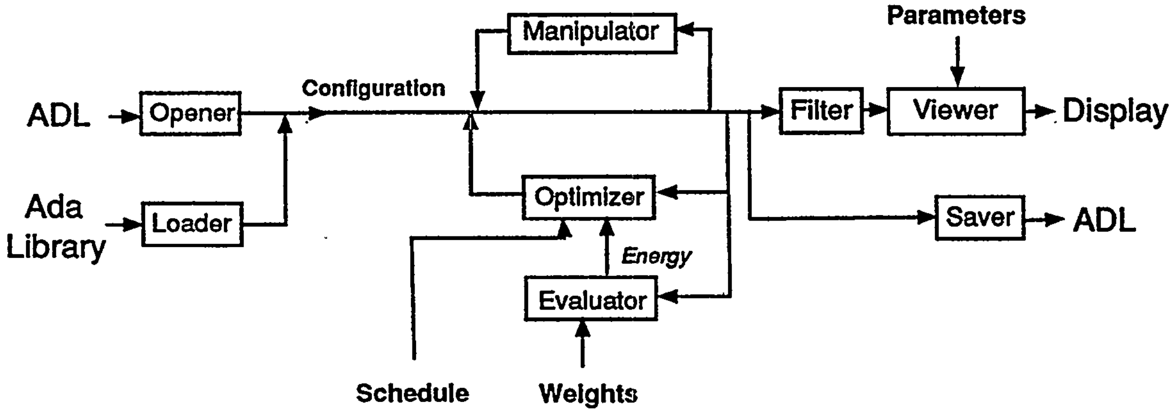


Figure 7.1 A-Vu block diagram

A-Vu currently accepts two forms of input. System dependency information can be input via A-Vu's Definition Language (ADL) as described in Appendix A. The ADL files can be created manually using a typical text editor, generated automatically by another tool, or retrieved from a previously saved A-Vu session. Alternatively, A-Vu can extract dependency information directly from a source code program library. The OPENER and LOADER modules shown in Figure 7.1 perform these respective functions.

The ADL language interface is provided as a general-purpose mechanism for examining arbitrary dependency structures. Information specified in an arbitrary syntax can be processed by A-Vu via an intermediate ADL parser/translator. The ADL language definition closely mimics the configuration definition presented in Chapter 2. When working with an program libraries, A-Vu parses a program library file description and extracts all the necessary module and module dependency information. The current implementation of the tool specifically supports Ada program libraries. Dependency information input in any of these forms is stored internally using a data structure representation that also closely resembles the A-Vu model's configuration definition. This data structure is built using the operations presented in Section 3.1. Once this data structure is constructed, the resulting configuration is continuously processed, manipulated, transformed, and displayed by the remainder of the A-Vu system components.

When a new system is initially input or originally loaded, a default layout is automatically generated by filling in a single plane visualization space of predefined width starting from left to right and top to bottom using the ARRANGE_DEFAULT operation (Section 3.4). The height of the visualization space is expanded as necessary. This provides a common starting point for all subsequent configuration operations.

The MANIPULATOR component of the A-Vu system allows a user to perform any of the editing, selection, arrangement, compaction, and reduction operations presented in Section 3.2 through Section 3.6. Alternatively, an optimization schedule may be defined and the OPTIMIZER component run in order to seek a minimal energy solution as described in Chapter 6. The EVALUATOR component is used to direct the OPTIMIZER component via its energy functions suite as described in Chapter 5. A weight set is input to the EVALUATOR to define the desired minimization criteria. The SAVER component may be used to archive any new configuration that is generated throughout this process as discussed in Section 3.7.

With each new generation, the configuration data structure is passed through the FILTER component (Section 4.4) and displayed via the VIEWER component (Section 4.1) System elements are automatically connected with a line segment according to the originally dependency information specified. The viewing parameters defined in Section 4.2 are used to control the display process.

7.3 User Interface

The A-Vu system is implemented as an X-Windows based application using the Open Software Foundation's (OSF) Motif widget set. The user interface is modeled after a typical window-based text editor. Analogous to text, individual nodes within a layout are selected using a mouse pointing device. Graph nodes may then be collectively copied, cut, and pasted throughout the configuration as desired. Configurations can be "reformatted" using any of the manipulation or optimization algorithms. These algorithms can be applied to all or a selected subset of nodes in the configuration. Additional algorithms to find certain node arrangements analogous to finding particular text sequences can also be applied. Unlike typical editors, however, all dependency information including node/edge attributes and edge connections are automatically updated after each operation.

The A-Vu command set is organized into five pull-down type menus appearing in its main window. A-Vu's main window is shown in Figure 7.2 with a typical default configuration display. The FILE menu provides access to all of the archival operations as presented in Section 3.7. For most user sessions, either the OPEN or the LOAD menu items within the FILE menu are typically the first to be selected. The EDIT menu provides access to all remaining operations presented in Chapter 3 via additional cascading menus. This includes the depth-first and breadth-first search algorithms, the connected component reduction routines, the various selection operations, etc. The viewing parameters from Section 4.2 are adjusted using the VIEW menu. The OPTIMIZE menu is used to control the optimization process and provides access to the automated sequencer. Filtering options and miscellaneous parameters display parameters are controlled via the OPTIONS menu.

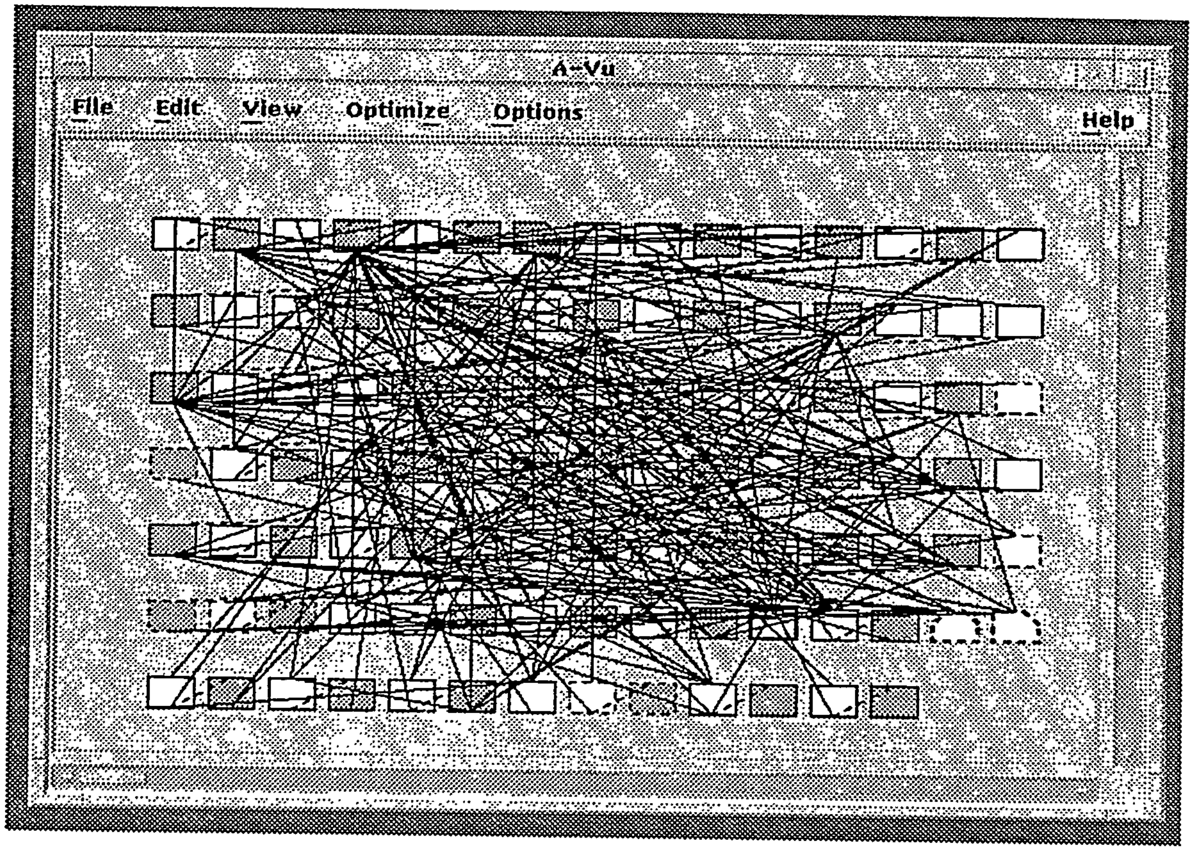


Figure 7.2 Typical default configuration

Upon completion of each user operation, the A-Vu system automatically displays the updated configuration via a REFRESH operation. Translation of the resulting visualization space can be performed as necessary using the standard horizontal and vertical window scroll bars. An overview window (Figure 7.3) is also automatically updated after each editing operation to reflect the current size, dimensionality, and view perspective of the visualization space. Under the restraints of a discrete visualization space implementation, A-Vu allows each spaces to be viewed along any of the three major axes with both axis rotation and axis reflection. The user can select either a planar view and step through the space a plane at a time, or a composite view and display a parallel projection of all nodes in the space simultaneously.

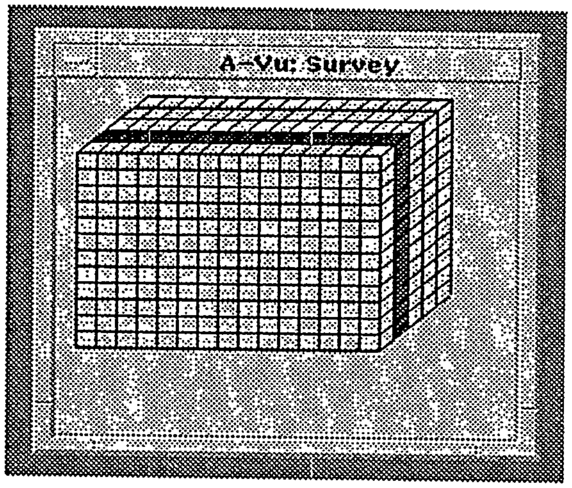
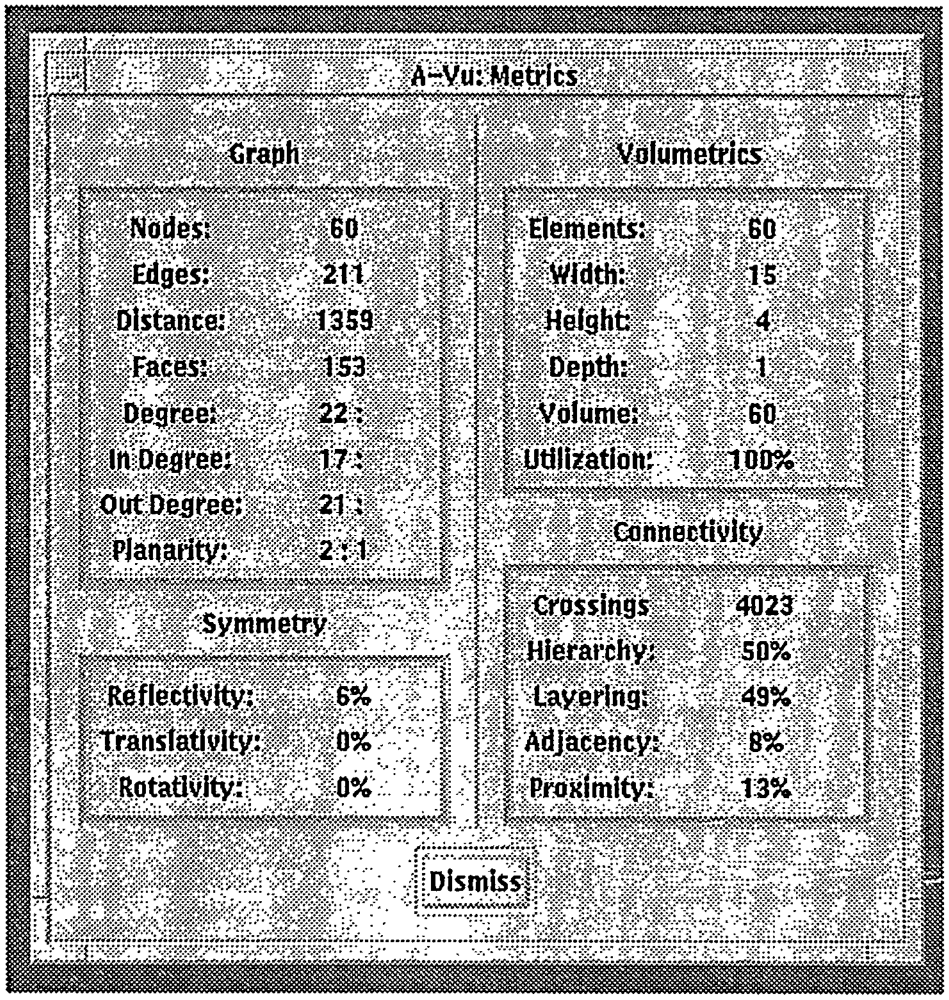


Figure 7.3 Visualization space overview window

To help analyze the complexity (or simplicity) of each configuration, the A-Vu system integrates a display of many of the evaluation functions presented in Chapter 5. Currently displayed metrics include volumetric complexity measures (visualization space size and dimensions), graph measures (nodes, edges, distance, minimum/maximum degree, etc.), symmetry metrics (reflectivity, translativity, rotativity), and connectivity metrics (number of crossings, hierarchical dependencies, layerings, etc.) The values of these metrics are displayed in a window (Figure 7.4) and are updated at the completion of each user operation. With the exception of edge crossings, all of these values may be computed in linear time or better. Due to the $\Theta(|E|^2)$ expense in determining the number of edge crossing J_E, A-Vu provides an option for disabling this computation. This option is linked directly to the edge filtering algorithm and is particularly useful in very large configurations where the utility of simultaneously displaying all edges is of limited value.



Graph

Nodes:	60
Edges:	211
Distance:	1359
Faces:	153
Degree:	22 :
In Degree:	17 :
Out Degree:	21 :
Planarity:	2 : 1

Symmetry

Reflectivity:	6%
Translativity:	0%
Rotativity:	0%

Volumetrics

Elements:	60
Width:	15
Height:	4
Depth:	1
Volume:	60
Utilization:	100%

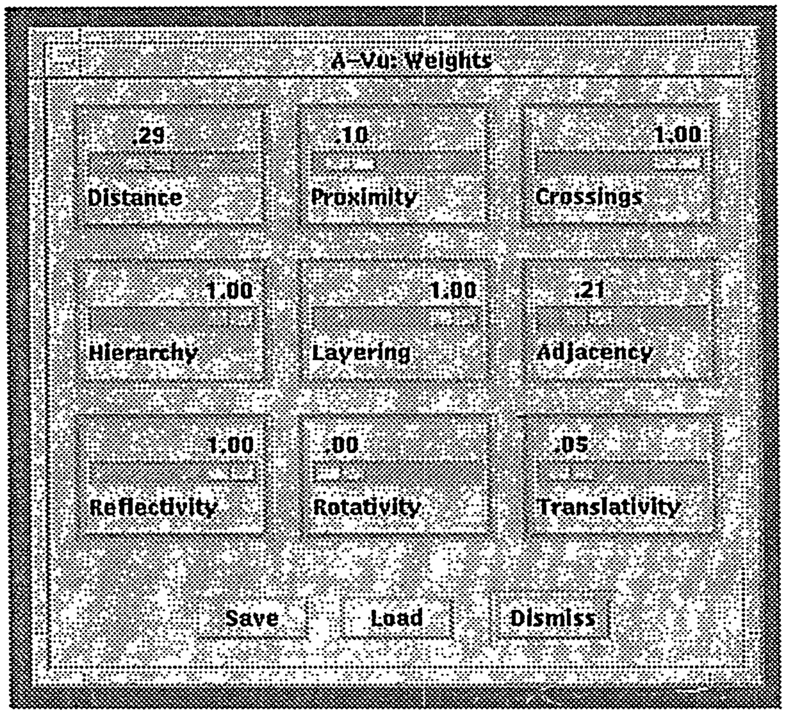
Connectivity

Crossings	4023
Hierarchy:	50%
Layering:	49%
Adjacency:	8%
Proximity:	13%

Figure 7.4 Metrics display

While the editing tools are useful for manual manipulation of configurations, their exclusive use for complex dependency structure analysis is inadequate due to the trial-and-error style of interaction that is required to seek a near optimal solution. The optimization techniques presented in Chapter 6 are integrated with A-Vu via two additional windows. The weights associated with each component of $J(C)$ are set with the control panel shown in Figure 7.5. The weights assigned to each function may vary between

0.00 and 1.00. As described in Chapter 5, all weights associated with complexity measures are automatically treated as positive values while those associated with simplicity measures are automatically treated as negative values. Additional weight controls may be attached to this panel as evaluation functions are added to the A-Vu energy function suite.



.29 Distance
.10 Proximity
1.00 Crossings
1.00 Hierarchy
1.00 Layering
.21 Adjacency
1.00 Reflectivity
.00 Rotativity
.05 Translativity

Figure 7.5 Weight control panel

Once the desired weights for an optimization sequence have been selected, a cooling schedule must be selected next. The control panel shown in Figure 7.6 is used for this purpose. The initial temperature, cooling rate, and freeze temperature can each be individually adjusted. Similarly, the number of iterations (i.e. configuration generations) per pass and the size of the sample set can be adjusted. The node placement scheme and node placement constraints are also selected via this control panel. At present, all but the genetic place modes have been implemented. Genetic placement is currently approximated with a variant of hill-climbing that incorporates a mutation operator. Optimization schedules can be saved for reuse at another time via the SAVE and LOAD commands.

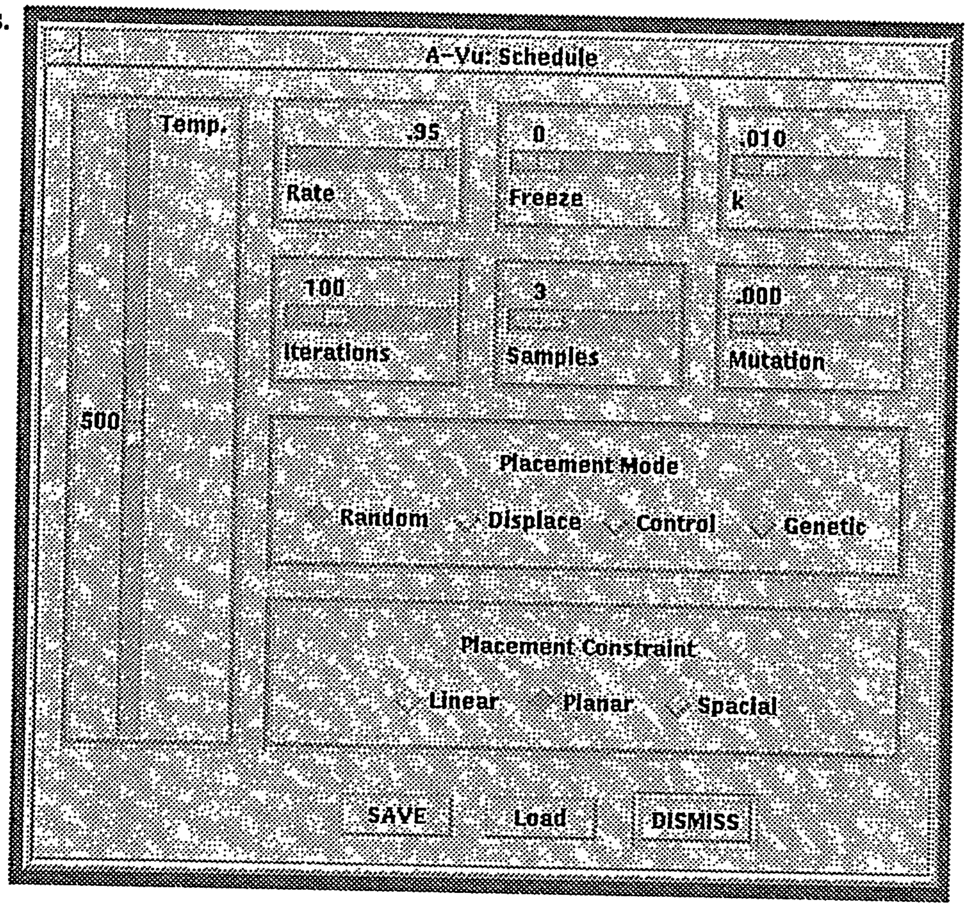
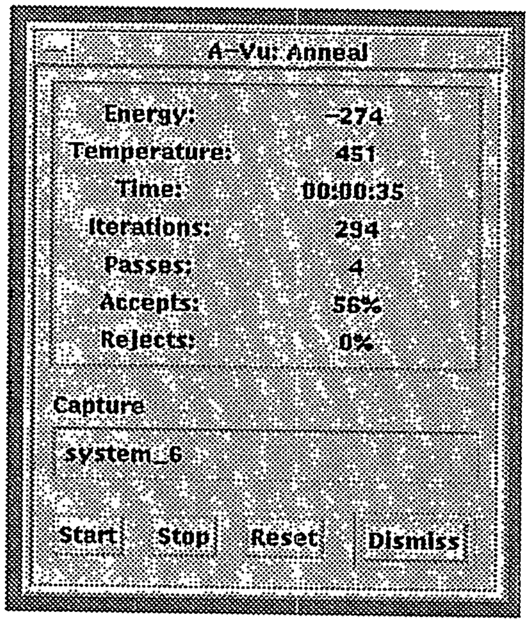


Figure 7.6 Cooling schedule controls

Once the desired cooling schedule has been assigned, the optimization sequence can be initiated via the optimization control panel shown in Figure 7.7. As a sequence progresses, the current temperature and energy is displayed on the panel. Similarly, the execution time, the number of iterations, and the number of passes are also displayed. The most recent configuration is displayed at the completion of each pass. The sequence can be stopped at any time allowing the user to manually modify the configuration or to adjust the cooling parameters and evaluation weights. The sequence can then be re-initiated. The energy values for each iterations may also be captured in a log file for plotting at a later time.


A–Vu: Anneal

Energy: –274
Temperature: 451
Time: 00:00:35
Iterations: 294
Passes: 4
Accepts: 56%
Rejects: 0%

Capture
system_6

Figure 7.7 Optimization controls

8. Performance Analysis

With all elements of the A-Vu system defined, it is now possible to validate the performance of the system and its associated visualization methods. Using a suite of test samples described below in Section 8.1, a performance analysis was conducted for each of operations presented in Chapter 3 and Chapter 4. The results of this analysis are presented in Section 8.2 and Section 8.3, respectively. The performance of the evaluation function *J*(*C*) from Chapter 5 was characterized and is summarized in Section 8.4. A comparative analysis of the different optimization methods discussed in Chapter 6 was also performed and is presented in Section 8.5. The overall effectiveness of the A-Vu sequencing methods are presented in Section 8.6.

During the execution of this analysis, several obvious flaws with the prototype system implementation were identified and corrected. The majority of these flaws manifested during the analysis of the very large configurations and resulted in excessive compute times or storage allocation. The analysis also identified several areas where the prototype system's performance was worse than predicted due to the specifics of the implementation. An explanation of these discrepancies are included below.

8.1 Test Samples

To conduct the performance tests below, a suite of test samples was prepared. These samples were taken from actual software systems currently in operation [Sm'87]. All of these programs were developed in Ada. The small systems were developed by individuals while the larger systems were a cooperative team effort. The dependency structure analyzed was extracted from the context clause structure (i.e. with statements) of each program via the A-Vu LOAD command. A brief summary of each test sample is given here.

System 1:	Single node program, no dependencies, provided for comparison.
System 2:	Example system used throughout Chapter 1 and Chapter 2.
System 3:	Database report generation program.
System 4:	Database manipulation program.
System 5:	Parser/compiler program.
System 6:	Medium-size interactive application.
System 7:	Large aggregate application.
System 8:	Large process control/monitoring application.
System 9:	Very large aggregate application.

The size of each sample system is shown in Table 8.1. The variable *n* refers the number of nodes in the sample, *k* refers to the number of edges. Note that the number of edges in this sample set is approximately 2-5 times the number of nodes.

Table 8.1 Test Sample Sizes

Sample	n	k
System 1	1	0
System 2	8	11
System 3	19	31
System 4	35	70
System 5	65	210
System 6	103	208
System 7	206	560
System 8	459	2001
System 9	1021	4282

8.2 Manipulation Performance

The performance of each operation described in Chapter 3 is presented here. Since performance differences are more dramatic among the larger samples, only the timing measurements associated with the five largest test samples (5, 6, 7, 8, and 9) are tabulated below. These measurements were conducted using the A-Vu prototype system running on a standalone, single-user VaxStation 3100-76. This low to medium performance, complex instruction set workstation has a computational rating of approximately 7-8 MIPS (million instructions/second). Timing measurements were based on the system's clock and taken with 0.01 second resolution. In several instances, particularly with the small test samples, time measurements less than 0.01 second were recorded. The constant ∂t is used in the tables below whenever this occurred.

To validate operation performance, each timing measurement was compared against the number of nodes and against the number of edges in the system. A best-fit approximation of all timing data was applied using one of five different models:

- Constant (CON)
- Logarithmic (LOG)
- Linear (LIN)
- Exponential (EXP)
- Power (PWR)

The model yielding the highest correlation is indicated in each table. The measured computational complexity of each operation is given in the last column of each table.

Initialization

The performance of the initialization operations (Section 3.1) is shown in Table 8.2. Although none of these operations are directly accessible by the user, the performance of these operations is crucial as they comprise the fundamental primitives used by all other high-level operations. The A-Vu system was appropriately instrumented to collected this information. As expected, the majority of these operations completed in constant time. The DELETE_NODE, CREATE_EDGE, and DELETE_EDGE operations completed in small, but linear time as predicted in Section 3.1 due to the additional edge and layout data structure maintenance that is required. Note, however, that the DELETE_LAYOUT operation performs in constant time rather than linear time as predicted. This is explained by the fact that only a small number of layouts (i.e. <3) were associated with the default configurations used by the test.

Table 8.2 Initialization Operation Performance

Operation	5	6	7	8	9	Fit	Θ
INITIALIZE	∂t	∂t	∂t	∂t	∂t	CON	c
CREATE_NODE	∂t	∂t	∂t	∂t	∂t	CON	c
DELETE_NODE	∂t	0.01	0.01	0.10	0.07	LIN	n
CREATE_EDGE	0.01	0.02	0.02	0.05	0.13	LIN	n
DELETE_EDGE	0.01	0.02	0.02	0.06	0.12	LIN	n
SET_NODE_ATTRIBUTE	∂t	∂t	∂t	∂t	∂t	CON	c
SET_EDGE_ATTRIBUTE	∂t	∂t	∂t	∂t	∂t	CON	c
CLEAR_NODE_ATTRIBUTE	∂t	∂t	∂t	∂t	∂t	CON	c
CLEAR_EDGE_ATTRIBUTE	∂t	∂t	∂t	∂t	∂t	CON	c
CREATE_LAYOUT	∂t	∂t	∂t	∂t	∂t	CON	c
DELETE_LAYOUT	∂t	∂t	∂t	∂t	∂t	CON	c
CREATE_BINDING	∂t	∂t	∂t	∂t	∂t	CON	c
DELETE_BINDING	∂t	∂t	∂t	∂t	∂t	CON	c

Editing

The performance of the editing operations described in Section 3.2 is shown in Table 8.3. The RECONNECT, INSERT_NODE, and DELETE_NODE operations have been excluded since they are embedded and used only by the CUT and PASTE operations. The linear-time rather than constant-time performance of the SET_NODE_POSITION, SELECT, and DESELECT operations is attributed to the linear-list implementation of sets in the prototype. While the CUT and PASTE operations performed extremely well for large configurations, their performance began to degrade with very large configurations. This was tracked to the $\Theta(|V|^2)$ component in the RECONNECT operation. The complexity measures for CUT and PASTE in Table 8.3 indicate the range computed with respect to nodes and with respect to edges. This format will be used in subsequent tables below.

Table 8.3 Editing Operation Performance

Operation	5	6	7	8	9	Fit	Θ
SET_NODE_POSITION	0.04	0.06	0.12	0.27	0.64	LIN	n
SELECT	∂t	∂t	∂t	0.01	0.02	LIN	n
DESELECT	∂t	∂t	∂t	0.01	0.02	LIN	n
CUT	0.05	0.10	0.37	1.64	9.25	PWR	$n^{1.1}$ .. $n^{1.4}$
COPY	∂t	0.01	0.02	0.05	0.10	LIN	n
PASTE	0.06	0.11	0.34	1.47	8.86	PWR	$n^{1.1}$ .. $n^{1.4}$

Selection

Table 8.4shows the performance of the selection operations described in Section 3.3. The majority of these operations performed as predicted, many demonstrating linear behavior as desired. Since an adjacency list data structure was not implemented in the prototype, the SELECT_ROOT, SELECT_LEAVES, and SELECT_BODY operations approached $\Theta(|V|^2)$ performance as expected. Unexpected however, SELECT_WEAK, SELECT_STRONG, and SELECT_CYCLIC, SELECT_MAX_IN, SELECT_MAX_OUT, and SELECT_MAX_IN_OUT each approached $\Theta(|n|^2)$ performance rather than linear performance. Upon re-examining their definitions, an assumption in the original analysis was discovered regarding the availability of the node adjacency list.

Although the current implementation of the A-Vu system incorporates a series of node and edge lookup caches, a full adjacency list implementation is not supported by the prototype. While this may seem like a minor enhancement, the ability to cut and paste nodes and dynamically inherit edges and attributes from other nodes contained in other spaces complicates this structure significantly. Based on these performance results, however, a composite adjacency list structure is being examined as a subsequent A-Vu refinement.

Table 8.4 Selection Operation Performance

Operation	5	6	7	8	9	Fit	Θ
SELECT_ALL	∂t	∂t	0.01	0.01	0.02	LIN	n
SELECT_NONE	∂t	∂t	0.01	0.01	0.02	LIN	n
SELECT_INVERSE	∂t	∂t	0.01	0.01	0.02	LIN	n
SELECT_ROOT	0.06	0.17	1.03	3.57	24.81	PWR	$n^{1.1}$ .. $n^{1.5}$
SELECT_LEAVES	0.17	0.40	2.04	5.90	56.80	PWR	$n^{1.2}$ .. $n^{1.6}$
SELECT_BODY	0.19	0.46	2.57	9.33	63.10	PWR	$n^{1.4}$ .. $n^{1.8}$
SELECT_WEAK	0.18	0.26	1.11	4.57	31.57	PWR	$n^{1.3}$ .. $n^{1.4}$
SELECT_STRONG	0.18	0.44	1.68	8.18	40.82	PWR	$n^{1.2}$ .. $n^{1.6}$
SELECT_CYCLIC	0.18	0.45	1.66	8.64	39.84	PWR	$n^{1.2}$ .. $n^{1.6}$
SELECT_RELATIVE	0.04	0.09	0.35	1.70	9.01	LIN	n
SELECT_ABSOLUTE	0.04	0.09	0.36	1.75	9.06	LIN	n
SELECT_REFERENCED	0.08	0.18	0.71	3.40	18.8	LIN	n
SELECT_NAME	0.01	0.02	0.04	0.15	0.29	LIN	n
SELECT_PATTERN	0.68	1.02	2.01	4.62	10.04	LIN	n
SELECT_NODE_ATTRIBUTE	∂t	∂t	0.01	0.01	0.02	LIN	n
SELECT_EDGE_ATTRIBUTE	0.01	0.01	0.01	0.06	0.09	LIN	n
SELECT_MAX_IN	0.19	0.46	1.79	8.51	74.28	PWR	$n^{1.3}$ .. $n^{1.6}$
SELECT_MAX_OUT	0.20	0.45	1.72	8.35	42.38	LIN/PWR	n .. $n^{1.3}$
SELECT_MAX_IN_OUT	0.30	0.76	2.98	22.70	74.93	PWR	$n^{1.3}$ .. $n^{1.7}$

Arrangement

All of the arrangement operations described in Section 3.4 performed as predicted with the exception of ARRANGE_BREATH, ARRANGE_LATERAL, and ARRANGE_ADJUSTED. While the performance of ARRANGE_BREADTH is still acceptable for very large configurations, a close examination of the algorithm implementation reconfirmed its linear time implementation, but identified an inefficiency that was due to the lack of rapid edge lookup algorithm for each node. Hence, it is believed that ARRANGE_BREADTH, as implemented, will converge to linear time given an extremely large configuration. The relatively low exponent associated with its computed complexity confirms this suspicion. The ARRANGE_LATERAL and ARRANGE_ADJUSTED operations, however, yielded poorer, unexpected performance again due to the assumption of an edge adjacency list. The performance data for each arrangement operation is shown in Table 8.5.

Table 8.5 Arrangement Operation Performance

Operation	5	6	7	8	9	Fit	Θ
ARRANGE_DEFAULT	0.01	0.02	0.02	0.05	0.12	LIN	n
ARRANGE_HIERARCHICAL	0.73	4.89	10.21	204.82	337.22	PWR	$n^{1.5}$ .. $n^{1.9}$
ARRANGE_DEPENDENT	0.32	1.69	3.36	46.06	184.90	PWR	$n^{1.4}$ .. $n^{1.8}$
ARRANGE_LAYERED	0.69	1.07	4.94	13.89	98.27	PWR	$n^{1.4}$ .. $n^{1.7}$
ARRANGE_BREADTH	0.05	0.14	0.43	2.18	19.69	PWR	$n^{1.1}$ .. $n^{1.4}$
ARRANGE_DEPTH	0.13	0.21	0.82	6.14	17.38	LIN	n
ARRANGE_UNIFORM	0.04	0.06	0.14	0.30	0.62	LIN	n
ARRANGE_CENTERED	0.04	0.04	0.10	1.31	1.97	LIN	n
ARRANGE_LATERAL	0.14	0.55	3.01	3.76	18.05	PWR	$n^{1.2}$ .. $n^{1.5}$
ARRANGE_ADJUSTED	0.14	0.51	1.48	3.83	17.75	PWR	$n^{1.2}$ .. $n^{1.5}$

Reduction

Among all of the reduction operations from Section 3.5, only the REDUCE_STRONG, REDUCE_WEAK, and REDUCE_RESTRICTED performed completely as predicted. Each of these operations were executed using default arrangements of the initial configurations. The REDUCE_SELECTED test was conducted by cutting every node in the initial configuration and pasting them into a single composite node. The quadratic behavior exhibited by REDUCE_SELECTED is therefore attributed to the $\Theta(|V|^2)$ overhead of the CUT and PASTE operations identified above.

While the REDUCE_SELECTED operation performed worse than expected, the computed complexities of the REDUCE_NAMES and REDUCE_IMPLIED operations actually appeared better. The linear behavior of both of these operations is most likely attributed to the extensive constant time overhead required in assembling the large number of layouts and bindings associated with these operations. An extremely large configuration is required to confirm these suspicions.

The REDUCE_ATTRIBUTES operation was excluded from the analysis since it is equivalent to a SELECT_NODE_ATTRIBUTES, REDUCE_SELECTED sequence. The results of the reduction operation measurements is shown in Table 8.6.

Table 8.6 Reduction Operation Performance

Operation	5	6	7	8	9	Fit	Θ
REDUCE_SELECTED	0.41	0.92	3.21	20.30	81.75	PWR	$n^{1.5} .. n^{1.9}$
REDUCE_WEAK	0.10	0.30	6.55	23.84	81.64	LIN	n
REDUCE_STRONG	0.18	0.41	1.61	12.09	68.76	PWR	$n^{1.5} .. n^{1.9}$
REDUCE_NAMES	0.81	1.92	6.23	43.59	175.66	LIN	n
REDUCE_IMPLIED	1.08	2.36	7.94	49.97	269.49	LIN	n
REDUCE_RESTRICTED	0.32	0.65	2.79	12.62	69.71	PWR	$n^{1.5} .. n^{1.9}$

Compaction

The compaction operations presented in Section 3.6 all produced linear results. These simple algorithms are similar to the ARRANGE_UNIFORM operation described above. Each of these operations were applied to a configuration that had been uniformly arranged and repositioned a unit distance from each axis. Table 8.7 contains a summary of the results.

Table 8.7 Compaction Operation Performance

Operation	5	6	7	8	9	Fit	Θ
COMPACT_X	0.01	0.01	0.07	0.18	1.23	LIN	n
COMPACT_Y	0.01	0.01	0.07	0.09	1.25	LIN	n
COMPACT_Z	0.01	0.01	0.09	0.06	1.40	LIN	n
COMPACT_VOLUE	0.01	0.01	0.01	0.01	0.20	LIN	n

Archival

The performance of the archival operations is shown in Table 8.8. The LOAD measurements were performed starting with a null configuration. The SAVE and OPEN operations were performed upon completion of an ARRANGE_DEFAULT operation following the LOAD. The performance measurements for the OPEN operation are currently not available due to recent changes in the configuration definition and the ASL language. All of the operations yielded linear time performance as expected. The LOAD operation, however, appears disproportionate to the other operations. This performance difference was traced to the initialization of several large cache data structures used in support of rapid node/edge references. The export operations were not included in the analysis, but are nearly identical to the SAVE operation.

Table 8.8 Archival Operation Performance

Operation	5	6	7	8	9	Fit	Θ
OPEN							
SAVE	0.70	1.17	1.95	4.99	11.52	LIN	*n*
LOAD	0.78	1.31	3.14	17.35	56.60	LIN	*n*
CLOSE	0.07	0.15	0.41	2.24	11.12	LIN	*n*

8.3 Viewing Performance

Of all the viewing operations identified in Chapter 4, only the REFRESH operation could not be performed in constant time. Nearly all of the other operations involve the manipulation of viewing variables which require negligible time to perform. These variables are used primarily for initialization of the viewing transformations that are applied during each REFRESH operation. Regardless of the specific viewing parameter value, these transformations are applied equally each pass. As a result, the REFRESH operation remains nearly constant as these parameters are changed. A dramatic performance change, however, can be observed by applying filtering. Table 8.9 contains the timing measurements of the REFRESH operation with and without edge filtering enabled. Note that for very large configurations, REFRESH performance, although linear time, becomes nearly intolerable for interactive applications. The filtered REFRESH operation shows a noticeable improvement. This performance difference is traced to the overhead involved in simply processing and displaying such a large number of edges. Due to its linear time behavior, the REFRESH operation is expected to be proportionately faster on a higher performance workstation.

Table 8.9 Viewing Operation Performance

Operation	5	6	7	8	9	Fit	Θ
REFRESH (unfiltered)	0.43	0.63	1.43	5.91	21.78	LIN	n
REFRESH (filtered)	0.21	0.22	0.45	1.05	2.23	LIN	n

8.4 Evaluation Performance

The performance of the primary evaluation functions described in Chapter 5 that are currently implemented within A-Vu are presented in Table 8.10. These measurements were taken after a default arrangement of each configuration was generated. The node and edge counting functions performed in linear time as expected. Although the total number of nodes and edges in the configuration are directly accessible in the configuration data structure implementation, these values must be computed as they pertain to a particular layout view or cross-section. The values of these functions are cached by the A-Vu system for later reference. Consequently, the performance of the volumetric and cyclomatic functions yield constant time performance. With the exception of J_E, the remaining functions yielded linear time performance as expected. The crossing function performed in $\Theta(|E|^2)$, also as expected. It is obvious from Table 8.10, however, that the use of J_E for very large, unreduced configurations will result in unacceptable interactive performance. Whenever possible, the reduction operations from Section 3.5 should be applied to a configuration first to consolidate nodes and significantly reduce the number of visible edges.

Table 8.10 Evaluation Function Performance

Function	5	6	7	8	9	Fit	Θ
Nodes - J_n	∂t	∂t	∂t	∂t	0.01	LIN	n
Edges - J_k	∂t	∂t	∂t	0.03	0.08	LIN	n
Volumetric - J_V	∂t	∂t	∂t	∂t	∂t	CON	c
Cyclomatic - J_M	∂t	∂t	∂t	∂t	∂t	CON	c
Connectivity - J_I	0.01	0.01	0.04	0.09	0.21	LIN	n
Distance - J_D	0.05	0.05	0.23	0.40	0.89	LIN	n
Proximity - J_P	0.02	0.02	0.06	0.18	0.41	LIN	n
Crossings - J_E	2.46	4.26	16.31	230.0	900.0	PWR	$n^{1.7} .. n^{2.1}$
Hierarchcy - J_H	0.03	0.02	0.05	0.19	0.36	LIN	n
Layering - J_L	0.02	0.03	0.04	0.16	0.37	LIN	n
Reflectivity - J_R	0.02	0.02	0.06	0.18	0.41	LIN	n

Recall from Table 8.1 that the ratio of edges to nodes among the larger samples was typically in the range of 2 to 5. Examination of a much larger set of software samples reaffirmed this linear relationship. The analysis of over 200 programs written by numerous multi-developer teams yielded a maximum edge to node ratio of 10. This linear relationship was further confirmed in each of the tables above. Whenever node versus timing information was best fit to a power curve, a similar best fit appeared on the number of edges.

Also of interest is the relationship between the number of edges and the number of edges crossings in a default configuration. The information obtained from the test samples is shown in Table 8.11. When best fit to a curve, a relationship approaching quadratic performance (i.e. $J_E \approx 0.2 \times k^{1.9}$) is again observed. These empirical relationships are important as they allow an upper bound to be established on the maximum number of system elements that should be represented within a given layout for a particular class machine.

Table 8.11 Edge Crossing Relationship

Sample	k	Crossings
System 1	0	0
System 2	11	35
System 3	31	118
System 4	70	330
System 5	210	3036
System 6	208	6423
System 7	560	31,870
System 8	2001	343,310
System 9	4282	1,763,398

Let $J_E \approx ck^2$ where c is an constant and let r equal the expected ratio of edges to nodes. Since $k \approx rn$, then $J_E \approx c(rn)^2$. Solving for n yields the following empirical relationship:

$$n \approx \sqrt{\frac{J_E}{rc}}$$

Let x be the maximum number of crossings that can be calculated per second and let δt be the maximum acceptable user response time delay in seconds. The maximum allowable value for J_E is therefore $x\ \delta t$. Substituting for J_E yields the following:

$$n \approx \sqrt{\frac{x\delta t}{rc}}$$

Assume that reasonable interactive performance is say, one second, and that maximum expected edge to node ratio is 5 as in Table 8.1. Assuming $c = 0.2$ as above, the maximum recommend value for n is then:

$$n \approx \sqrt{\frac{x\delta t}{rc}} = \sqrt{\frac{x \cdot 1}{5(.2)}} = \sqrt{x}$$

From the data in Table 8.10 and Table 8.11, the crossing calculation rate x is approximately 2000 per second (e.g. 1,763,398/900.0). Thus the maximum recommended number of nodes per layout on the machine used for this analysis is approximately 45. Using contemporary high-performance workstation technology, this number can be increased to approximately 240, providing reasonable performance for most large-sized programs.

8.5 Optimization Performance

In contrast to its direct algorithm-based operations (i.e. Chapter 3), the iterative improvement methods employed by the A-Vu system appear costly. However, deterministic polynomial-time algorithms are not always available nor possible depending upon the optimization criteria requested. Fortunately, A-Vu provides a great deal of flexibility that allows a user to carefully control the computational investment to be expended.

To demonstrate this flexibility, the second test sample (System_2) originally presented in Example 1.1 and Example 1.2 will be used. An initial configuration for this system is constructed using the following ASL sequence:

```
load system_2.lis
arrange default
select root
arrange breadth
compact volume
arrange depth
arrange default
```

The layout shown in Figure 8.1 illustrates the results of this sequence. The A-Vu command **option identify** was used to enable node identification. Note the resemblance to Figure 1.1.

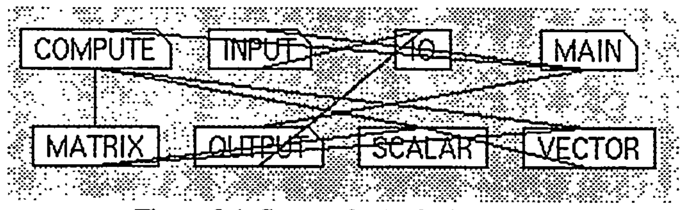


Figure 8.1 System 2 - Default Layout

This initial configuration can now be transformed according to the desired layout criteria and cooling schedule. Using the control panel shown in Figure 7.5, the user selects the appropriate evaluation function weights. The control panel shown in Figure 7.6 is used to adjust the cooling schedule. For these examples, a cooling rate of $\alpha = 0.90$, an initial temperature of $T = 500°$, $k = 0.010$, and $m = 100$ iterations per pass were selected. The node placement mode was initially set to random and the node placement constraint was set to planar (i.e. two-dimensional, unconstrained). Figure 8.2 shows the results of the optimization process using only the distance criteria J_D (i.e. all other evaluation function weights are set to 0). Note that the process effectively found a graph articulation point and clustered the eight nodes into two subgraphs. The top cluster contains all of the input/output support nodes while the bottom cluster contains all the computational nodes.

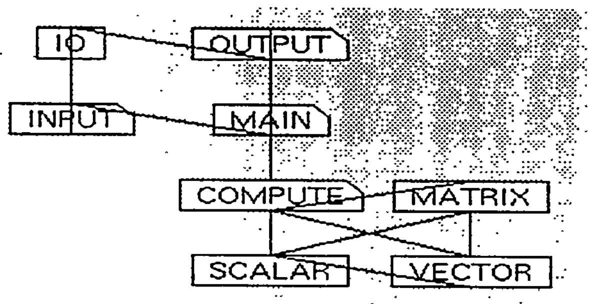


Figure 8.2 System_2 - Distance Minimization

While the clustering information obtained from distance minimization is useful, the general hierarchical structure of the system is difficult to extract from Figure 8.2. This structure is much more visible in Figure 8.3 where only hierarchy minimization was performed using J_H. Note that the process identified five distinct levels within the system, corresponding to the five levels originally desired as in Figure 1.2.

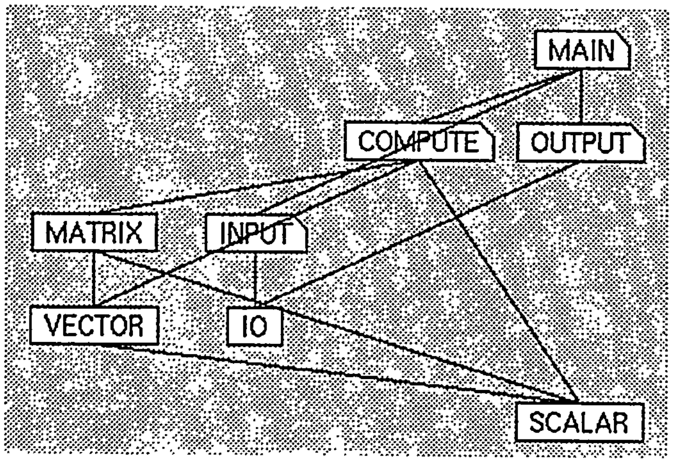


Figure 8.3 System_2 - Hierarchy Optimization

While hierarchy optimization is useful in identifying dependency levels, knowledge of the system's abstraction layers is not readily visible in Figure 8.3. This was partially rectified in Figure 8.4 by performing only layering optimization using J_L. Three distinct layers appear; the top layer containing program input/output and control elements, the middle layer containing the compute element, and the bottom layer containing the closely coupled linear algebra components.

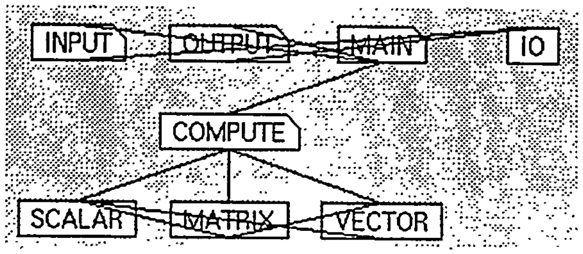


Figure 8.4 System_2 - Layering Optimization

While the layouts in figures 8.1 through 8.4 reveal some useful information, the organization of these diagrams still appear complicated. This visual complexity can be reduce via edge crossing minimization using J_E (Figure 8.5) or symmetry (reflectivity) optimization using J_R (Figure 8.6).

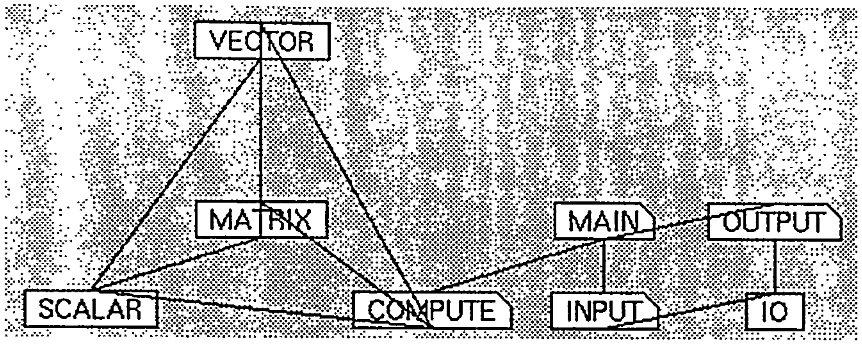


Figure 8.5 System_2 - Edge Crossing Minimization

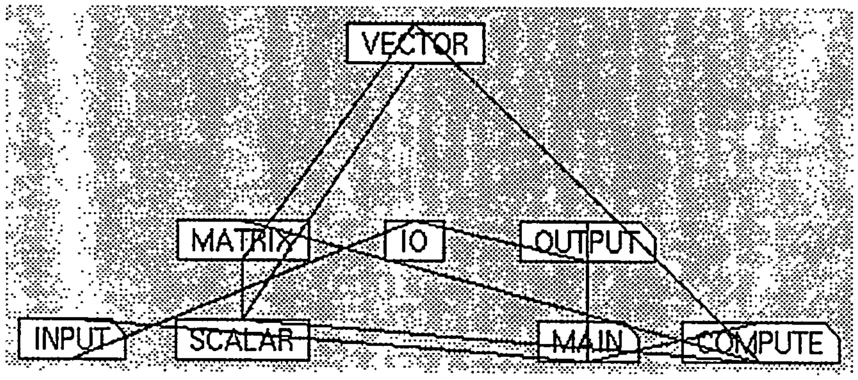


Figure 8.6 System_2 - Symmetry Optimization

While each of the above diagrams possess their own individual strengths and weaknesses, a layout using a composite of these criteria is actually desired. The layout in Figure 8.7 was generated with the weights shown in Table 8.12. To bring out the dependency and abstraction structure, hierarchy and layering were weighted most heavily. Edge crossings and reflectivity were weighted next for visual appeal. Distance was weighted lowest to insure a compact diagram without jeopardizing the other factors. Note the presence of the reflectivity component in the positioning of the SCALAR and VECTOR components. Ideally, these components should be centered beneath the MATRIX component to convey their close relationship. This can be accomplished by further reducing the reflectivity weight, or more effectively, by introducing an attribute-based function that captures the MATRIX-VECTOR-SCALAR relationship.

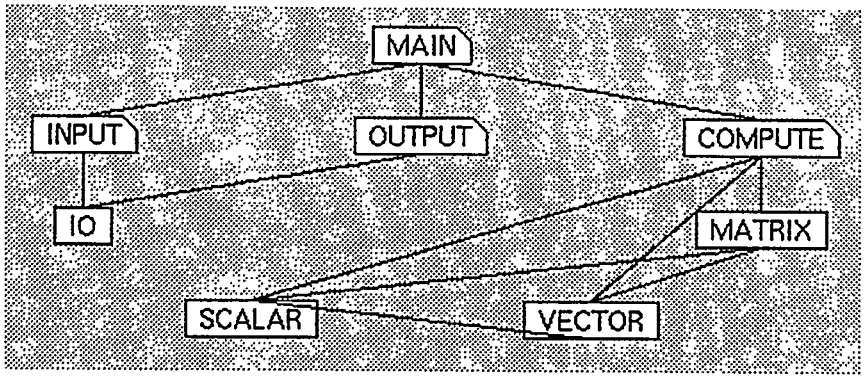


Figure 8.7 System 2 - Composite Optimization

Table 8.12 Weight Assignments for Figure 8.7

Criteria	Weight
Distance - J_D	0.10
Hierarchy - J_H	1.00
Layering - J_L	0.80
Crossings - J_E	0.75
Reflectivity - J_R	0.50

The total number of iterations required to generate the diagrams in Figure 8.2 through Figure 8.6 are shown in Table 8.13. Due to the random nature of the process, these counts vary with each run and can be tuned to the desired level of performance by adjusting the cooling rate, freeze temperature, and pass iteration count using the schedule control panel. An energy profile of each optimization sequence is shown in Figure 8.8.

Table 8.13 System 2 Optimization Iterations Counts

Criteria	Count
Distance - J_D	2100
Hierarchy - J_H	500
Layering - J_L	1000
Crossings - J_E	1200
Reflectivity - J_R	2600
Composite - $J(C)$	2700

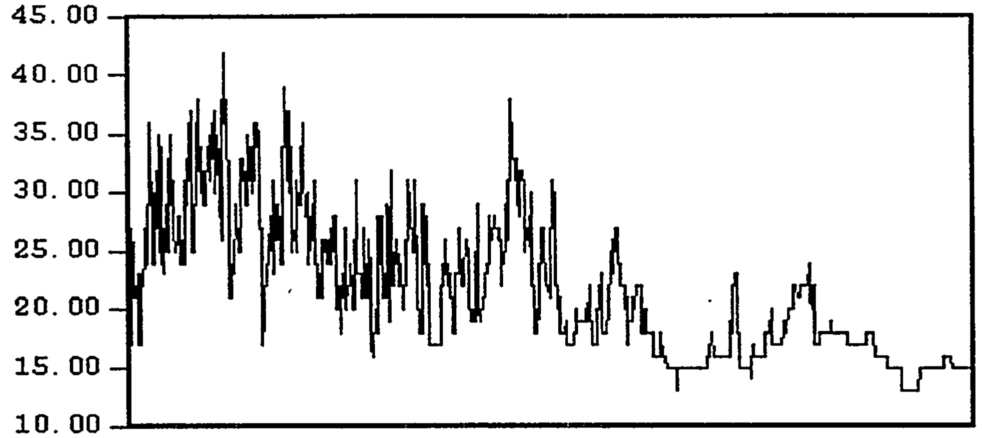


Figure 8.8(a) Distance Energy Profile

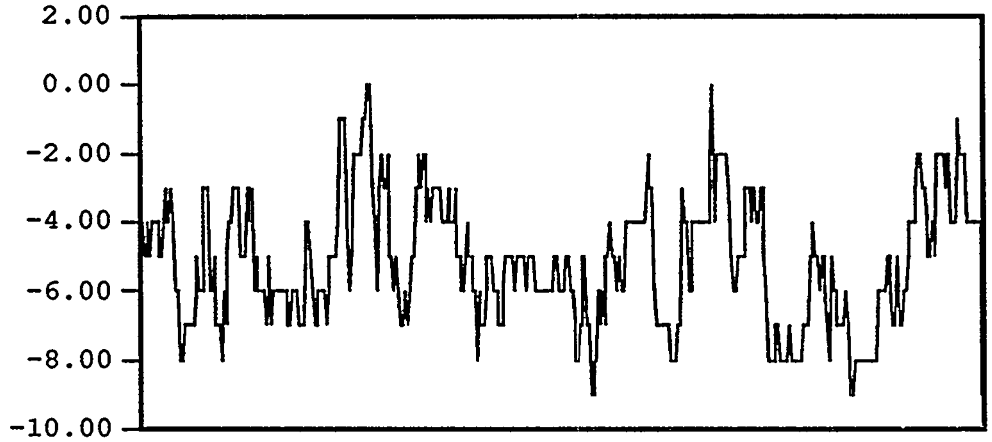


Figure 8.8(b) Hierarchy Energy Profile

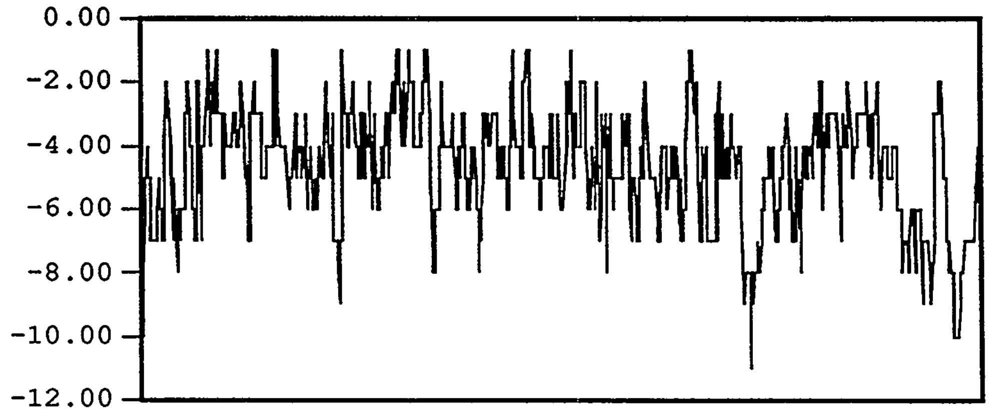


Figure 8.8(c) Layering Energy Profile

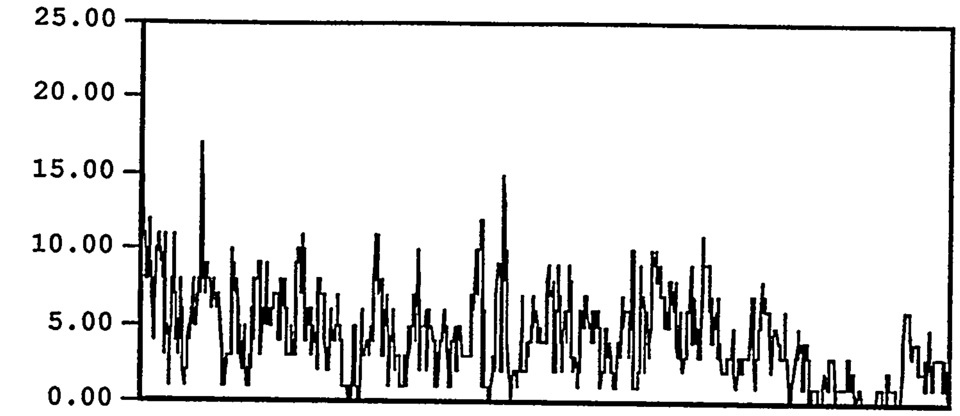


Figure 8.8(d) Crossing Energy Profile

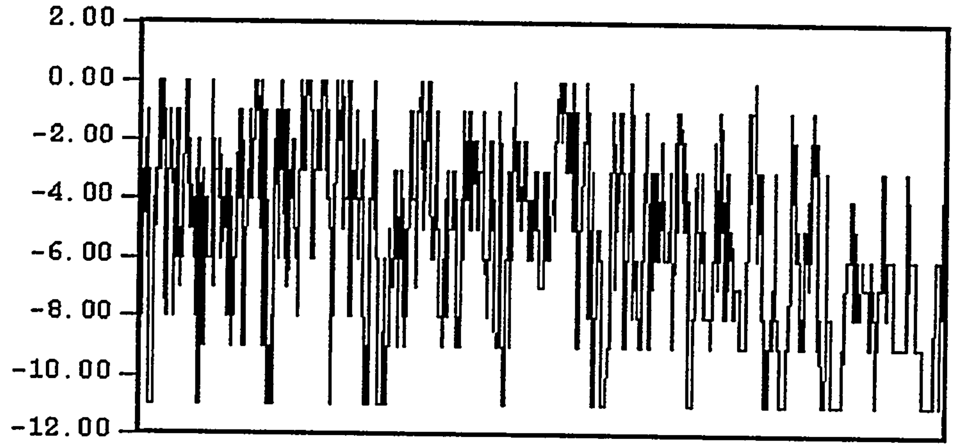


Figure 8.8(e) Reflectivity Energy Profile

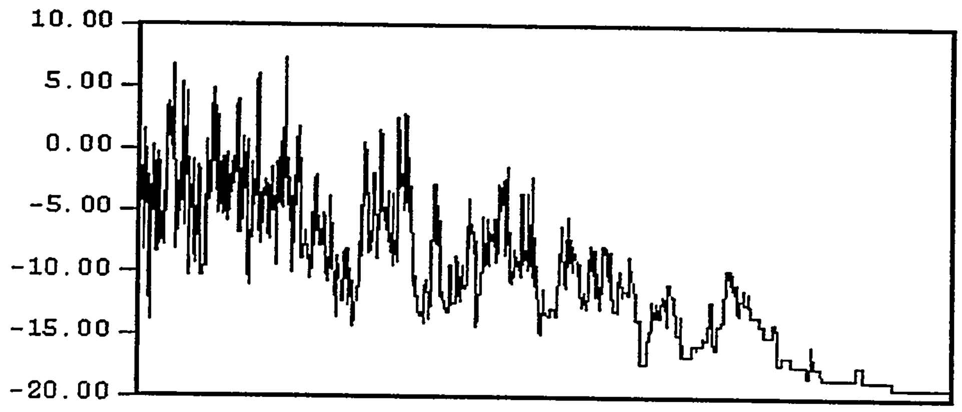


Figure 8.8(f) Composite Energy Profile

Figure 8.8 System 2 - Optimization Sequences

Examining the energy profiles in Figure 8.8 reveals some interesting results. Only the distance and composite profiles demonstrate a clear convergence to a minimum energy state. From Table 8.13, it is known that these profiles represent a considerable number of iterations while those for hierarchy, layering, and edge crossing are in the range of 43% to 81% less. As a result of the higher number of iterations, the exponential limiting effect of higher energy transitions is more pronounced. The hierarchy, layering, and edge crossing profiles demonstrate the random aspects of this process at higher temperatures. The three optimization sequences terminated due to having meet the cooling requirement based on a repeated energy sample pattern as controlled by the cooling schedule panel.

Of unusual interest is the reflectivity profile which yields no apparent convergence except perhaps at the very end. This behavior is attributed to the very narrow energy span and the discontinuity inherent in the reflectivity function. Similar to the edge crossing function, a single displacement of a node can dramatically alter the energy level within this energy range. Configurations which offer a much broader energy differential and functions with a more continuous nature produce much cleaner convergence profiles. These effects appear naturally as the number of nodes and edges in the system increases as seen below.

To demonstrate the effects of different placement modes, System 6 is used as an example. Because of the much larger configuration size, it is useful to first simplify the amount of work the optimization process will have to perform. This can be accomplished by first applying the ARRANGE_HIERARCHICAL, ARRANGE_DEPENDENT, or ARRANGE_LAYERED operations on a reduced configuration. When coupled with a layered placement constraint, these operations eliminate the need for hierarchical and/or layering minimization criteria. The linear option in the schedule control panel initiates this constraint. A reduced configuration of System 6 is prepared using the following sequence:

```
load system_6.lis
arrange default
select pattern junk.pat
cut
reduce implied
select root
arrange breadth
compact volume
arrange dependent
```

The energy profiles for the different placement modes are shown in Figure 8.9. The respective acceptance and rejection rates associated with each pass of the optimization algorithm is shown in Figure 8.10.

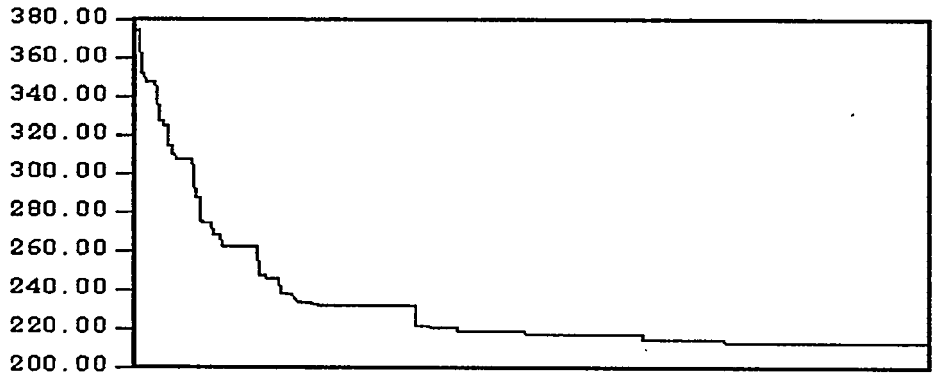


Figure 8.9(a) Hill Climbing

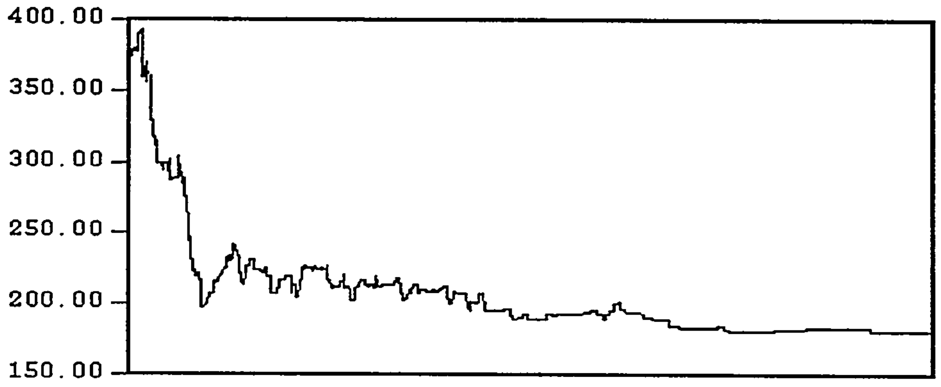
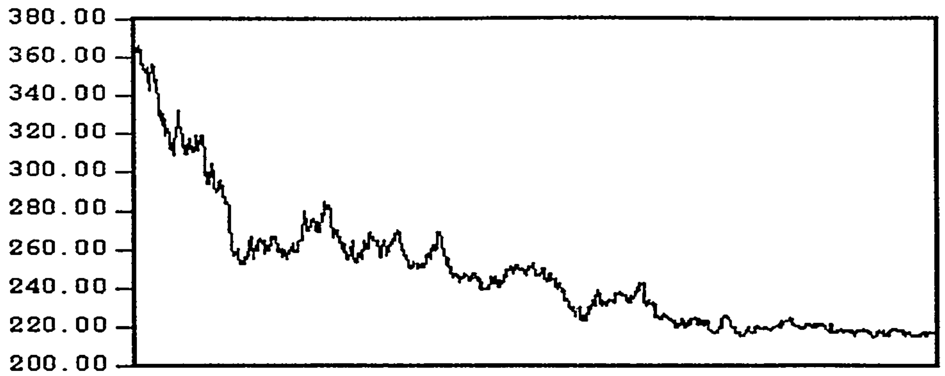


Figure 8.9(b) Random Placement



8.9(c) Random Displacement

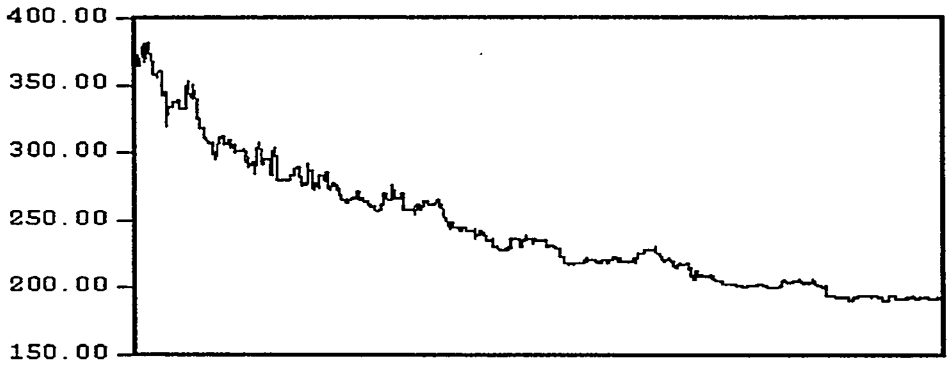


Figure 8.9(d) Controlled Displacement

Figure 8.9 System 6 - Node Placement Energy Profiles

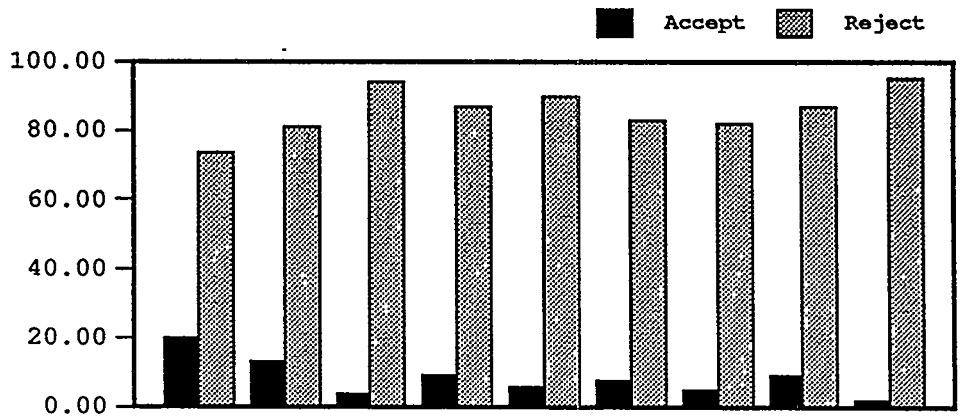


Figure 8.10(a) Hill Climbing

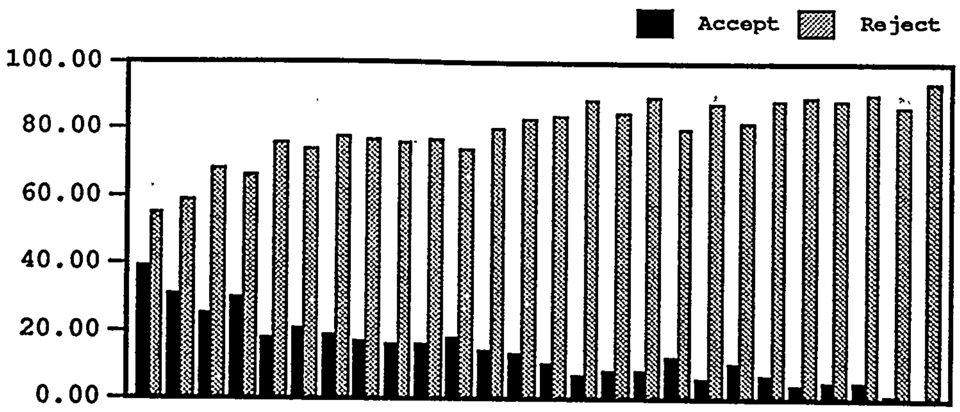


Figure 8.10(b) Random Placement

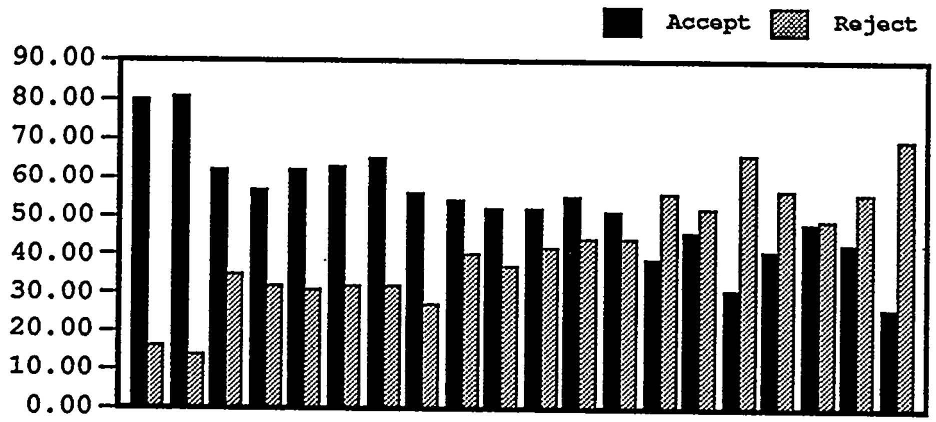


Figure 8.10(c) Random Displacement

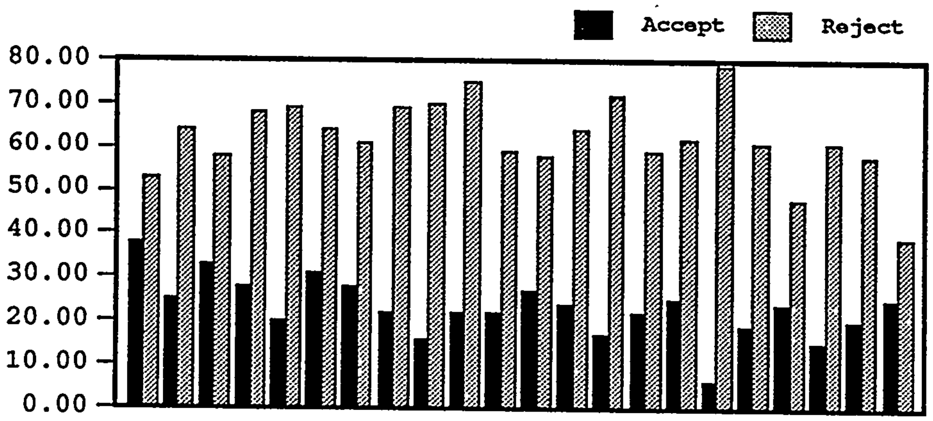


Figure 8.10(d) Controlled Displacement

Figure 8.10 System 6 - Placement Mode Acceptance/Rejection Rates

The results of the optimization process on System 6 using the four different placement modes is shown in Table 8.14. The count column contains the number of iterations the sequence executed before coming to completion. The energy column indicates the final energy state. The final column indicates the energy reduction from the initial energy state of 375.

Table 8.14 System 6 Optimization Results

Placement Mode	Count	Energy	Reduction
Hill Climbing	900	213	43%
Random	2600	181	52%
Random Displacement	2000	215	43%
Controlled Displacement	2200	191	49%

For this example, the simple hill climbing technique (i.e. $T = 0$) performed remarkably well. Based on repeated trials, the hill climbing technique appears to be particularly useful in making initial coarse energy reductions with a minimum computational investment. Because of the very low acceptance rate as seen in Figure 8.10(a), far fewer configuration alternatives are explored. Consequently, “fine tuning” of the configuration does not appear.

Surprisingly, the random placement scheme consistently produced the best (lowest energy) results, but tended to required more time to complete. At the start of the process, the placement generator is free to explore a broad range of possibilities. From Figure 8.9(b) it can be seen that it quickly reached a low energy state, but continued to explore higher energy alternatives. The high rejection rate and the continually decreasing acceptance rate that is reached and sustained after only a few passes confirms this observation.

Of the four methods, the random displacement performed the poorest with respect to both iteration count and final energy state. Throughout an optimization sequence, this method is restricted to small fixed displacements of nodes resulting in smaller incremental changes. As a result, this method tends to sustain a much higher acceptance rate. Much of this rate can be attributed to higher energy state transitions. Because of the smaller displacements, the ability to locate lower energy layouts which are substantially different from an initial configuration may not occur until much later in the sequence. Unfortunately, the cold temperatures at this time make the necessary intermediate high energy transitions less like to succeed.

The final technique represents an attractive compromise between the random and random displacement methods. At the start of the process, the controlled displacement technique behaves very similar to the random placement examining more diverse sections of the solution space. At cooler temperatures, this method begins to act more and more like the random displacement method, performing the necessary configuration fine tuning. The acceptance and rejection rates appear relatively constant throughout the entire process. While the end energy results is not always as low as what can be achieved by the random method, fewer iterations are generally required.

8.6 Sequencing Results

One of the most powerful features of the A-Vu system is its ability to integrate all of the different techniques presented above into a single execution sequence that can be saved, modified, and executed again at a later time. These sequences can be saved with a system as documentation and updated as major architectural modifications are adopted.

To illustrate this point, the discussion again returns to System 6 presented above. This system actually represents the software architecture for the A-Vu tool itself. This architecture has undergone numerous changes as the A-Vu model has been refined and new capabilities added. The sequence used for its visualization was developed early on and has been maintained as the system evolved. In the early days of its develop, only manual node manipulation and optimization were available as ASL components. Since then, ASL has grown much richer, providing considerably greater flexibility with many performance improvements. A typical sequence now begins with an OPEN or LOAD statement, followed by a default arrangement, a series of reductions, an optimization phase, and a final filtering phase. Figure 8.11 shows System 6 immediately after it initial loading.

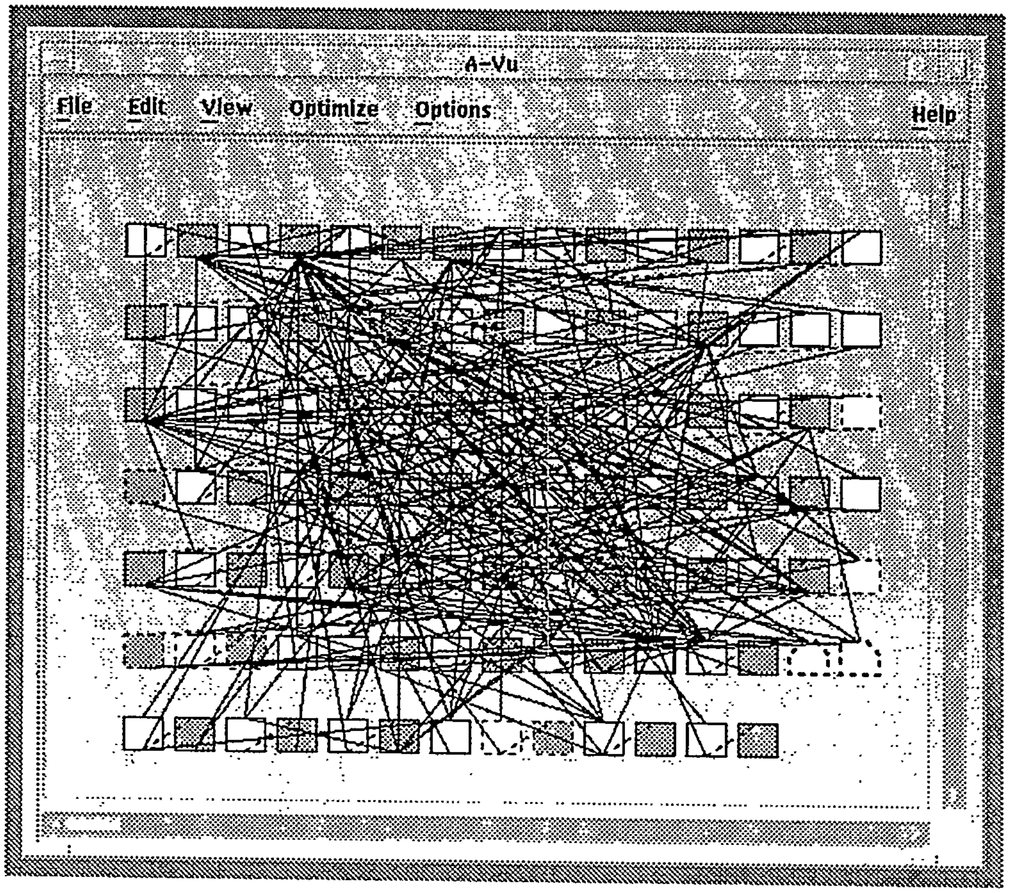


Figure 8.11 Default configuration of the A-Vu system (System 6)

The sequence currently used by A-Vu to visualize itself is the following:

```
load system_6.lis
arrange default
select pattern support.pat
reduce selected
modify support
select none
reduce implied
select none
find *callbacks
reduce selected
modify callbacks
select none
find *er
find *or
reduce selected
modify operations
arrange dependent
compact volume
optimize
option transitive off
```

The configuration which was generated as a result of this sequence is shown in Figure 8.12. The total time to run the entire sequence was 22 seconds. Of this time, 12 seconds were required to complete the optimization sequence.

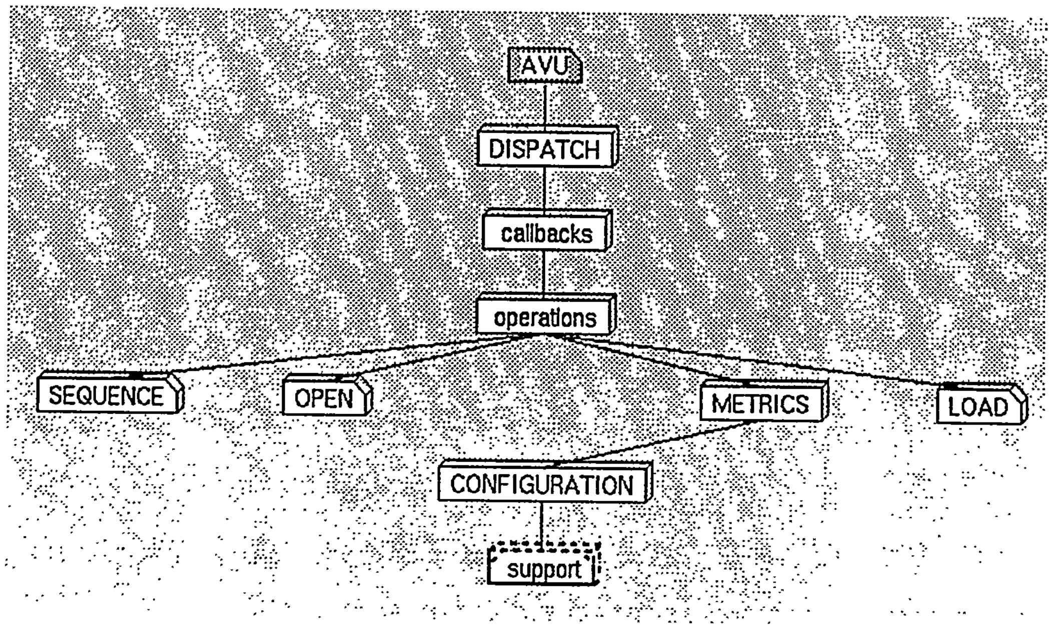


Figure 8.12 A-Vu System at Completion of Sequence

The difference in complexity between Figure 8.11 and Figure 8.12 is striking. Furthermore, no information was lost in the process. Via the A-Vu user interface, each node can be examined individually, filtering turned on and off, nodes continued to be cut and paste, etc. Using the methods, one final refinement which captures the designers high level design for the system is shown in Figure 8.13. This diagram was created from Figure 8.12 by removing the support tools, moving several elements into the operations node, and combining the METRICS and CONFIGURATION nodes into the database node. These final operations were initiated with a few quick mouse clicks and could easily be added to the automated sequence above for future use.

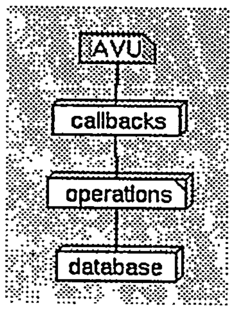


Figure 8.13 A-Vu Designer's Overview.

As a final example, the automated sequence described in Appendix C will now be applied to Figure 1.3 (System 8). The total time to run this entire sequence up to the optimization phase including the load operation was 3 minutes, 15 seconds. The optimizer was then run for approximately 15 seconds to reduce edge crossings and strengthen symmetry. Interim views of the system's configuration at various stages of the sequence are shown in Figure 8.14 and Figure 8.15. The final configuration at the completion of the sequence is shown in Figure 8.16. The total energy $J(C)$ of the system was reduced from 319,641 in Figure 1.3, to 54,089 in Figure 8.14, to 616 in Figure 8.15, and ended at $J(C) = 5$ in Figure 8.16. The configuration in Figure 8.16 was displayed with the node name identification option enabled.

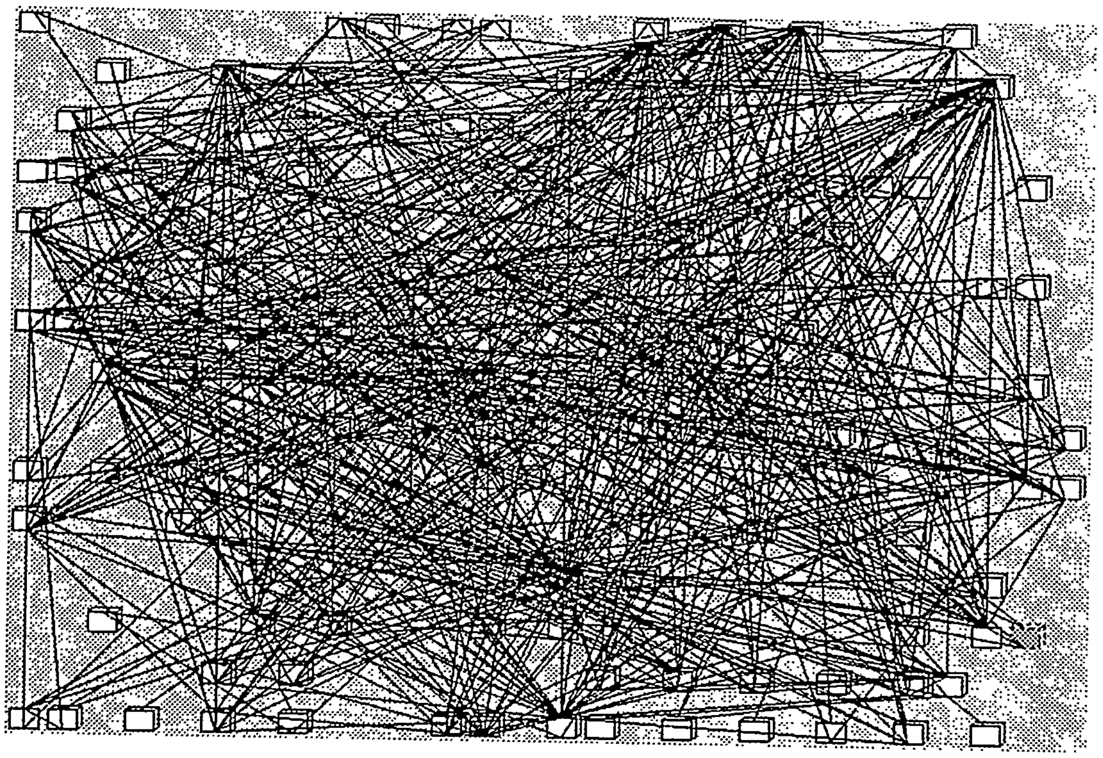


Figure 8.14 System 8 - Interim Configuration, $J(C) = 54,089$

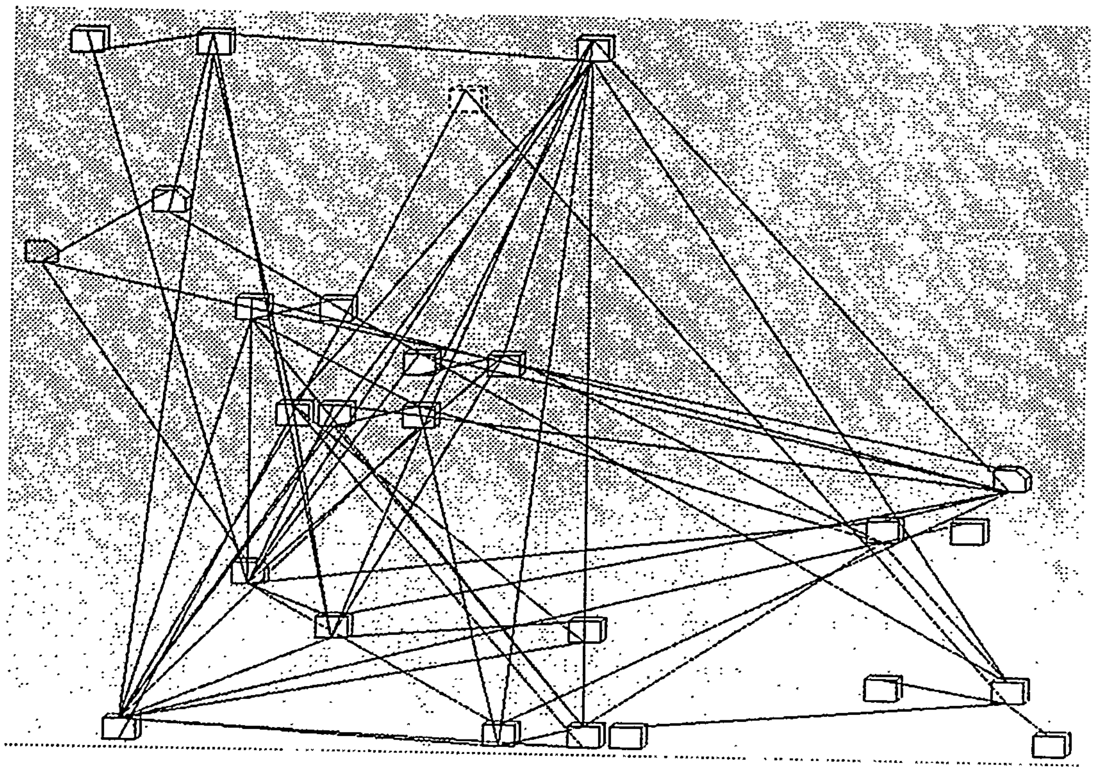


Figure 8.15 System 8 - Interim Configuration, $J(C) = 616$

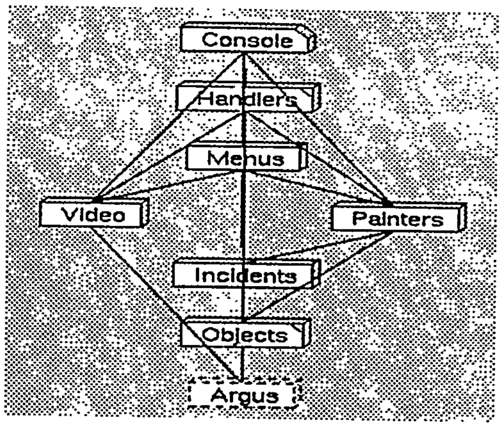


Figure 8.16 System 8 - Final Configuration, $J(C) = 5$

9. Conclusion

This dissertation is intended to serve as a comprehensive investigation of the dependency analysis process for complex systems. To meet the research objectives presented in Section 1.3, a model was initially developed for capturing dependency information and then subsequently refined to address a diverse range of issues that have limited traditional graph-based approaches. Based on this model, a collection of primitive operations were developed that defined an environment compliant with this model. These operations were then extended to address issues specifically relating to user interaction and visualization. Due to the potential performance risk associated with many graph theoretic problems, each operation was methodically defined within a framework for characterizing acceptable performance.

With a wealth of operations available, a quantitative set of complexity measures were then defined to help evaluate and direct the application of these operations. The combination of these operations and complexity measures provided the foundation for an optimization strategy and an automated sequencing mechanism. Integrating all of these concepts, the design of a successful prototype tool was constructed in accordance with the requirements presented in Section 1.4. Using this tool, a performance analysis was then conducted to validate the effectiveness of the original model. This analysis concluded favorably identifying areas for future algorithmic research. Reviewing Section 1.3 affirms that the original objects of this research have been met.

This investigation now concludes with some historical background on how this effort had unfolded (Section 9.1), a summary of the advantages this approach offers (Section 9.2), and some suggestions for continuing research and development (Section 9.3).

9.1 Concept Evolution

During the early phases of this effort, an approach was sought that could be used to gain a better understanding of how arbitrary complex systems are organized before, during, and particularly after they have been engineered. The original method proposed to extract various symmetry properties from systems and map these properties onto a group theoretic model. This method was founded on the premise that visual simplicity and design comprehension are closely related. Such an approach is of great theoretical interest due to the mathematical foundation (i.e. group theory) and related visual elegance than can be used to express very complicated structures. Unfortunately, actual systems in use exhibited relatively few properties of this nature that could be used to infer any specific organizational structure. Indeed, it was found that system organization involved numerous criteria that fell outside the limits of this framework. Consequently, the symmetry constraints were relaxed equating the problem to more traditional dependency analysis. Nevertheless, the ability to engineer new systems under this framework remains an open question.

Recasting the problem in directed graph parlance, an optimization strategy was sought that would allow the graphs to be minimized using an initial symmetry function suite. Simulated annealing was selected as the optimization approach of choice due to the ease with which it could be integrated and tailored in an interactive environment. Initial applications of this approach on sample systems led to the rapid identification of numerous other evaluation criteria. As this suite continued to expand, the limitations of iterative improvement techniques became readily apparent.

Simulated annealing and other related techniques are notorious for their computational expense. As new layout criteria were identified, a set of polynomial time algorithms for achieving specific layout objects were developed. The extensive set of operations outline in this dissertation are a result of that effort. The limitations of fixed layout criteria, however, has been clearly stated. The simulated annealing engine therefore remains quite useful in identifying and exploring new layout methods. The attribute mechanisms, composite layouts, meta-configuration, viewing operations, etc. were developed as a natural progression of this work.

9.2 A-Vu Advantages

The dependency visualization model that resulted from this work has several advantages over traditional graph layout techniques. While most graph layout techniques are based on fixed criteria, the approach developed here allows a user to select the desired layout criteria from a large criteria set. New layout criteria can be added to the system without modification to the model or its underlying optimization algorithm. The weighting of any individual criterion can be readily adjusted by the user. This enables a user to customize the visualization system for the particular problem at hand.

Unlike many graph layout techniques, the A-Vu model is specifically intended for interactive use. Traditional graph layout algorithms, a variety of software visualization tools, and the layout optimizer are all integrated into a single package. Consequently, the user is free to explore a system from many different perspectives, yet is capable of generating specific layouts.

In a typical session, the user might first apply a sequence of pattern matches to consolidate, reposition, or eliminate system elements of little or no consequence. Next the user may wish to perform a sequence of rudimentary graph algorithms such as a root node search and a hierarchical arrangement. An optimization sequence which constrains nodes to a particular plane could next be performed to reduce edges crossings and increase proximity. Based on the visual result, the user my wish to perform some additional editing operations and repeat the optimization sequence. A single session could involve numerous editing/optimization operations to help the user obtain the organizational information they desire.

While conventional graph layout techniques are generally constrained to examine only node and edge structure, the A-Vu model provides a mechanism which gives meaning to various nodes and edges. This mechanism enables the A-Vu system to address a much broader variety of layout concerns applicable to complex system understanding.

Finally, A-Vu's composite mechanism is a powerful tool for reducing system dependency. Many of the same systems engineering concepts in widespread use today can be implied with this construct. The ability to automatically track node containment, maintain node dependencies, and update node and edge attributes as nodes are moved throughout multiple visualization spaces is a significant improvement to previous planar graph methods. Under the A-Vu design, dependency structures are now multi-dimensional objects that may be freely explored, yielding a significant departure from traditional methods.

9.3 Future Directions

As an exploratory research effort, there are still many issues that remain to be addressed. While the A-Vu model and its current implementation provide a strong foundation for structural and functional system analysis, the unification of dynamic and behavior properties requires an additional research investment. Based on the discussion in Section 2.10, it appears that this can be accomplished within the existing A-Vu framework by treating layouts, attributes, bindings, and configurations in general as functions of time. This is perhaps the area of greatest of open research. Natural enhancements to the existing A-Vu system include the animation of layouts to track data flow, examine data structures, view state information, and follow execution paths.

To compensate for the computational inefficiencies of iterative improvement strategies, continuance of the evolutionary process employed throughout this research effort is envisioned. One of the most critical elements in this process has been the characterization of the objective function, $J(C)$. Additional components may continue to be incorporated in the function suite. Care must be taken to avoid the introduction of a seemingly endless array of options. The effectiveness of each component as it relates to problem dependent factors must be explored in unison as endeavored in Chapter 3.

As important new evaluation criteria and useful layout sequences are identified and tested, traditional algorithms or heuristic equivalents should be devised and incorporated into the A-Vu system. Hence, ongoing expansion to the operation "catalog" in Chapter 3 is envisioned. Comparable expansion of the sequence language is also anticipated. Of particular is the need for a recursive iteration construct in the ASL language that would allow sub-sequences to be applied to composite structures and their dependents. A macro facility for constructing ASL sequences by user example is also desired.

Numerous open issues regarding the optimization process are present as well. What size and type of visualization space is needed to represent an arbitrary software system? Are there more effective methods for placing nodes and generating new configurations? How can optimization parameters be automatically selected? What are adequate tests for minimum conditions and how can these conditions be selected automatically? Can this process be applied over an entire system rather than just a single layout at a time. Finally, how can an adequate computational balance be maintained between the energy function J, the node placement techniques, and the automated sequencing task to obtain the most effective, high-performance implementation. These issues currently remain under investigation.

The application of genetic algorithms as an alternative node placement method remains an open issue. Although an algorithmic framework for their application was devised, an actual implementation was not completed. A simple hill-climbing variant exists as a placeholder in the current A-Vu implementation. Consequently, a comparative analysis between genetic placement and the other approaches has not been completed.

The intended result of future efforts is the continuing expansion and improvement of the A-Vu model and its associated implementation. As more challenging organization structures are encountered, the A-Vu model must similarly evolve. By applying these techniques to existing systems, reverse engineering their designs, and extracting vital dependency information, insight into new organization techniques may ultimately be discovered enabling new system constructs to be formulated. In conjunction, new visualization techniques are sought that move beyond basic node/edge structures in concisely and elegantly depicting systems of ever increasing complexity. The notion of bus-oriented, distributed, or perhaps even quantum mechanical dependency architectures [Fe'8x] all offer exciting possibilities for the future.

References

[Ah'83] Aho, A, J. Hopcroft, J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[Ar'88] Armstrong, M, *Groups and Symmetry*, Springer-Verlag, 1983.

[Bi'89] Biggerstaff, T., "Design Recovery for Maintenance and Reuse," *Computer*, pp. 36-48, July 1989.

[Bi'91] Bischof, C. and J. Hu, "Utilities for Building and Optimizing a Computational Graph for Algorithmic Decomposition," ANL/MCS-TM-148, Argonne National Laboratory, April 1991.

[Bl'8x] Blattner, M., Sumikawa, D., Greenberg, R., "Earcons and Icons: Their STructure and Common Design Principles," *Human-Computer Interaction*, Vol. 4, pp. 11-44, 1989.

[Br'88] Brown, M., "Exploring Algorithms Using Balsa-II," *Computer*, pp. 136-157, May 1988.

[BG'92] Brown, P. and T. Gargiulo, "An Object Oriented Layout for Directed Graphs," *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, pp. 164-171, May 1992.

[BS'92] Brown, P. and D. Stafford, "Graph Services for Program Understanding Tools," *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, pp. 238-251, May 1992.

[Bu'84] Buhr, R., "Introduction to Pictorial Descriptions of System Architectures," *System Design with Ada*, pp. 38-56, Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[Ca'80] Carpano, M., "Automatic Display of Hierarchized Graphs for Computer-Aided Decision Analysis," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-10, No. 11, pp. 705-715, November 1980.

[Ch'79] Chiba, T., I. Nishioka, and I. Shirakawa, "An Algorithm of Maximal Planarization of Graphs," *Proceedings of ISCAS*, pp. 649-652, 1979.

[CC'90] Chikofsky, E. and J. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, pp. 13-17, January 1990.

[CM'91] "Software Size Measurement with Application to Source Statement Counting," Software Metrics Definition Working Group, Carnegie Mellon UNiversity, August 1991.

[CS'90] Choi, S. and W. Scacchi, "Extracting and Restructuring the Design of Large Systems," *IEEE Software*, pp. 66-71, January 1990.

[Da'89] Davidson, R. and D. Harel, "Drawing graphs nicely using simulated annealing," Report CS89-13, Department of Applied Science, The Weizmann Institute of Science, Rehovot, Israel, 1989.

[Ea'84] Eades, P., "A Heuristic for Graph Drawing," *Congressus Numerantium*, Vol. 42. pp. 149-160, 1984.

[Ea'86] Eades, P., B. McKay, and N. Wormald, "On an Edge Crossing Problem," *Proceedings of the 9th Australian Computer Science Conference*, pp. 327-334, 1986.

[Ea'90] Eades, P. and K. Sugiyama, "How to draw a directed graph," *Journal of Information Processing*, Vol. 13, No. 4, 1990.

[Fe'8x] Feynman, R., "Quantum-Mechanical Computers," Foundations of Physics, Vol. 16, No. 6, 1986.

[Fr'91] Fruchterman, T. and E. Reingold, "Graph Drawing by Force-directed Placement", *Software Practice and Experience*, Vol. 21, No. 11, pp. 1129-1164, November 1991.

[Ga'92] Gansner, E., E. Koutsofios, S. North, K. Vo, "Graph Visualization in Software Analysis," *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, pp. 226-237, May 1992.

[Ga'93] Gansner, E., E. Koutsofios, S. North, K. Vo, "A Technique for Drawing Directed Graphs," *IEEE Transactions on Software Engineering*, Vol. 19, No. 3, pp. 214-230, March 1993.

[Go'90] Goldberg, D. E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1990.

[Gr'85] Grefenstette, J. J. (Ed.), *Proc. Intl. Conf. Genetic Algorithms and their Applications*, Pittsburgh, PA 1985.

[Ha'77] *Elements of Software Science*, Elsevier North Holland, Inc., 1977.

[Ha'88] Harel, D., "On Visual Formalisms," *Communications of the ACM*, Vol. 31, No. 5, pp. 514-530, May, 1988.

[Ha'83] Harrington, S., *Computer Graphics - A Programming Approach*, pp. 88-104, pp. 211-226, McGraw-Hill, 1983.

[He'84] Henry, S., G. Kafura, "Software structure metrics based on information flow," *IEEE Computer*, November 1984.

[He'89] Henry, S. and R. Goff, "Complexity Measurement of a Graphical Programming Language," *Software-Practice and Experience*, Vol. 19, No. 11, pp. 1065-1088, November 1989.

[He'87] Helfman, J., "Bonsai: A Prototype Graph Browser and Graphical Interface Tool," AT&T Bell Laboratories, CMP Computing, 1987.

[Ho'74] Hopcroft, J. and R. Tarjan, "Efficient planarity testing," *Journal of the ACM*, Vol. 21, No. 4, pp. 549-568, October 1974.

[Ho'79] Hopcroft, J. and J. Ullmann, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesely, 1979.

[IS'82] Open Systems Interconnection Reference Model, International Organization for Standardization, ISO 7498, 1982.

[Le'80] Lehman, M., "Program, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE*, Vol. 68, No. 9., pp. 1060-1076, September 1980.

[My'86] Myers, B., "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy," *Conference Proceedings, CHI'86: Human Factors in Computing Systems*, pp. 59-66, 1986.

[Ja'88] Jayakumar, R. and K. Thulasiraman, "Planar Embedding; Linear-Time Algorithms for Vertex Placement and Edge Ordering," *IEEE Transactions of Circuits and Systems*, Vol. 35, No. 3, March 1988.

[Ka'89] Kamada, T. and S. Kawai, "An Algorithm for Drawing General Undirected Graphs," *Information Processing Letters*, Vol. 31, pp. 7-15, April 1989.

[Ki'83] Kirkpatrick, S., C. D. Gelatt Jr. and M. P. Vecchi, "Optimization By Simulated Annealing," *Science*, Vol. 220, pp. 671-680, 1983.

[Ko'91] Kosak, C., J. Marks, and S. Shieber, "A Parallel Genetic Algorithm for Network-Diagram Layout," 1991.

[Kr'83] Kramlich, D., G. Brown, R. Carling, C. Herot, "Program Visualization: Graphics Support for Software Development," *Proceedings of the 20th Design Automation Conference*, pp. 143-149, 1983.

[Le'80] Lehman, M., "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE*, Vol. 68, No. 9, pp. 199-215, September 1980.

[Li'89] Lieberman, H., "A Three-Dimensional Representation For Program Execution," *IEEE Proceedings Workshop on Visual Languages*, pp. 111-116, 1989.

[Me'53] Metropolis, N., A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, Vol. 21, 1087, 1953.

[Na'73] Nassi, I. and B. Shneiderman, "Flowchat Techniques for Structured Programming," *SIGPLAN Notes*, Vol. 8, No. 8, pp. 72-78, 1973.

[Om'90] Oman, P., "Maintenance Tools," *IEEE Software*, pp. 59-65, May 1990.

[Os'90] Osborne, W. and E. Chikofsky, "Fitting Pieces to the Maintenance Puzzle," *IEEE Software*, pp. 11-12, January 1990.

[Pa'72] Parnas, D., "On criteria for the decomposition of modules," *Communications of the ACM*, pp. 1053-1058, December 1972.

[Pa'90] Newberry-Paulisch, F., W. Tichy, "EDGE: An Extendible Graph Editor," *Software Practice and Experience*, Vol. 20, pp. 63-88, 1990.

[Pe'93] Peterson, I. "Reviving Software Dinosaurs," Science News, Vol. 144, No. 6, pp. 88-89, August 1993.

[Pr'75] Pratt, T., Programming Languages: Design and Implementation, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[Re'85] Reiss, S., "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 276-285, March 1985.

[Re'87] Reiss, S., "Working in the Garden Environment for Conceptual Programming," *IEEE Software*, pp. 16-27, November 1987.

[Re'89] Reiss, S., S. Meyers, and Carolyn Duby, "Using GELO to Visualize Software Systems," *Proceedings of the Second Annual Symposium on User Interface Software and Technology*, pp. 149-157, November 1989.

[Ri'90] Rich, W. and L. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, pp. 82-89, January 1990.

[Ro'85] Rogers, D., *Procedural Elements of Computer Graphics*, McGraw-Hill, 1985.

[Ro'87] Rowe, L., M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan, "A Browser for Directed Graphs," *Software-Practice and Experience*, Vol. 17, No. 1, pp. 61-76, January 1987.

[Ro'91] Robertson, G., J. Mackinlay, S. Card, "Cone Trees: Animated 3D Visualizations of Hierarchical Information," *Proceedings of CHI'91 Human Factors in Computing Systems*, pp. 189-194, ACM 1991.

[Ru'89] Rutenbar, R. A., "Simulated annealing algorithms: an overview," *IEEE Circuits and Devices*, pp. 19-26, January 1989.

[Sa78] Saltzer, J. H., "Naming and binding of objects," Operating Systems: An Advanced Course, Springer-Verlag, pp. 583-591, 1978.

[Sc'93] Schwartz, M. and D. Wood, "Discovering Shared Interests Using Graph Analysis," *Communications of the ACM*, Vol. 36, No. 8, pp. 78-89, August 1993.

[Sh'88] Shepperd, M., "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, pp. 30-36, March 1988.

[Sh'91] Shahookar, K. and P. Mazumdar, "VLSI cell placement techniques," *ACM Computing Surveys* Vol. 23, No. 2, pp. 143-220, June 1991.

[Sm'87] Smart, J., "The Livermore Security Console System," *Proceedings of the 1987 Carnahan Conference on Security Technology*, April 1987.

[Sm'92] Smart, J. and V. Vemuri, "Aesthetic Graph Layout or Program Understanding," *Proceedings of 3rd Reverse Engineering Forum*, September 1992.

[SV'92] Smart, J. and V. Vemuri, "A-Vu: A Visualization Tool for Complex Software Systems," *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, pp. 172-182, May 1992.

[Su'81] Sugiyama, K., S. Tagawa, and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-11, No. 2, pp. 109-125, February 1981.

[Ta'88] Tamassia, R. G. Di Battista, and C. Batini, "Automatic Graph Drawing and Readability of Diagrams," *IEEE Transactions on Systems, Man, and Cypernetics*, Vol. 18, No. 1, January/February 1988.

[Ta'89] Tamassia, R. and P. Eades, "Algorithms for drawing graphs: an annotated bibliography," Technical Report No. CS-89-09, Department of Computer Science, Brown University, February 1989.

[Tr'88] Tripp, L., "A Survey of Graphical Notations for Program Design-An Update," *ACM SIGSOFT Software Engineering Notes*, Vol. 13, No. 4, pp. 39-44, 1988.

[Tu'63] Tutte, W., "How to Draw a Graph," *Proceedings of the London Mathematical Society*, Vol. 3, No. 13, pp. 743-768, 1963.

[Ve'78] Vemuri, V., *Modeling of Complex Systems*, Academic Press, 1978.

[Wa'74] Warfield, J., "Structuring Complex Systems," *Battelle Monographs*, No. 4, April 1974.

[Wa'77] Warfield, J., "Crossing Theory and Hierarchy Mapping," IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-7, No. 7, pp. 502-523, 1977.

[We'52] Weyl, H., *Symmetry*, Princeton University Press, 1952.

Appendix A: A-Vu Definition Language (ADL) Specification

The A-Vu system is capable of processing system descriptions specified in an intermediate language referred to here as the A-Vu Definition Language or ADL. ADL is based on the A-Vu model presented above. Its programming language-like syntax, however, makes it more convenient for specifying system configurations than the formal definition given above. Dependency diagrams produced from other tools can therefore by processed by A-Vu by means of an appropriate translator.

Listed below is a BNF-type syntax summary of ADL. Square brackets enclose optional items. Braces enclose a repeated item which may appear zero or more times. Bold-faced items indicate reserved words. An italicized item represents a value from a predefined semantic category.

```
configuration ::=
    configuration configuration_identifier is
        configuration_body
    end configuration;

configuration_body ::=
    {node_declaration}
    {edge_declaration}
    {layout_declaration}
    {binding_declaration}

node_declaration ::=
    node node_identifier is
        {name_declaration}
        {node_attribute_declaration}
        {node_layout_declaration}
    end node ;

node_declaration ::=
    node node_identifier
        [: edge_identifier {, edge_identifier} ] ;
```

```
name_declaration ::=
    name : quoted_string ;

node_attribute_declaration ::=
    attribute : node_attribute {, node_attribute} ;

node_layout_declaration ::=
    layout : layout_identifier {, layout_identifier} ;

edge_declaration ::=
    edge edge_identifier is
        from_declaration
        to_declaration
        {edge_attribute_declaration}
    end edge ;

edge_declaration ::=
    edge edge_identifier : (node_identifier, node_identifier) ;

from_declaration ::=
    from : node_identifier ;

to_declaration ::=
    to : node_identifier ;

edge_attribute_declaration ::=
    attribute : edge_attribute {, edge_attribute} ;

layout_declaration ::=
    layout layout_identifier is
        [name : quoted_string ; ]
        [current : boolean ;  ]
        [view : view_type ; ]
        [perspective : perspective_type ;  ]
        [reflection : boolean ;  ]
        [rotation : boolean ;  ]
        [reference : (number, number, number) ; ]
        [space : (number, number, number) ; ]
        {node node_identifier :  (number, number, number) ; }
    end layout ;
```

binding_declaration ::=
  binding : (*node_identifier, layout_identifier*) ;

A configuration consists of a *configuration_identifier* and a configuration_body. The item *configuration_identifier* designates a unique name for the configuration using the identifier conventions found in typical programming languages. A configuration_body contains optional node_declaration, edge_declaration, layout_declaration, and binding_declaration sections.

The configuration_body section of a configuration may contain zero or more node declarations. Nodes may be declared using one of two different methods. The first method allows nodes to be completely specified as per the A-Vu model. This node_declaration form consists of a node identifier *node_identifier*, an optional name_declaration section, an optional node_attribute_declaration section, and an optional node_layout_declaration section. The node identifier must be unique across all configurations as it may frequently be referenced from another visualization space. The second method is provided for notational convenience in specifying directed graph structures without attribute information. This form consists simply of a node identifier followed by the node's adjacency list.

The optional name_declaration field in the full node form allows the node to be labeled with a specific text string such as a module or procedure name. The node_attribute_declaration is used to specify any attributes that are to be association with the node. Legal values for node attributes are from the following set: {*universal, procedural, functional, parallel, aggregate, standard, generic, instantiation, specification, implementation, foreign, composite*}. The meaning of these attributes are discussed in Section 2.6. Additional attributes may be added as the A-Vu model is further developed. The node_layout_declaration section is provided as a short-cut method for defining node-layout bindings. Alternatively, bindings may be defined explicitly in the binding_declaration section of the configuration_body.

Similar to nodes, edges can also be defined using one of two methods. The first form allows attributes to be associated with the edge while the second form provides another short-cut method when no edge attributes are involved. The direction of the edge is determined by the from_declaration and to_declaration sections in the full form or by the order in which the nodes are specified in short-cut form. The edge_attribute_declaration is used to specify any attributes that are to be associated with the edge. Legal values for edge attributes are from the following set: {*universal, implied, restricted, inherited, induced, visible*}. The meaning of these attributes are discussed in Section 2.8. Additional edge attributes may be added as the A-Vu model is further developed.

The layout_declaration portion of a configuration_body is used to defined which nodes are embedded in a visualization space, where they are located in that space, and how the space is viewed. The **name** field provides an optional text label for the layout for display purposes. For convenience, selected portions of the configuration state information ψ may be directly specified for each layout. The **current** field indicates whether or not the layout is selected as the current layout Λ. The **view, reference, perspective, reflection,** and **rotation** fields are all used to convey layout view parameters as described in Section 4.2.

The **space** field of a layout_declaration is used to declare the dimensions of the layout's visualization space. The *number* values specified may be either positive integers or positive reals. Integer values indicate a discrete space; real values indicate a continuous space; mixed values of integers and reals indicate a hybrid space. An unspecified space field indicates that the dimensions of the visualization space are undefined and that no node placement has been performed. The **node** field of a layout_declaration is used to specify which nodes are embedded in the layout and at what position. The *number* value types must match with each other and those declared in the space field.

The following is the ADL language definition for the system given in Table 1.1:

```
configuration TABLE_1.1 is
    node COMPUTE       :   MATRIX, SCALAR,VECTOR;
    node INPUT         :   IO;
    node IO;
    node MAIN          :   COMPUTE, INPUT, OUTPUT;
    node MATRIX        :   SCALAR, VECTOR;
    node OUTPUT        :   IO;
    node SCALAR;
    node VECTOR        :   SCALAR;
end configuration;
```

The configuration shown in Figure 2.3 is specified in ADL as follows:

```
configuration FIGURE_2.3 is

    node 1 is
        name   : "COMPUTE";
        attribute : procedural, visible;
    end node;

    node 2 is
        name : "INPUT";
        attribute : procedural, standard, visible;
    end node;

    node 3 is
        name : "IO";
        attribute : aggregate, foreign;
    end node;
    node 4 is
        name : "MAIN";
        attribute : procedural, visible;
    end node;

    node 5 is
        name : "MATRIX";
        attribute : aggregate. instantiation, visible;
    end node;
    node 6 is
        name : "OUTPUT";
        attribute : procedural. standard, visible;
    end node;

    node 7 is
        name : "SCALAR";
        attribute : aggregate. instantiation, visible;
    end node;

    node 8 is
        name : "VECTOR";
        attribute : aggregate. instantiation, visible;
    end node;

    edge 1   : (1, 5);
    edge 2   : (1, 8);
    edge 3   : (1, 7);
    edge 4   : (2, 7);
    edge 5   : (4, 1);
    edge 6   : (4, 2);
    edge 7   : (4, 6);
    edge 8   : (5, 8);
    edge 9   : (5, 7);
    edge 10  : (6, 3);
    edge 11  : (8, 7);

    layout 1 is
        name          : "Figure 2.3";
        current       : TRUE;
        view          : PLANAR;
        perspective   : FRONT;
        reflection    : FALSE;
        rotation      : FALSE;
        space         : (8,5,1);
        reference     : (4,3,1);
        node 1        : (4,2,1);
        node 2        : (3,2,1);
        node 4        : (4,1,1);
        node 5        : (4,3,1);
        node 6        : (5,2,1);
        node 7        : (4,5,1);
        node 8        : (4,4,1);
    end layout;

end configuration;
```

Appendix B: A-Vu Sequence Language (ASL) Specification

File commands

open { *filename* }

save { *filename* }

close { *filename* }

load { *filename* }

export containment { *filename* }

export dependency { *filename* }

analyze { *filename* }

Edit commands

cut

copy

paste

create { *name* }

modify { *name* }

delete

Move commands

move left

move right

move down

move up

move forward

move backward

Select commands

select all

select inverse

select root

select leaves

select body

select cyclic

select weak

select strong

select relative

select absolute

select referenced

select in

select out

select in out

select name {*search-expression*}

select pattern {*file*}

select node *attribute*

attribute ∈	{*universal, procedural, functional, parallel, aggregate, standard, generic, instantiation, specification, implementation, foreign, composite* }

select edge *attribute*

attribute ∈	{*universal, implied, restricted, inherited, induced* }

Arrange commands

arrange default

arrange dependent

arrange hierarchical

arrange layered

arrange breadth

arrange depth

arrange uniform

arrange centered

arrange lateral

arrange adjust

Reduce commands

reduce selected

reduce weak

reduce strong

reduce names

reduce attributes

reduce implied
reduce restricted

Compact commands

compact rows
compact columns
compact planes
compact volume

View commands

refresh
view planar
view composite
view spatial
view next
view previous
view front
view side
view top
view reflect
view rotate

Optimize commands

schedule {*filename*}
weights {*filename*}
optimize

Option commands

option connect
option disconnect
option identify on
option identify off
option length *n*
option transitive on

```
option transitive off
option reverse on
option reverse off
option level on
option level off
option shift
option swap
option directed
option undirected
option odd
option even
option random
option displacement
option constrained
```

Appendix C: Example Sequence

The following ASL sequence was used to automatically generate Figure 8.16 from Figure 1.3:

```
! Console reduction sequence
load console.lis

!Build default configuration
arrange default

!Perform graph reductions
reduce connected
reduce named
reduce restricted
reduce implied
select none

!Build composite space for support tools and predefined elements
select pattern support.pat
reduce selected
modify Support
move backward
view previous

!Build composite space for incident elements
select pattern incidents.pat
reduce selected
modify Incidents
select none

!Build composite space for object elements
select pattern objects.pat
reduce selected
modify Objects
select none

!Build composite space for Argus elements
select pattern argus.pat
reduce selected
modify Argus
select none

!Build composite space for dictionary elements
find *dictionary
reduce selected
modify Dictionaries
cut
view next
paste
view previous

!Build composite space for menu elements
find menu*
find *menu
reduce selected
modify Menus
select none

!Build composite space for handler elements
find *handler
find handle*
reduce selected
modify Handlers
select none

!Build composite space for painter elements
find *painter
reduce selected
modify Painters
select none

!Build composite space for video/camera elements
find video*
find camera*
reduce selected
modify Video
select none

!Build composite root elements
select root
find startup
find cleanup
reduce selected
modify Console
select none

!Generate final layout
arrange dependent
view composite
compact rows
compact volume
view plane
schedule default.sch
weights cross_mirror.wht
optimize
refresh
```