

REPORT DOCUMENTATION

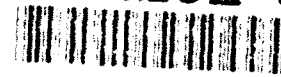
AD-A262 838

Form Approved

4-018

(2)

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing the collection of information, gathering and maintaining the data needed, and completing and reviewing the collection of information, including suggestions for reducing this burden, to Washington, Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Project, Washington, DC 20503.



Existing data sources:
Other aspect of this
series: 1215 Jefferson
20503

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE

February 1993

4. TITLE AND SUBTITLE

NetStat: A Probabilistic Network
Connectivity Analysis Tool

5. FUNDING NUMBERS

6. AUTHOR(S)

G. S. Marzot

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The MITRE Corporation
202 Burlington Road
Bedford, MA 01730

8. PERFORMING ORGANIZATION
REPORT NUMBER

M 93B0000034

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

The MITRE Corporation
202 Burlington Road
Bedford, MA 01730

10. SPONSORING / MONITORING
AGENCY REPORT NUMBER

M 93B0000034

11. SUPPLEMENTARY NOTES

DTIC
S E L E C T
APR 09 1993
S A D

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release;
distribution unlimited

12b. DISTRIBUTION CODE

A

13. ABSTRACT (Maximum 200 words)

This paper describes a computer program, NetStat, that is designed to analyze the connectivity of probabilistic networks. The program, which offers a full graphic user interface (GUI), allows the user to visually layout a network topology and specify the destruction probability, Pd, for each network element. A Monte-Carlo simulation is utilized to obtain estimates of various connectivity metrics. Some details of the implementation are discussed and an example application is illustrated. A user manual is provided. Source code is also provided, written in Symantec's Think Pascal for the Macintosh personal computer.

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION
OF REPORT

Unclassified

18. SECURITY CLASSIFICATION
OF THIS PAGE

Unclassified

19. SECURITY CLASSIFICATION
OF ABSTRACT

Unclassified

20. LIMITATION OF ABSTRACT

NetStat: A Probabilistic Network Connectivity Analysis Tool

M 93B0000034
February 1993

G. S. Marzot

93-07413



10/1/93

MITRE

Bedford, Massachusetts

93-07413

NetStat: A Probabilistic Network Connectivity Analysis Tool

M 93B0000034
February 1993

G. S. Marzot

Contract Sponsor N/A
Contract No. N/A
Project No. D50B
Dept. D058

Approved for public release;
distribution unlimited.

MITRE

Bedford, Massachusetts

Abstract:

This paper describes a computer program, NetStat, that is designed to analyze the connectivity of probabilistic networks. The program, which offers a full graphic user interface (GUI), allows the user to visually layout a network topology and specify the destruction probability, P_d , for each network element. A Monte-Carlo simulation is utilized to obtain estimates of various connectivity metrics. Some details of the implementation are discussed and an example application is illustrated. A user manual is provided. Source code is also provided, written in Symantec's Think Pascal for the Macintosh personal computer.

DTIC TAB

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Acknowledgments:

I would like to acknowledge the contribution of Dr. B. D. Metcalf under whose direction the initial work on this simulation was performed. I would also like to acknowledge Dr. V. C. Georgopoulos who also participated in the previous study and whose insights I found valuable. I would like to thank Dr. N. P. Shein who provided me with the "Graph Theory" perspective on the problem as well as many fine reference texts. I would also like to express my appreciation to Prof. C. H. Chang and the Tuft's University Electrical Engineering Department for their support and supervision of this work. A special thanks to Hilary Marzot.

TABLE OF CONTENTS

SECTION	PAGE
1.0 Introduction.....	1
2.0 Theory of Operation.....	2
3.0 Implementation.....	7
4.0 Example Application.....	13
5.0 Verification of Results.....	24
6.0 Summary.....	27
References and Bibliography.....	28
APPENDIX A - User's Manual	
APPENDIX B - Source Code	

LIST OF FIGURES

FIGURE	PAGE
1a An Unsegmented Network.....	3
1b A Segmented Network.....	3
2 Record Structure for Nodes and Links.....	9
3 Pascal Data Structure Representing Network of Nodes and Links.....	10
4 Flow Chart for Monte-Carlo Analysis of Network Topology.....	11
5a Star Topology.....	14
5b Ring Topology.....	14
6 Probability of Segmentation for a 12 Node Ring with 0 Crosslinks.....	15
7 Probability of Segmentation for a 12 Node Star and Hardened Hub,HPd=0.0...16	
8 Probability of Segmentation for a 12 Node Star and Hardened Hub,HPd=0.1...17	
9 Comparison of 12 Node Star to 12 Node Ring with 0 to 6 Crosslinks.....	19
10 Probability of Segmentation for a 12 Node Ring with 1 Crosslink.....	20
11 Probability of Segmentation for a 12 Node Ring with 2 Crosslink.....	21
12 Probability of Segmentation for a 12 Node Ring with 3 Crosslink.....	22
13 Probability of Segmentation for a 12 Node Ring with 6 Crosslink.....	23
14a Comparison of Closed Form Solution for 12 Node Simplex Ring.....	25
14b Comparison of Closed Form Solution for 12 Node Duplex Ring.....	25
15 Convergence of Monte-Carlo Simulation with Number of Trials.....	26

1.0 Introduction:

The study of networks has found application in many diverse areas. These range from air-traffic routing, to power and commodity distribution, telecommunication and computer networks as well as epidemiology and circuit analysis. One quality which network designers/analysts are often concerned about is the degree to which the network elements or nodes are connected. In the case of supply networks or communication networks the system designer may want to provide reliable connection between two given nodes or a uniform level of reliable connection to all nodes. A network architect may also be faced with certain performance constraints such as delay and throughput which limit his ability to meet the other requirements. As is often the case, the cost and availability of components and technology further constrain the designer. In this demanding environment of tradeoffs it is desirable to be able to quantify the performance of a given network, including its connectivity, to determine when the design goals have been adequately met and to enable resources to be allocated in the most intelligent way.

Historically a great deal of work has been done on analyzing the reliability of networks under a deterministic threat environment; that is, assuming an attacker has complete fore-knowledge of the network topology[1][7][15]. The goal of this previous work often centered on minimizing the number of steps required to partially or completely segment a network, and a number of closed-form solutions have been derived[2][7]. This approach is reasonable given the assumption of a deterministic threat. The closed-form result is also practical in light of the fact that computer simulated solutions were not viable until only a relatively short time ago.

However, networks are often subject to random factors such as weather, component life time, and collateral damage, either due to benign (construction/car accident) or hostile (military attack on adjacent target) sources. In this environment it is natural for

network component reliability to be described in a probabilistic fashion, yielding probabilistic networks. Analysis of this type of network has been characterized as an "NP-hard problem" and closed form solutions become intractable for all but the most constrained problems[13].

It is the aim of the computer application described in this paper to provide the network designer/analyst with a tool to assess various aspects of connectivity for these probabilistic networks. It is also the goal of the tool to be easy to use and generally applicable to a wide array of real world problems. Therefore a number of different types of network components are provided for problem setup. And several connectivity metrics can be computed allowing the analyst to answer several types of questions including: What topology is more reliable? Where is the best location for an additional link to enhance connectivity? How often should a given resource be replicated in a distributed computing environment? And more.

2.0 Theory of Operation:

To understand the operation of the tool, let's first define some vocabulary. The terms described here are defined primarily in the context of the tool but are also generally used in other contexts (see texts on Graph Theory for rigorous definition of terms[2][3]). A network is a collection of any number of nodes and links where N nodes can be joined by M links. A link must always be terminated by a node at each end (does not have to be a different node for a self-loop although self-loops are not considered here).

A link is represented as an edge or line which joins two nodes. There are two types of links considered here: simplex and duplex. Simplex links are directed edges which allow connection between nodes only in one direction. Duplex links are undirected edges which allow bidirectional connection between nodes. No distinction is made between half-duplex and full-duplex here.

Nodes are represented as points either located at the end of one or more links or separately as unconnected elements. Three types of nodes are considered here. The term node can be used generally to describe all types or specifically referring to the "Node" element in the computer application. The other node types in the application are "Hubs" and "Bypass Switches". "Hubs" only differ from the generic "Node" in a conceptual way and are treated by the application identically to "Nodes". "Hubs" are generally considered the central connecting elements in star type networks. They are included here as a separate type of element so that type specific parameters can be set for all hubs in a given network. The

"Bypass Switch" is another specialized node. It differs from the other nodes in that it is not considered when computing segment length or when assessing segmentation.

A segment is defined as a subset of the original network, consisting of nodes and links, within which each non-"Bypass Switch" node has a round trip path to every other non-"Bypass Switch" node. A network is considered segmented if it is made up of more than one segment. In other words, a network is segmented if at least one operational node is unable to establish a round trip path to at least one other operational node. Figure 1a and figure 1b illustrate this definition.

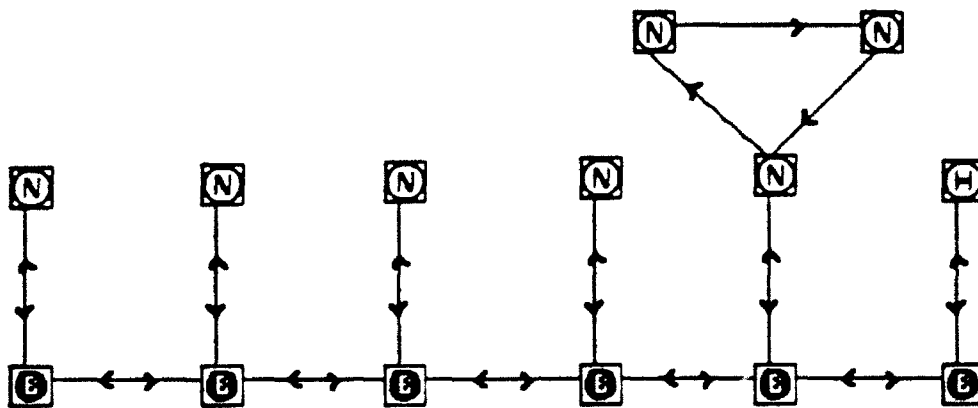


Figure 1a - An Unsegmented Network, the number of segments=1

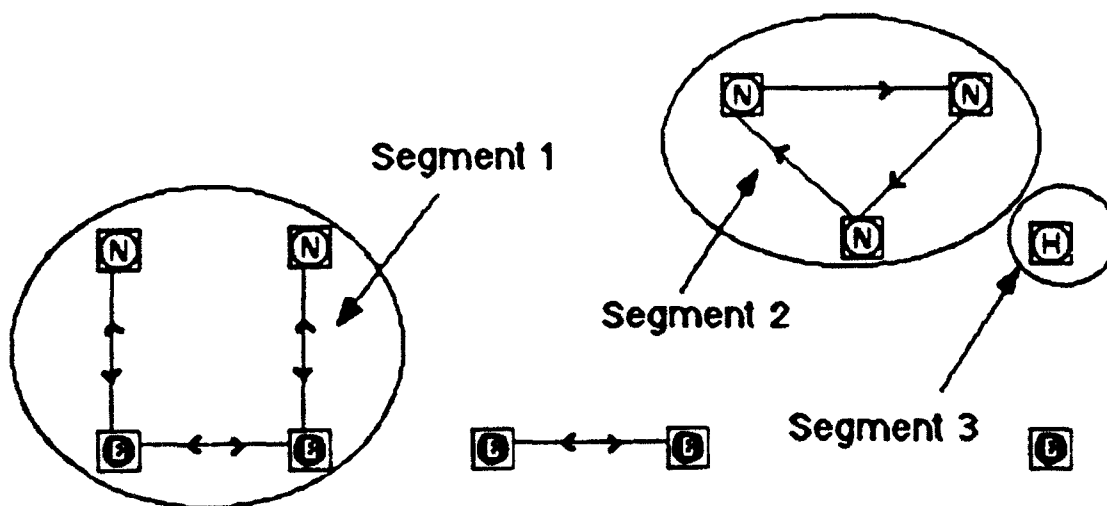


Figure 1b - A Segmented Network, the number of segments=3

Note: segments consisting of only Bypass Switches are not counted.

In a probabilistic network each network component is given a probability of failure or destruction, P_d . The element is considered to be either operative or inoperative based on a distribution described by P_d . This probability can also be thought of as the reliability of the component within an interval of time. Because each element in the network can assume a binary state it is apparent that in a network consisting of N nodes and M links that the total number of system states is given by:

$$N_{states} = \text{\#of possible system states} = 2^{N+M} \quad (1)$$

It should be noticed that this number can be quite large, even for small to medium sized networks.

Each system state, consisting of some combination of operative and inoperative network components, has a certain probability of occurrence given by the following:

let x_j = the event that a particular system state, j , occurs; where $1 \leq j \leq N_{states}$

p_i = the probability the i^{th} element is inoperative; $q_i = 1 - p_i$

$\epsilon_i = q_i$ if the i^{th} element in a particular system, j , is operative

$= p_i$ if the i^{th} element in a particular system, j , is inoperative

$P(x_j)$ = the probability x_j occurring

$$P(x_j) = \prod_{i=1}^{\text{\#of elements in system } j} \epsilon_i \quad (2)$$

Given that the network is in one of these system states, with some nodes and links inoperative, the resultant topology can be analyzed for qualities like segmentation and segment size.

Now it is useful to describe the connectivity metrics investigated by NetStat. The primary measure of connectivity looked at by this tool is the probability of segmentation, P_{seg} . This metric is very simply the probability that the network will consist of more than one segment. Stated more rigorously, P_{seg} is the probability that at least one operational node cannot establish bidirectional connection with at least one other operational node. This is seen as a good measure of how well the network is performing its function of connecting all nodes and it is desirable that P_{seg} be minimized in most cases. This metric

is an accepted reliability measure in the current literature [11]. As a note, a network consisting of one or zero nodes is not considered segmented by definition. Pseg can be calculated as follows:

$$\begin{aligned} \text{let } p_j &= 1 \text{ if the } j^{\text{th}} \text{ system state contains } > 1 \text{ segment} \\ &= 0 \text{ if the } j^{\text{th}} \text{ system state contains 1 or 0 segments} \\ P_{\text{seg}} &= \sum_{j=1}^{N_{\text{states}}} p_j P(x_j) \end{aligned} \quad (3)$$

Another well accepted measure of connectivity which is investigated by NetStat is the expected value of the number of connected pairs, Ecp [4]. For a given network, the number of connected pairs is defined by:

$$\begin{aligned} \text{let } K_j &= \# \text{ of segments in the } j^{\text{th}} \text{ system state} \\ N_k &= \# \text{ of nodes in the } k^{\text{th}} \text{ segment of the } j^{\text{th}} \text{ system state} \\ \# \text{ of connected pairs in a particular system, } j, &= N_{\text{cp}_j} = \sum_{k=1}^{K_j} \frac{N_k(N_k-1)}{2} \end{aligned} \quad (4)$$

It is the number of ways to choose 2 nodes from a set of N nodes summed for each segment in the network. The expected value of connected pairs is given by:

$$E_{\text{cp}} = \sum_{j=1}^{N_{\text{states}}} N_{\text{cp}_j} P(x_j) \quad (5)$$

Two other metrics are, the average or expected value of the segment length, Esl, and the expected value of the maximum segment length, Emsl. All possible segments and their probability of occurrence are used to compute the average segment size; whereas only the maximum segment length for each system state and its probability of occurrence is used to compute the expected maximum segment length. These measures give an idea of how well the network is held together even if segmentation does occur. They also give an indication of the most likely segment size which is useful when designing distributed systems on a probabilistic network. Their definition is stated mathematically as follows:

$$\begin{aligned} \text{let } L_k &= \text{the length of the } k^{\text{th}} \text{ segment} \\ M_j &= \text{the length of the maximum segment in the } j^{\text{th}} \text{ system state} \\ E_{\text{sl}} &= \sum_{j=1}^{N_{\text{states}}} \left(\frac{\sum_{k=1}^{K_j} L_k}{K_j} \right) P(x_j) \quad ; \quad E_{\text{msl}} = \sum_{j=1}^{N_{\text{states}}} M_j P(x_j) \end{aligned} \quad (6)(7)$$

For generality, when we look at Ecp it is normalized to the maximum number of connected pairs in the fully functional network. Similarly, Esl and Emsl are normalized to the total number of nodes in the whole system when all nodes are functional.

All of the metrics here can be computed exactly based on their definitions. One could examine every possible state of the network along with its probability of occurrence and keep track of segmentation, number of connected pairs, segment size etc. However, as stated before, the number of system states grows exponentially with the number of nodes and links and this method would be intractable for all but the most trivial networks.

Another approach, which works in some cases, takes advantage of system constraints and network symmetries to simplify the computation. Two of these analytical, closed-form expressions are shown here without proof for later comparison. The first is an expression for Pseg for an N node ring with duplex linking elements. The parameters are p, the node probability of failure, and q, the duplex link probability of failure:

$$P_{seg} = 1 - p^N + (N - 1)(1 - p)^N(1 - p)^N - N \sum_{j=0}^{N-1} p^j(1 - p)^{N-j}(1 - q)^{N-j-1} \quad (8)[8]$$

The second is an expression for Pseg for an N node simplex ring, where q is the simplex link probability of failure:

$$P_{seg} = 1 - p^N - N p^{N-1}(1 - p) - (1 - q)^N(1 - p)^N \quad (9)[12]$$

However, when the system under consideration has a fair sized number of nodes and a fairly complex topology, our only recourse is computer simulation. The technique used in NetStat is a Monte-Carlo simulation. The basic principal of the Monte-Carlo technique in this application is to randomly generate an ensemble of the system states so that estimates of the connectivity metrics can be computed over this ensemble. It is an essential requirement that the randomly generated system states in the Monte-Carlo simulation be statistically independent. This can be achieved if a random number, R_n , uniformly distributed between 0 and 1, is generated for each network component and compared to that components P_d in the manner of a Bernoulli experiment, as follows:

if $R_n \leq P_d$, the state of the i^{th} element is inoperative
 else if $R_n > P_d$, the state of the i^{th} element is operative

This also assumes that the uniform random numbers are statistically independent. This process is repeated for every element in the network thus generating one random state of the network.

The randomly generated system state can then be analyzed as before for segmentation, number of connected-pairs, segment size, etc. This process of generating a state and analyzing it is repeated as many times as is dictated by the simulation. The idea is that the statistics observed in this randomly generated ensemble will reflect the true ones and a good estimate of our metrics will be obtained if enough samples are used.

The benefit of the Monte-Carlo technique is that the number of states or trials that must be investigated can be significantly less than the total number of system states and is only dictated by the required accuracy. There is an expression which relates the number of Monte-Carlo trials required to the desired accuracy as follows. Without going into detail this expression relates the estimated result produced by the simulation to the exact result it reflects with the number of trials, N , as a parameter. It can be applied to general precision if the exact probability, p , is assumed to be the least significant digit in the desired accuracy:

$$\begin{aligned} p &= \text{the exact probability being estimated} \\ \hat{p} &= \text{the estimated probability} \\ p - 2\sqrt{p/N} &\leq \hat{p} \leq p + 2\sqrt{p/N} \end{aligned} \quad (10)[6]$$

This expression is valid with a 95% confidence level due to certain approximations in its derivation. An example of its application follows:

let $p = \text{the desired accuracy in } P_{\text{seg}} = 10^{-3}$

$N = \text{\#of Monte-Carlo trials} = 10^4$

$$\therefore 0.00036 \leq \hat{p} \leq 0.00163$$

From this we can expect to see fluctuation in the third decimal place and occasional fluctuation in the second decimal place when 10,000 Monte-Carlo trials are used.

3.0 Implementation:

The computer application, NetStat, was implemented on a Macintosh personal computer using Symantec's Think Pascal. The Macintosh platform was desirable due to its

inherent support for a graphical user interface as well as being an adequately powered computing engine. This support comes primarily through the, roughly, 700 ROM-based Pascal routines which are bundled with the Macintosh computer. These routines handle a wide array of low level tasks, including the creation of a windowing environment, pull-down menus, file I/O, basic drawing, etc. Although the description of the program which follows is focused primarily on the simulation related data structures and algorithms, it should be noted that more than 70% of the source code is devoted to creating the GUI and platform specific interaction. This is a typical percentage for applications with a full GUI.

The application was geared to the high end Macintosh II series computer (SE/30 also) which incorporates a separate math coprocessor to enhance the efficiency of floating-point operations. The source code may be recompiled to run on Macintoshes without a math coprocessor making minor adjustments. However this will result in a noticeable degradation in performance.

The choice of a Pascal-based development environment* was natural due to its compatibility with the operating system interface and the language's straightforward readability. Pascal also offers a number of language constructs which facilitate the implementation of this model. These include; abstract data structures such as "records" and "sets", the ability to link structures with pointers (e.g., linked list), and support for recursive procedure calling. In the description of the program implementation that follows it will be shown how these features were utilized to construct a model of the network and facilitate its analysis.

A typical user session on NetStat consists of three phases (see NetStat User's Manual for complete description of program environment and user interface). The three phases are; topology construction, parameter specification, and simulation run. During the topology construction phase, while the user visually lays out the network to be analyzed, an analogous data structure is constructed and held in the program's global data space. The data structure is basically a linked list consisting of two types of elements, node records and link records. Figure 2 shows the basic makeup of these two records (see NetStat Source Code, ConsVars Unit for exact record structure).

*Authors Note: The Think Pascal environment is an excellent full featured professional programming environment. It offers an integrated editing, debugging, linking and compiling shell with an intuitive and easy to use interface. I highly recommend it.

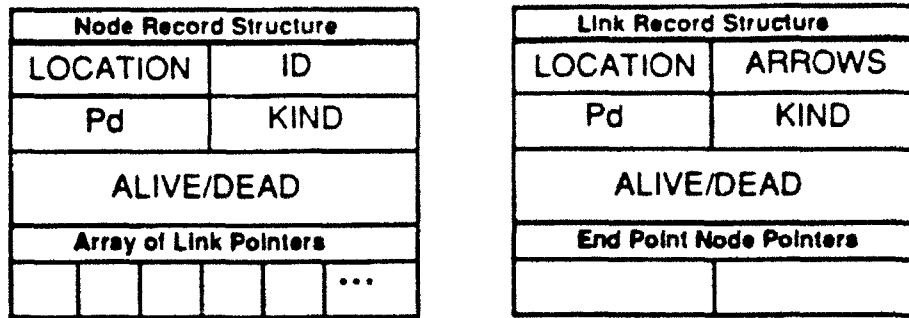


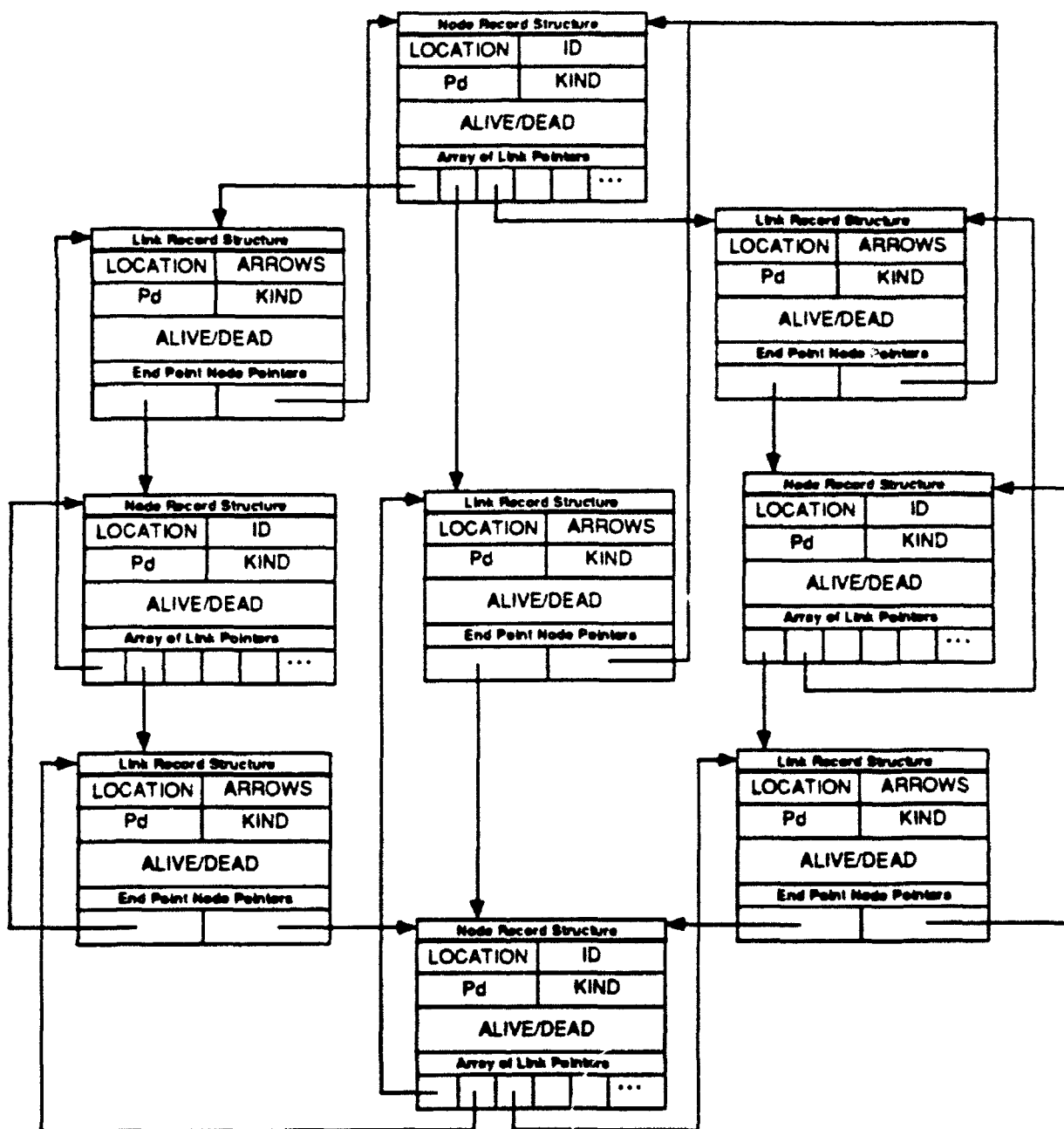
Figure 2 - Record Structure for Nodes and Links

The network element records contain graphical information such as the "location", which specifies, relative to the application drawing pad, where the element should be drawn. In the case of the link, additional information is stored showing where the "arrows" should be drawn. All elements have a "Pd" field which is its probability of failure and a "kind" field which specifies "Node", "Hub", or "Bypass" in the case of the node element and "Simplex" or "Duplex" in the case of the link. All elements also have a state field which specifies whether the element is operational or not. Most significantly, all element records have holders which can contain the address of, or pointers to, the other records in the network data structure. In the case of the nodes, an array of pointers to link records is used to keep track of which links emanate from a node. It should be noted that a pointer to a bidirectional link would be stored in the array of both its endpoint nodes whereas a simplex link pointer would only be stored in the node from which it emanates as indicated by its arrow. In the case of links, two fields are provided to keep track of the addresses for the two node records at each endpoint.

Figure 3 illustrates the resultant data structure when a 4 node ring with one diametric crosslink is constructed. All links are bidirectional.

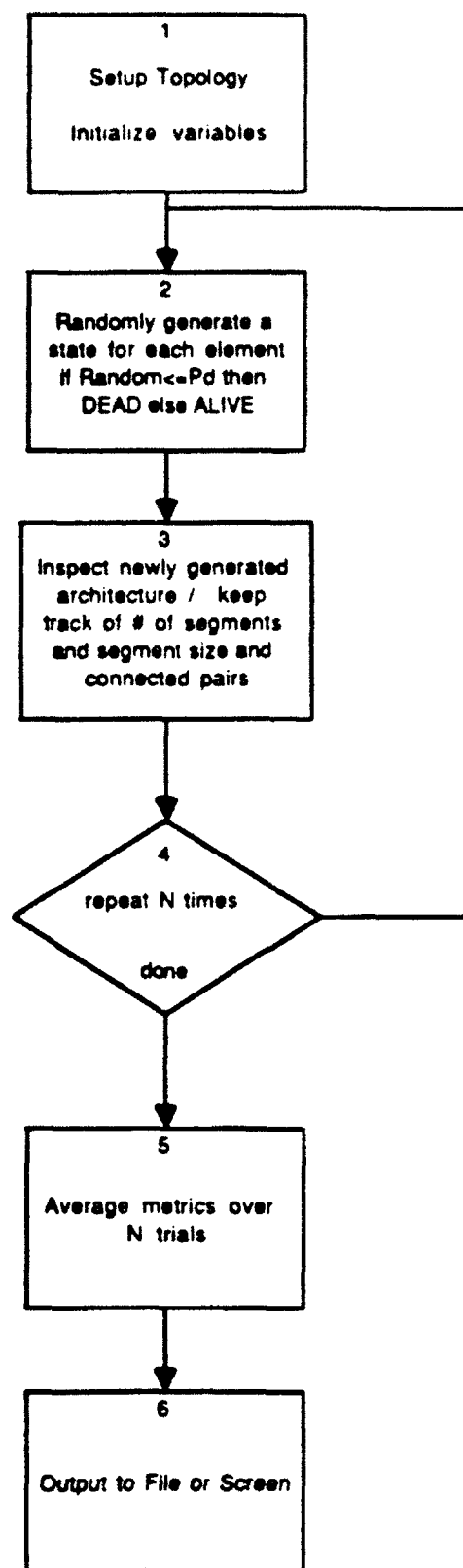
After the topology is constructed the user may specify the Pd's for the network components. This may be done uniformly by type; that is for instance, all "Hubs" have a $Pd = 0.1$. It may also be done individually by visually navigating the network (clicking on components) and setting each components Pd accordingly. Additionally, parameters such as the number of Monte-Carlo trials can be set.

When parameters such as these are set up, a simulation "run" can commence. A flow chart of the simulation operations is shown in figure 4. The state of each network component is determined for all the components in the network to produce a new topology or system state. This resultant topology is analyzed for the metrics described above and counters are maintained as to whether segmentation occurred, the number of segments



Pascal Data Structure Representing Network of Nodes and Links

Figure 3



Flow Chart for Monte-Carlo Analysis of Network Topology

Figure 4

produced and their sizes. This process is repeated N times where N is the number of Monte-Carlo trials. Then the counters are examined to determine Pseg, Ecp, Esl and Emsl.

The operations in block 3 of figure 4 are central to the working of the simulation and it is very important that this step be computationally efficient. Basically, this operation is implemented as a depth-first tree search which proceeds until all operational nodes are accounted for. The searching portion of code is called recursively while keeping track of "set" type variables which indicate which nodes are in the current segment and those that are accounted for in other segments (see NetStat Source Code, MultiEngine Unit, procedure inspectseg). The "set" language constructs included in Think Pascal are instrumental in these operations, providing a means to compute rapidly the union and intersection of various sets. The tree search must treat simplex links in a special way. The nodes encountered when traversing a simplex link are not added to the set of nodes in the current segment until a roundtrip path is established.

Another critical portion of the implementation is the nature of the random numbers generated when determining the state of each network component. These numbers must be statistically independent and a unique series of numbers must be generated each time a simulation is run. The random number generator used in this simulation is the native one found in the Macintosh ROM which has been well studied and determined to generate high quality output*. Steps were taken to re-seed the random number generator between runs.

Source code for NetStat is included at the end of this paper. The code is broken into three units; ConsVars, Main, and Multi-engine. ConsVars contains global constants and global variables as well as application specific data types and structures. The "Main" unit contains many routines to implement the user interface and construct the network data structure as well as the main-line of the program. The "MultiEngine" unit contains two versions of the code that analyzes a given topology, one which is optimized for Pseg and another which is more general in scope.

* Minimal Standard RNG - Park, Miller; Communications of the Association for Computing Machinery. Oct. 1988

4.0 Example Application:

To illustrate how NetStat can be used an example problem is posed. The example problem, which might be faced by a local area network designer, is the choice between two given topologies. Specifically, the tool will be used to evaluate the benefits of a star topology versus a ring topology for a twelve node network. For one of the given topologies the tool will also be used to determine the best way to enhance its reliability.

Some assumptions about the problem follow. The role that the network is designed to fulfill is the bidirectional communication between all nodes, and the network is considered failed if all nodes cannot communicate. The link medium is assumed to be bidirectional (duplex) such as ethernet coaxial cable. For the most part, equivalent resources are available in each case. That is, twelve nodes and twelve links are available to construct the network. In the case of the star however, an extra component is assumed for the central connecting node, or hub. (In the actual simulation a "bypass switch" is used because we do not want the central node counted for Pseg or segment length - see Users' Manual). In general we know very little about the actual destruction probabilities for the various components. We have made the assumption that all components of one type have the same destruction probability. Additionally in certain design environments the nodes are more likely to fail than the links. This assumption has been made for one of the comparisons described here.

Figure 5a and 5b show the two architectures as they were set up in the simulation environment. The probability of segmentation was chosen as the criterion of optimization because of its stringent nature and because of the previously stated goal of the network to provide communication to all nodes. As a reminder, our aim is to choose the topology which minimizes Pseg. Since we have very little information about the probabilities of failure, we must analyze the behavior of the network over a range of values for each component, thus generating a family of curves for each topology which paint a broad picture of their respective levels of connectivity. The main parameters to be varied are the probability of destruction for the nodes (NPd) and for the links (LPd). In the case of the star the central nodes destruction probability (HPd) may also be varied. Figure 6 and 7 show families of curves of Pseg versus LPd for different values of NPd, for the ring and the star architectures respectively. It turns out that the Pseg for the star is fairly sensitive to the value of HPd (see Figure 8). For the purpose of comparison we

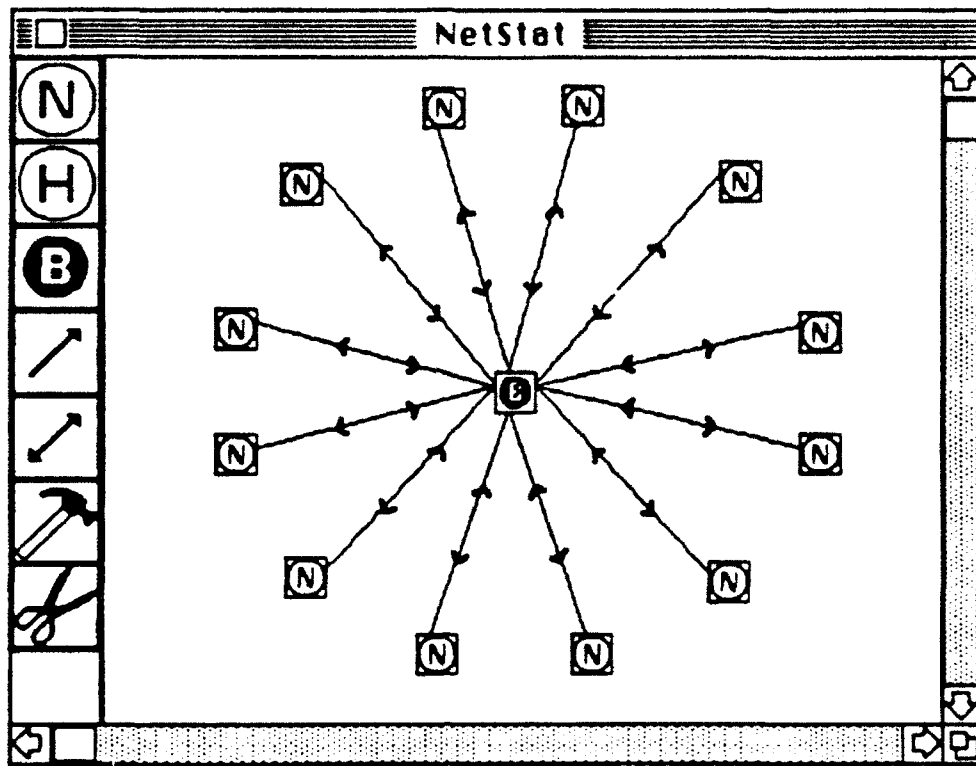


Figure 5a - Star Topology

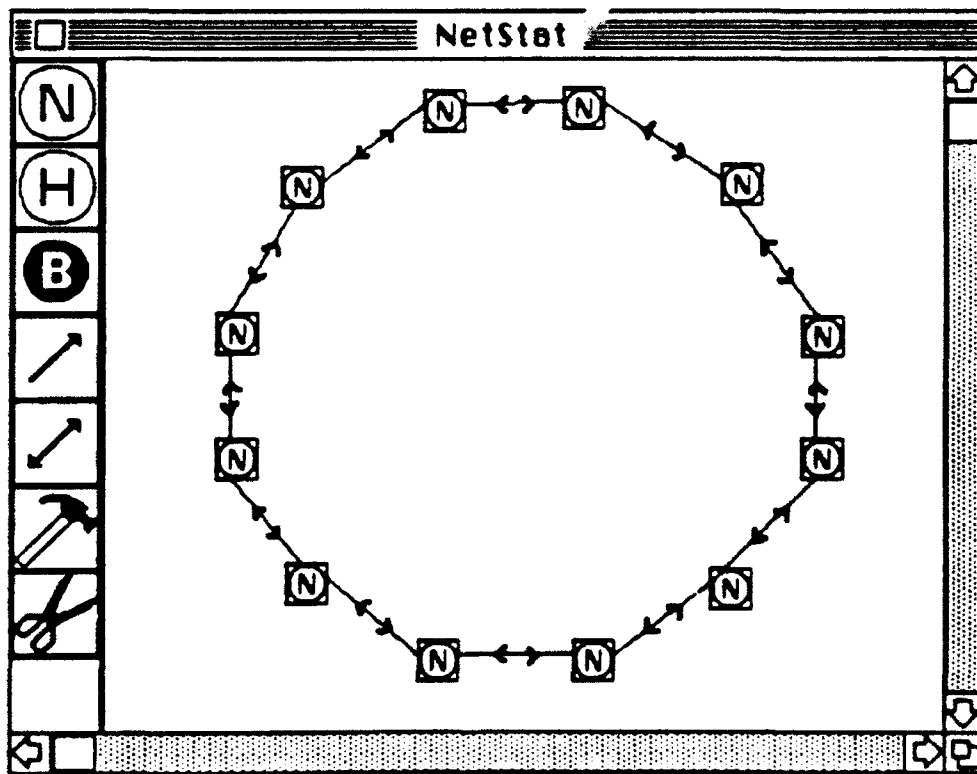


Figure 5b - Ring Topology

**Probability of Segmentation for
a 12 Node Ring with 0 Crosslinks**

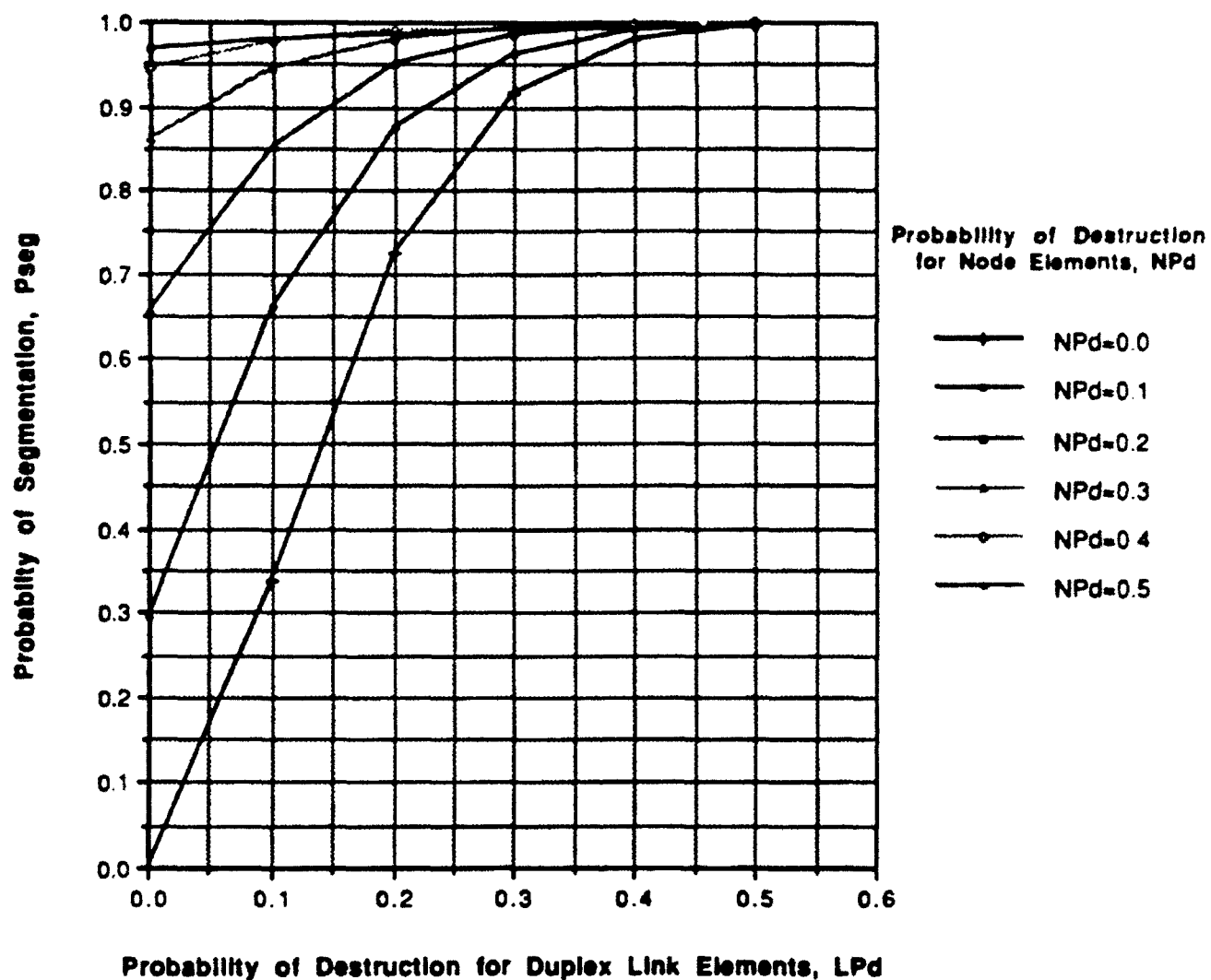


Figure 6

**Probability of Segmentation for
a 12 Node Star and Hardened Hub, $HP_d=0$**

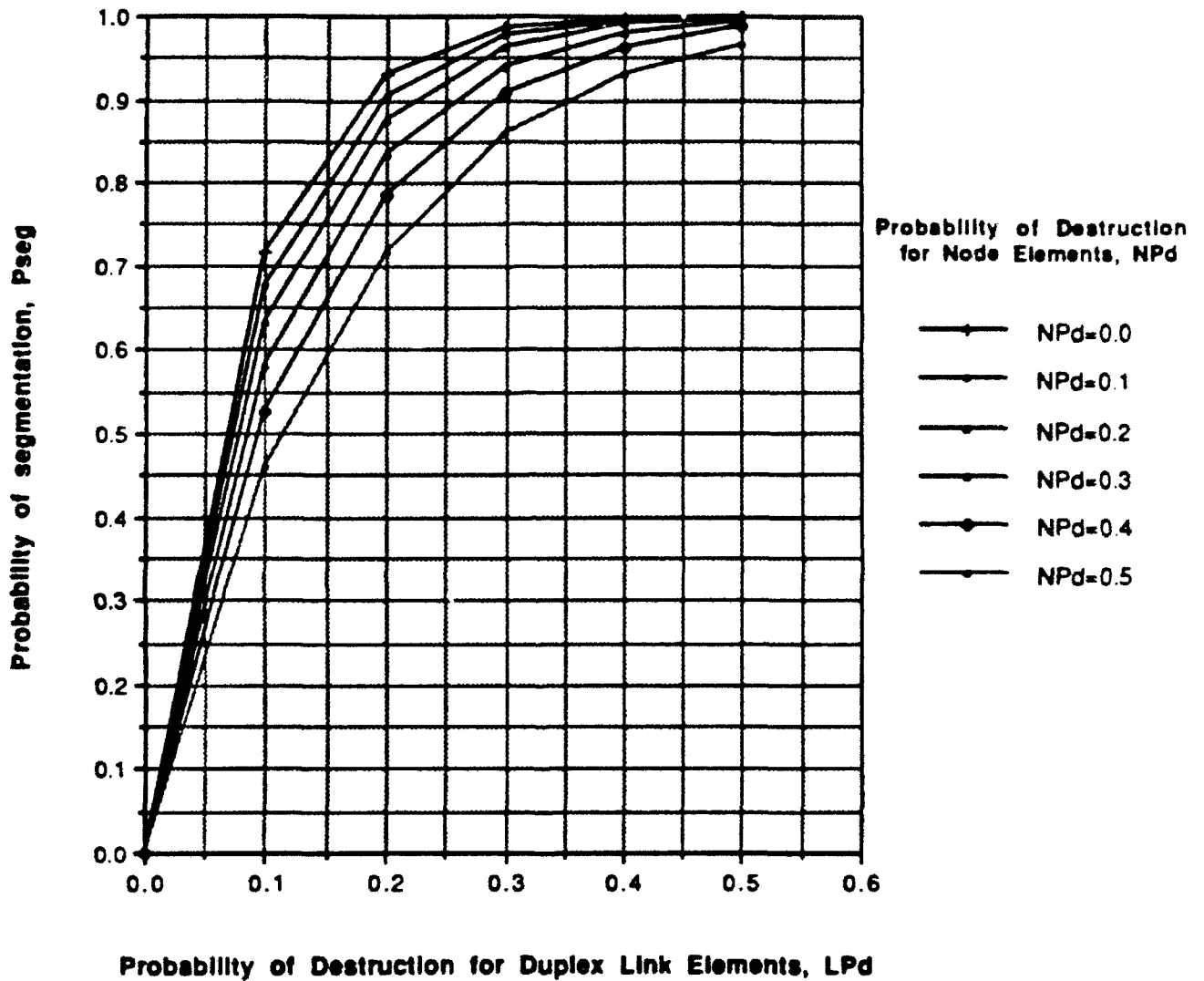


Figure 7

**Probability of Segmentation
for a 12 Node Star and $HP_d=0.1$**

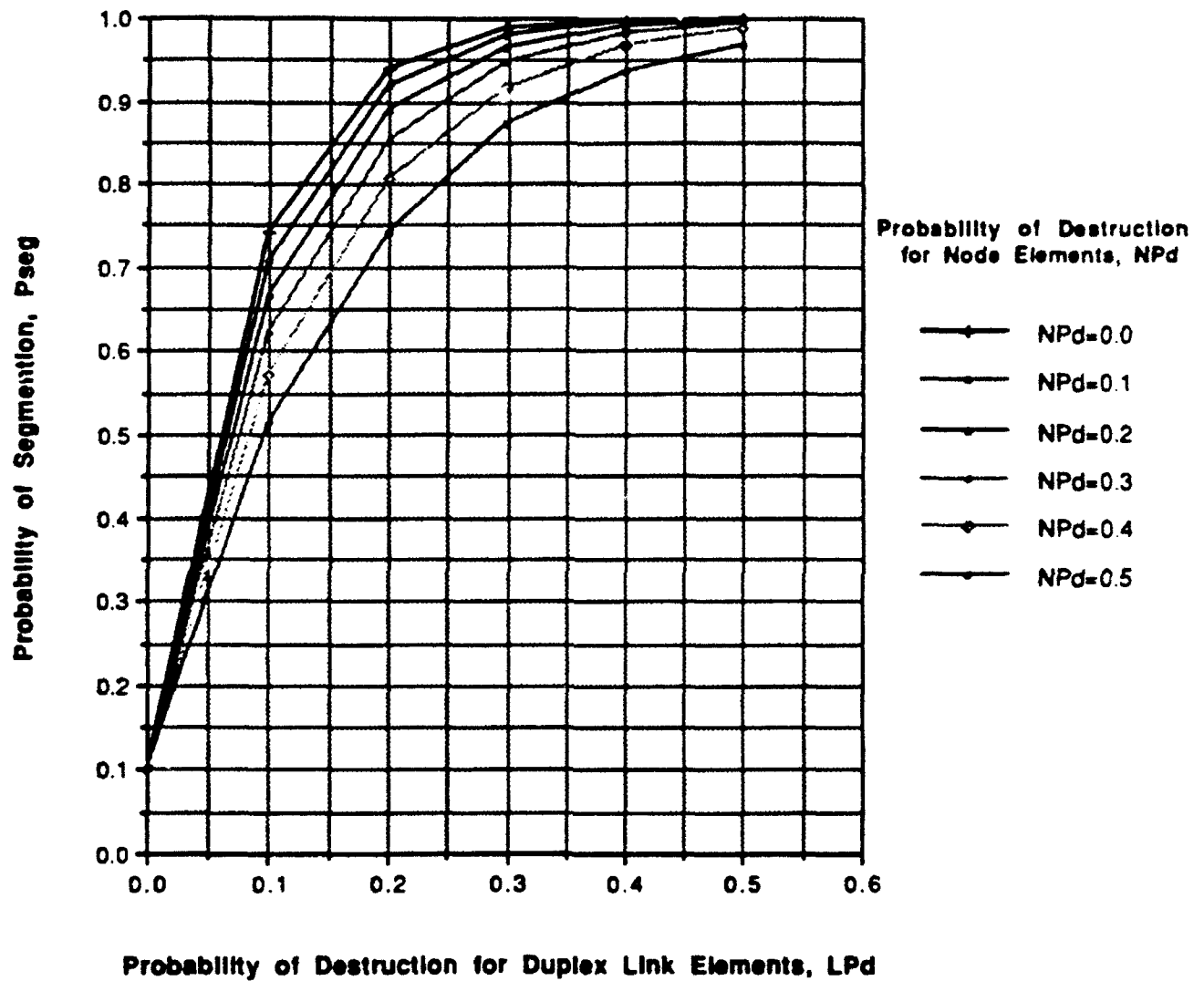


Figure 8

only examine two cases, $HPd=0$ and $HPd=0.1$, assuming a highly reliable hub can be used.

From the figures we can see that the stars reliability is more sensitive to LPd than to NPd , whereas the rings reliability is more equally sensitive to NPd and LPd . Because we know that nodes are more likely to fail than links, we would like to make a comparison with this constraint. We can (somewhat arbitrarily) let $LPd = 0.5 NPd$ and let NPd be varied. Figure 9 shows this for the star and ring. Also shown in this figure is the affect of adding crosslinks to the ring which will be discussed later. For all of the curves presented, failure probabilities beyond 0.5 were not investigated since the state of the network is less interesting when more than half of the components are removed (we assume that the network will almost assuredly be unusable in this state). Additionally, the value of $Pseg$ approaches 0 as NPd approaches 1, by definition. This is somewhat counter intuitive and is not constructive in the comparison.

In the last comparison only failure probabilities less than 0.1 were examined. This assumption of low levels of component failure can be reasonable in many cases, excepting possibly the military environment where high levels of damage can occur. The comparison in Figure 9 favors the ring over the star even with the generous assumption of $HPd = 0$.

Given that a ring has been chosen, one may be interested in ways to enhance its reliability. There are two ways to approach this. The first is to try to minimize the failure probabilities of the components and the other is to add resources, namely links, to provide redundant connection. Assuming that all has been done to minimize failure probabilities we must turn to the addition of links. This begs the question; where should this link be placed to minimize the probability of segmentation. In the case of the twelve node ring an additional link was added between all possible pairs of nodes and it was determined that the diametric crosslink provides the most additional reliability. Data was generated for the ring with increasing number of diametric crosslinks. The results are shown in figure 9 through 13. From these figures we can see the significant benefit to reliability realized by adding redundant links.

It is also important to note that this has a greater impact on reliability than increasing the reliability of the hub in the star network. This is a significant result because it is often cheaper to add more links than to increase the reliability of certain components.

**Comparison of 12 Node Star with Hardened Hub
to 12 Node Ring with 0 to 6 Crosslinks**

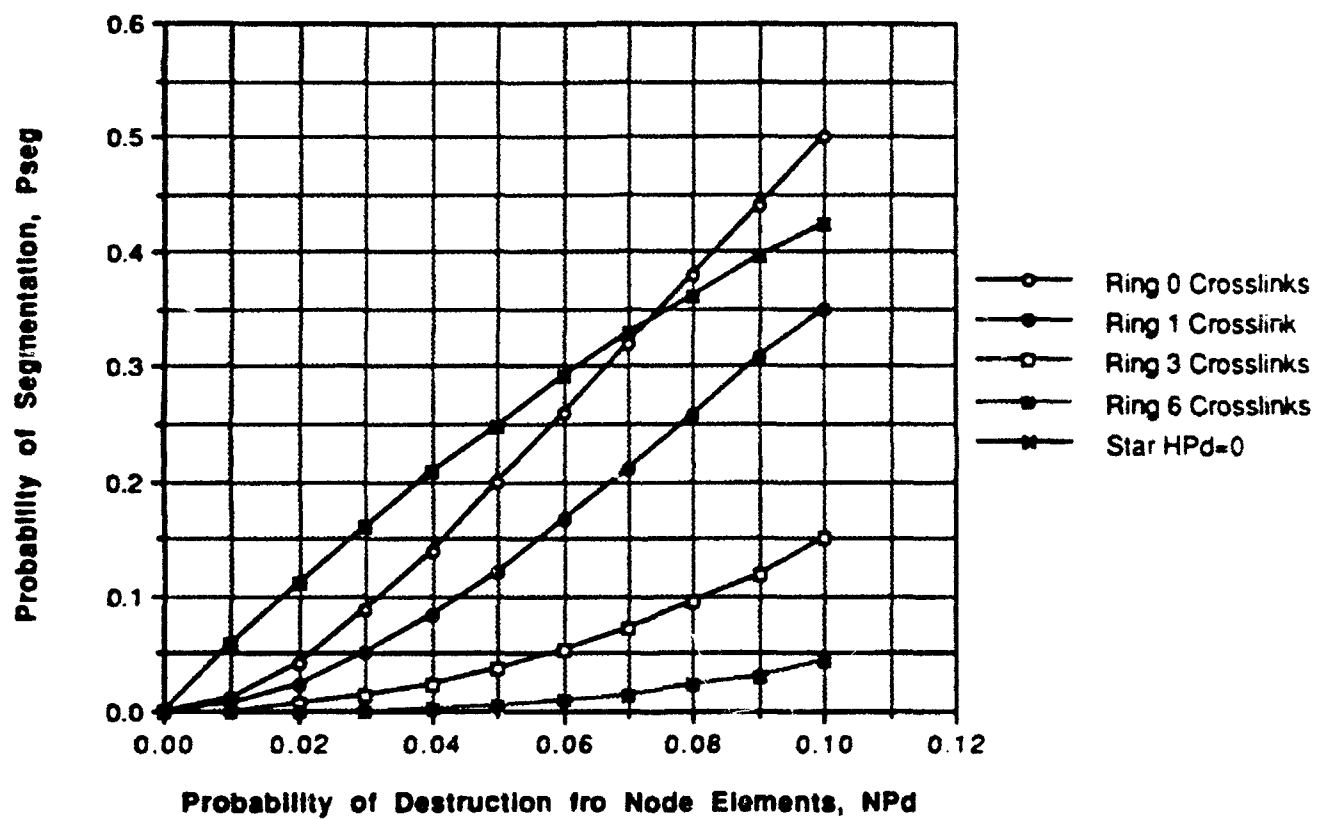


Figure 9

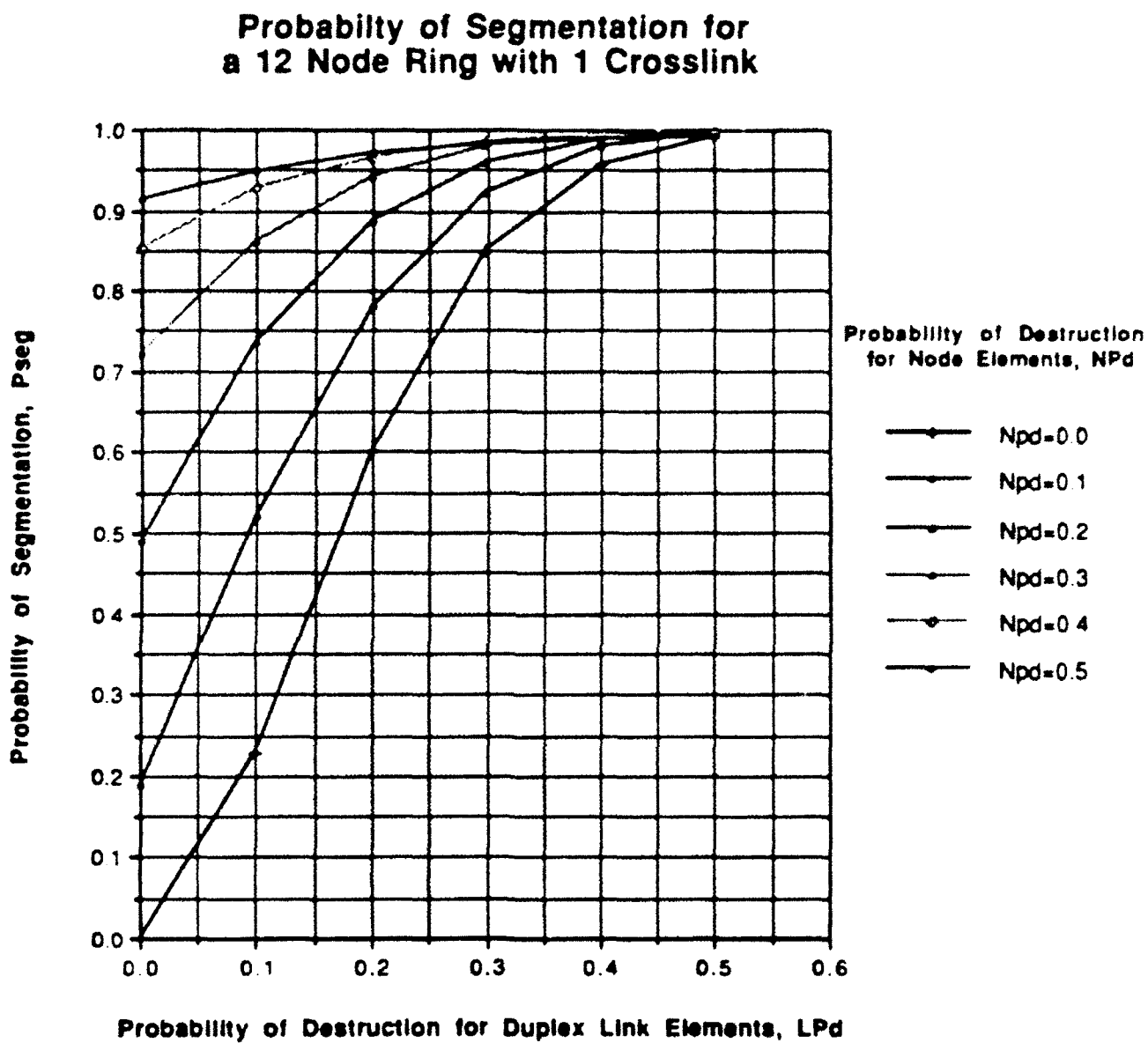


Figure 10

**Probability of Segmentation for
a 12 Node Ring with 2 Crosslinks**

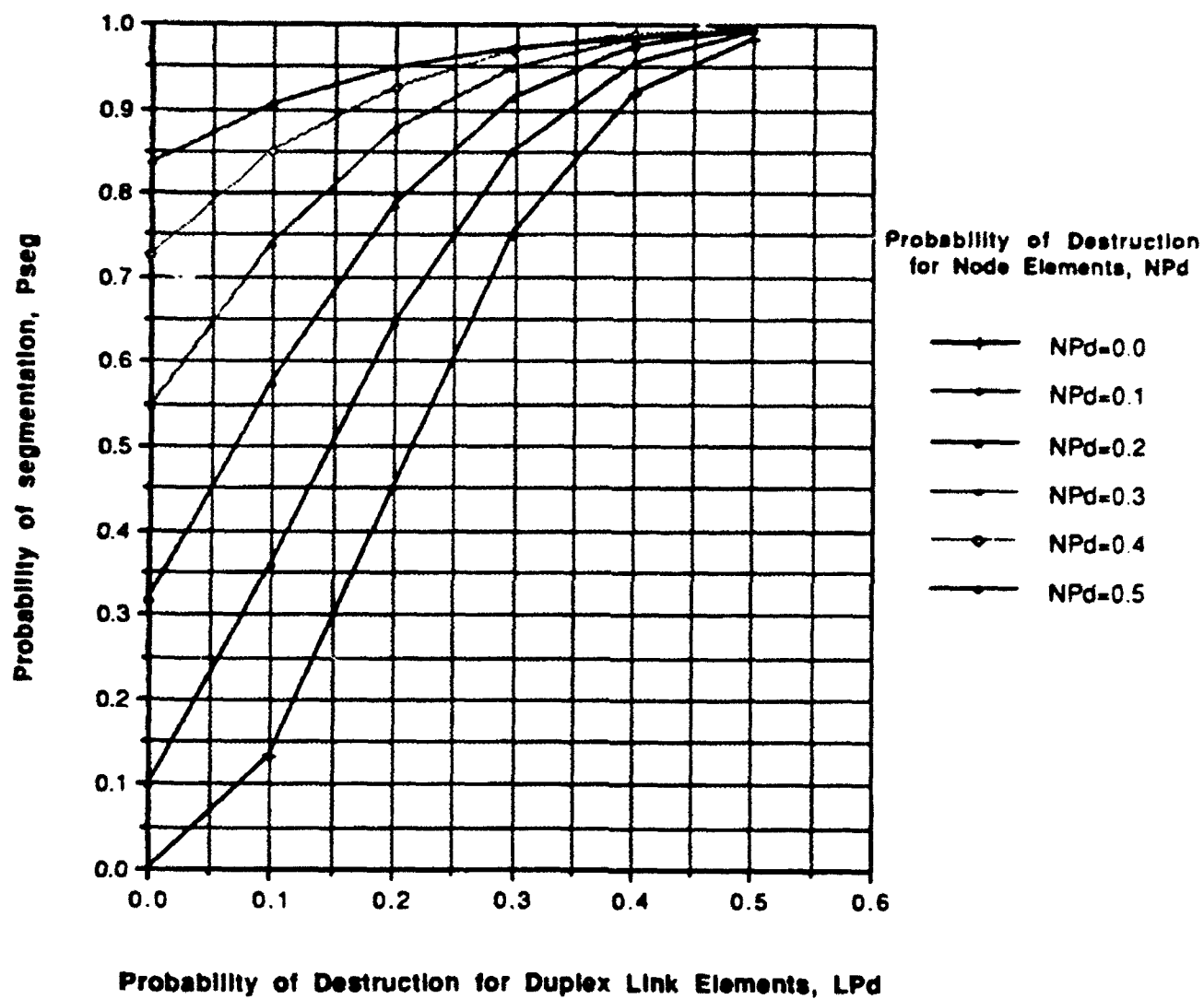


Figure 11

**Probability of Segmentation for
a 12 Node Ring with 3 Crosslinks**

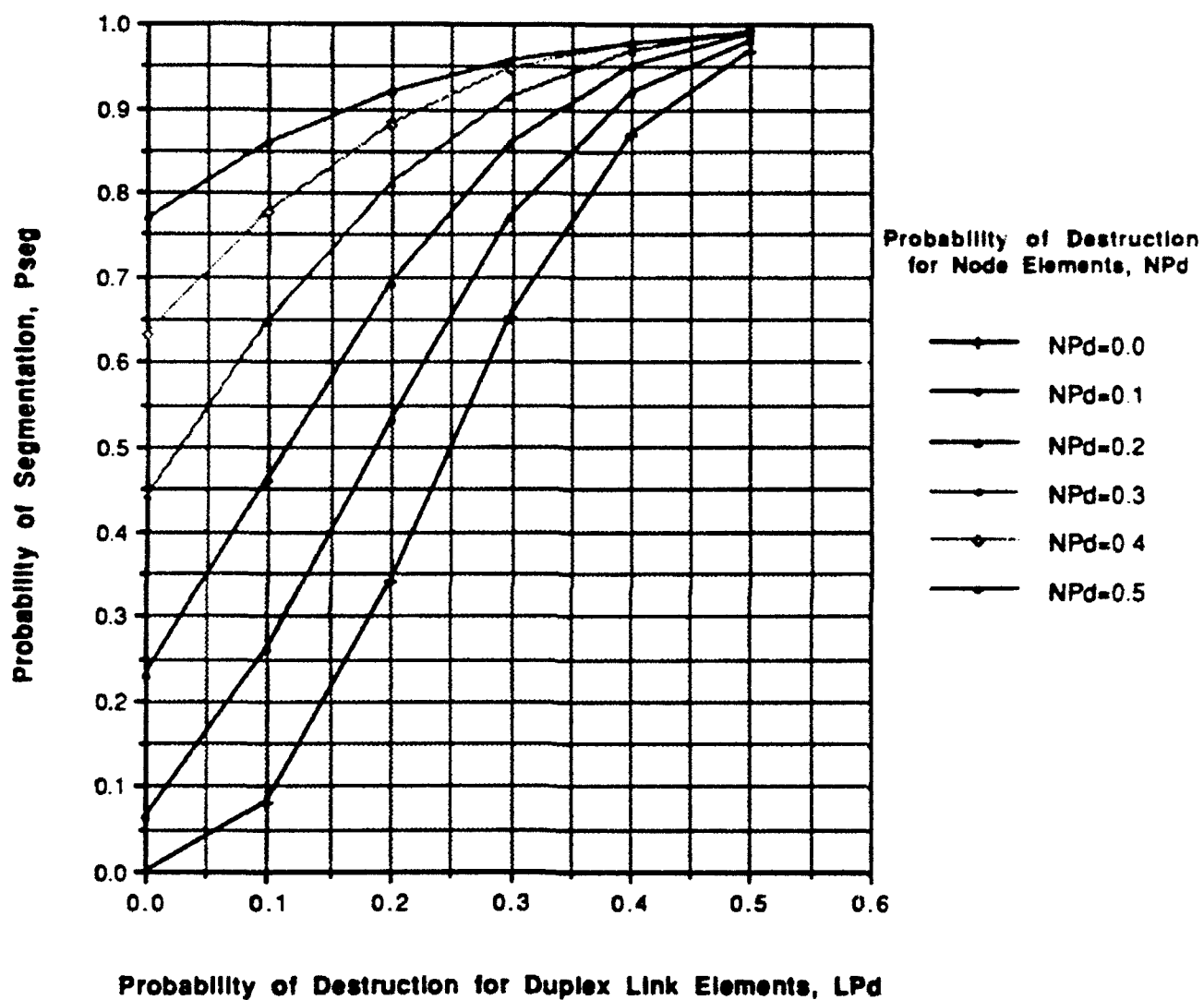


Figure 12

**Probability of Segmentation for
a 12 Node Ring with 6 Crosslinks**

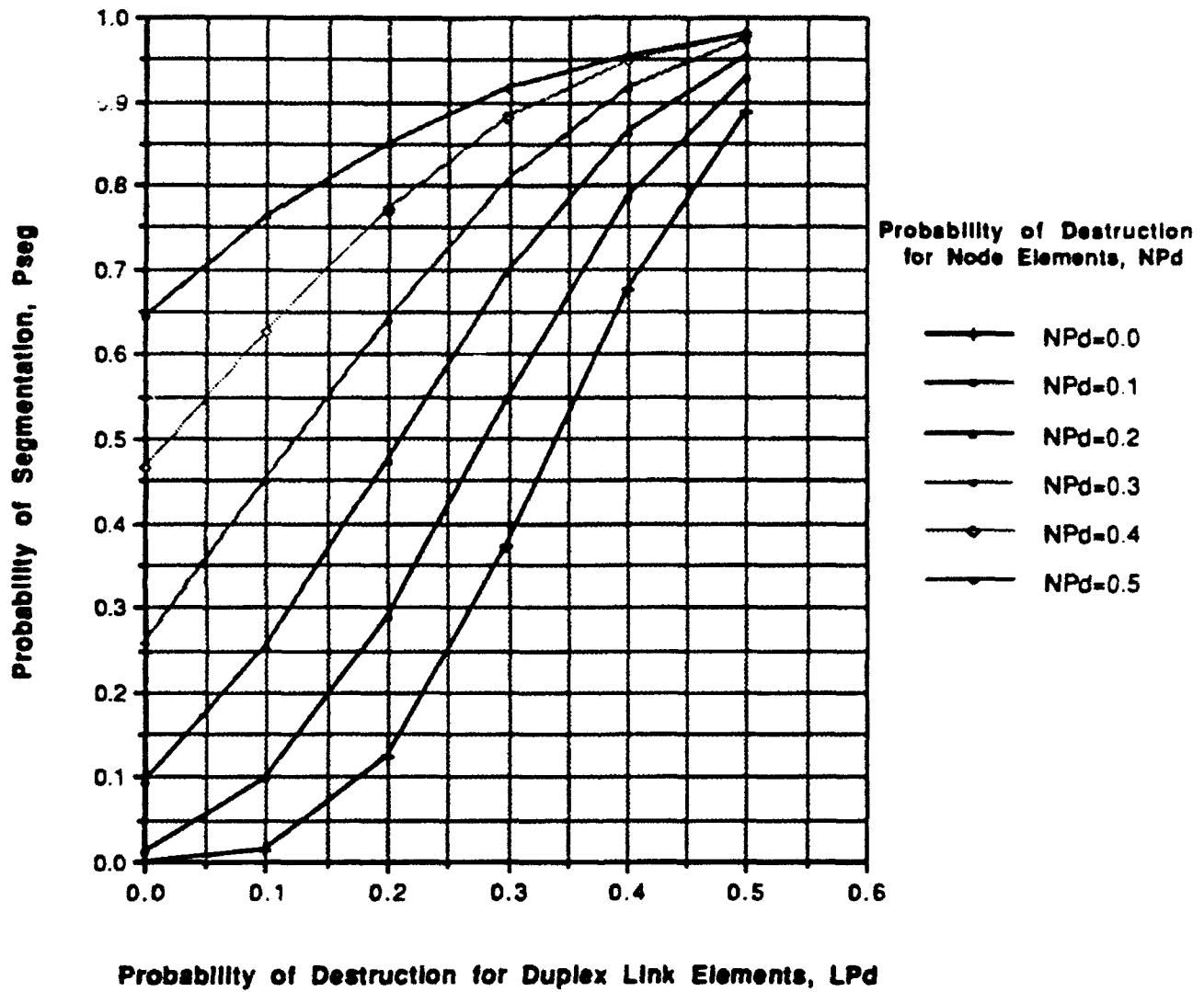


Figure 13

5.0 Verification of Results:

One of the inherent problems with a simulated result obtained in the absence of a closed form solution, is the difficulty in verifying the output. Even when the principal of operation is sound, algorithmic errors can corrupt the results. Therefore it is necessary to take extra steps to validate a simulation. In the case of the simulation described here, several techniques were used to gain confidence and corroborate the results.

One technique that is useful is to look at limiting cases. Specifically, we can look at the probability of segmentation as the P_d of the components approaches zero or one. P_{seg} should approach zero as component failure probabilities approach zero (provided the network was unsegmented to begin with). P_{seg} should also approach zero as the P_d of all the components approaches one, as stated before. This method can be applied to all components or selected ones. Another verification scenario using this technique is a star network where all the satellite nodes and links are given a P_d of 0 and the HP_d is allowed to vary. As HP_d approaches 0, P_{seg} approaches 0. In fact, P_{seg} should be exactly equal to HP_d at all values.

Another technique is to examine simple cases with a few number of nodes and links. These can be worked out by hand and compared to the simulated result. Taking this a step further, closed form solutions that have been derived for certain types of networks (see Theory of Operation) can be compared to the simulated result. To add further confidence the comparison should be made for varying number of nodes and probabilities of destruction. This type of comparison was performed for the two closed form solutions provided earlier (equ. (8) and (9)) with positive outcome. Figure 14a and 14b show the results of the comparison.

Another aspect of the simulation that can be verified is the convergence of the Monte-Carlo results with increasing number of trials. This technique was applied to a sample network by examining the spread of the computed result for P_{seg} over a set of 10 runs with the number of Monte-Carlo trials varying from 10 to 100000. The results of this test are shown in figure 15. The performance is in agreement with that predicted by the expression for Monte-Carlo confidence intervals (equ. (10)).

Comparison of Closed Form Solution to Simulated Results for 12 Node Duplex Ring

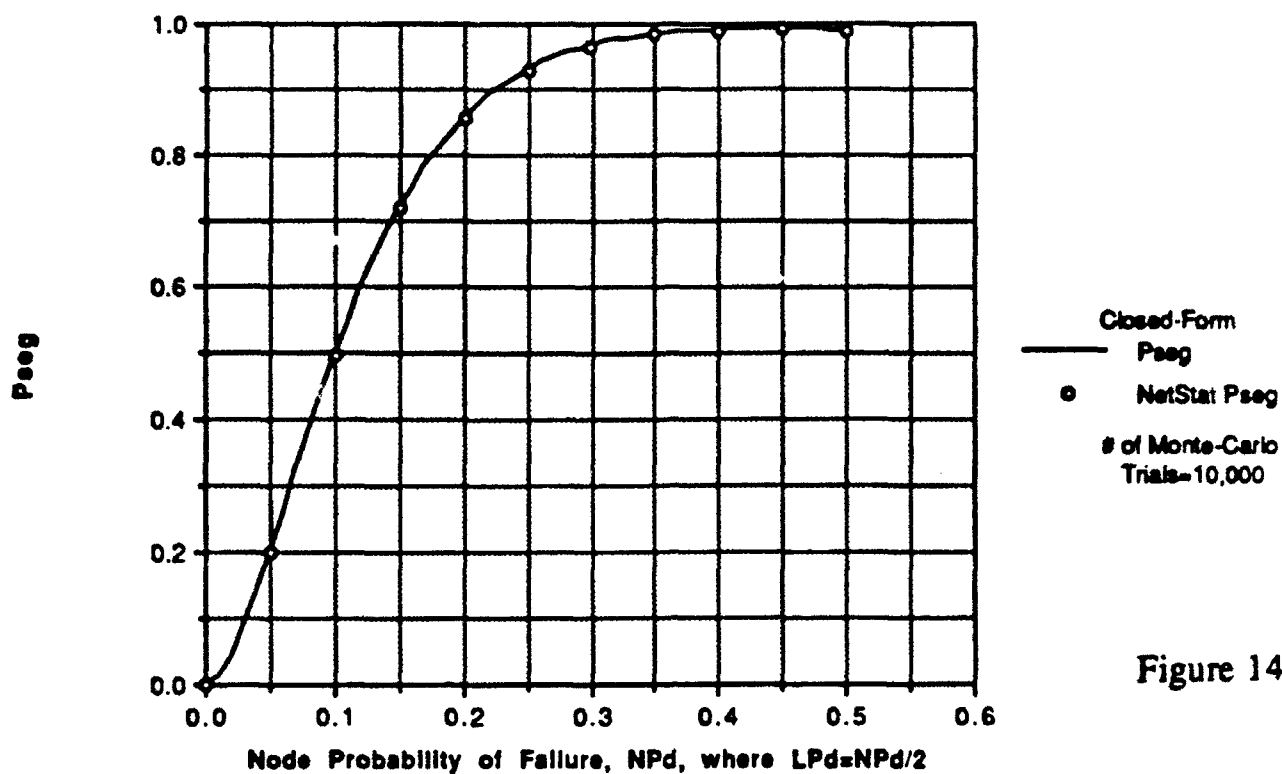


Figure 14b

Comparison of Closed Form Solution to Simulated Results for 12 Node Simplex Ring

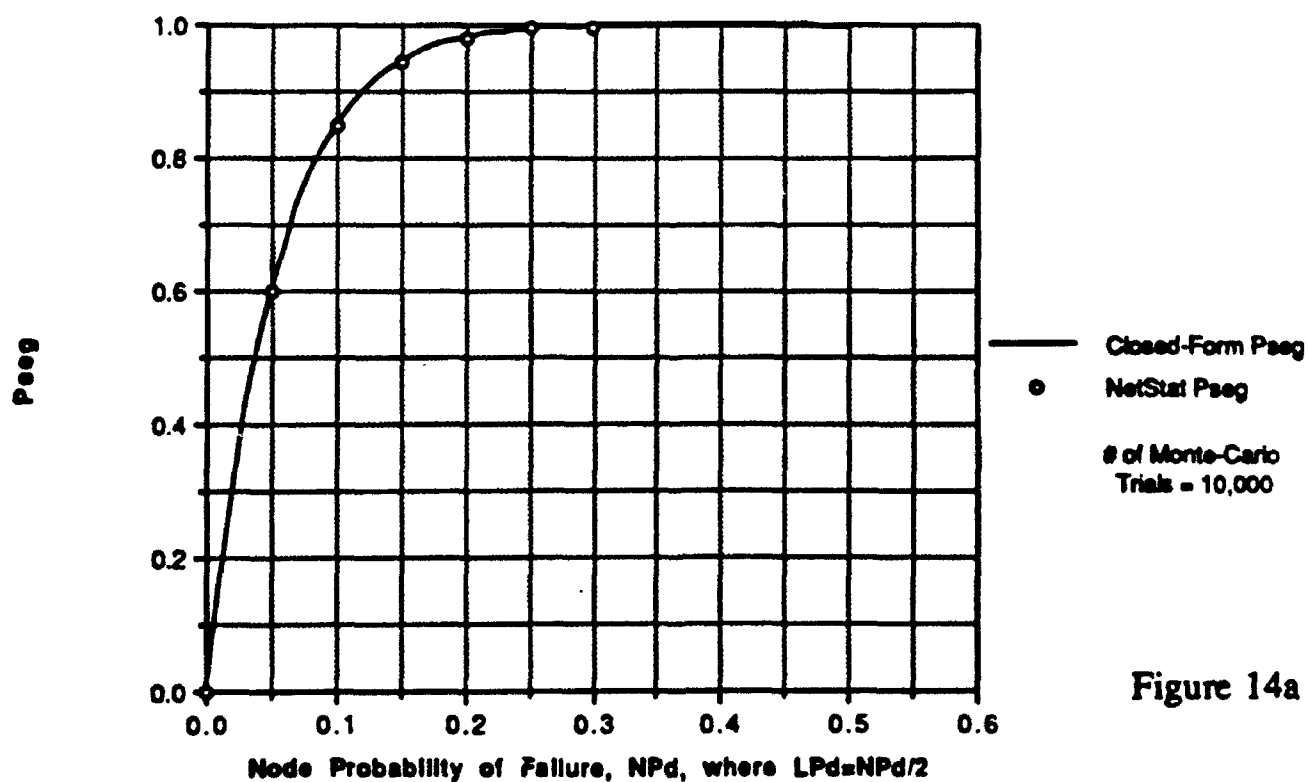


Figure 14a

**Graph Showing Convergence of Monte-Carlo
Simulation with Number of Trials**

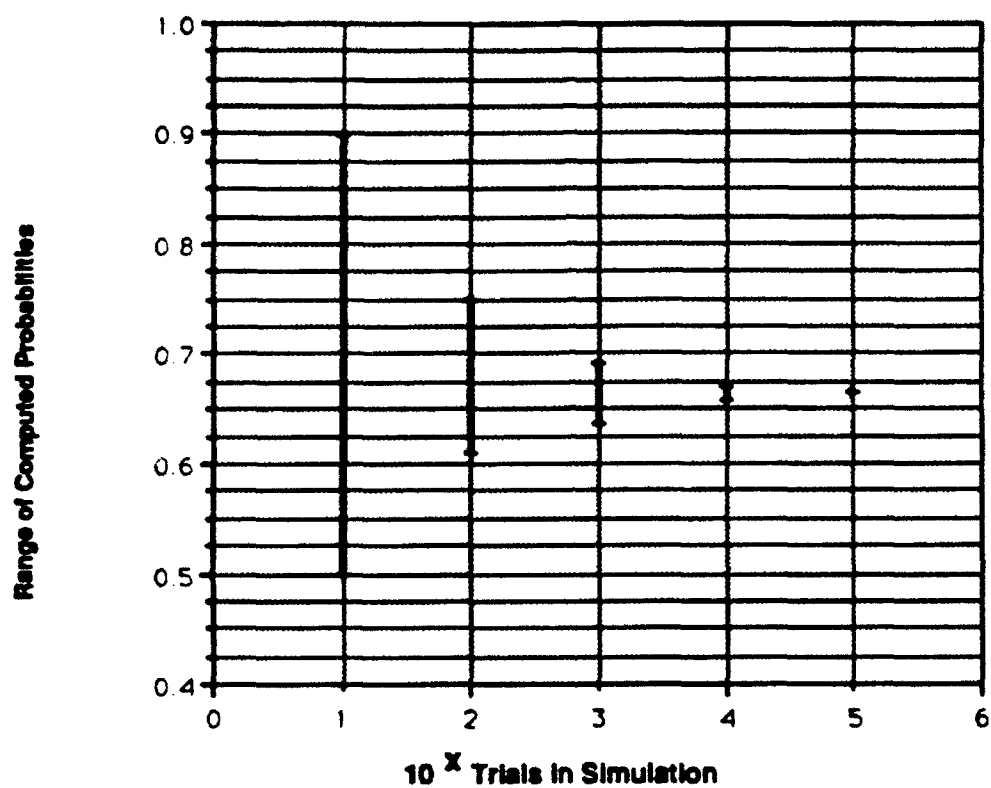


Figure 15

6.0 Summary:

NetStat has been demonstrated to be a useful tool in assessing the reliability and connectivity of networks. It is able to solve, through the use of Monte-Carlo simulation, a wide array of problems which are otherwise intractable when approached in a closed-form manner. The problem space addressed by NetStat is currently of great importance as infrastructures and information systems become more complex and new ones are designed.

Problems such as those addressed by NetStat have been treated in current literature to the extent that closed form solutions could be derived in certain constrained problems. The metrics addressed by NetStat are generally accepted by the "networking" community as valid measures of reliability. It is in the area where closed form expressions cannot be derived that NetStat finds its utility. Additionally NetStat can be used to cross-validate closed form expressions where a higher level of confidence is required.

It is the goal of the application to be generally applicable. To this end, NetStat provides a number of network elements for topology construction so that real world situations can be modeled as accurately as possible. These elements include specialized nodes such as hubs and bypass switches as well as bidirectional and unidirectional links. Failure probabilities for individual elements can be specified as the problem dictates. Once it is understood how NetStat treats the various components in the simulation, specialized scenarios can be constructed.

NetStat also provides an easy to use graphic user interface. In this way visual feed back is available during problem setup. It is the aim of the application to be intuitive to operate so that more time can be spent on interpreting results than on problem set up. NetStat is implemented on the well known Macintosh platform increasing its ease of use and accessibility. Finally, ranges of output data can be generated in an easily graphable form so that a broad view of the problem can be mapped out.

References and Bibliography:

- [1] Frank, H., et. al., Communication, Transmission, and Transportation Networks. Reading,MA: Addison-Wesley Co., 1971
- [2] Harary, F., Graph Theory. Reading,MA: Addison-Wesley Co., Second Ed., 1971
- [3] Deo, N., Graph Theory with Application to Engineering and Computer Science. , Englewood Cliffs , NJ: Prentice-Hall Inc., 1974
- [4] Tanenbaum, A., Computer Networks. Englewood Cliffs , NJ: Prentice-Hall Inc., 1981
- [5] Ed. Dallas, I.,et. al., Ring Technology Local Area Networks. Amsterdam,The Netherlands: Elsevier Science Pub. B.V., 1984
- [6] "On the Sample Sizes of Monte-Carlo Simulations - a Set of Useful Guidelines", The MITRE Corp. , Bedford, MA.
- [7] "Network Survivability Through Connectivity Optimization", The MITRE Corp. , Bedford, MA.
- [8] "A Survivable Network for Base Recovery After Attack", The MITRE Corp. , Bedford, MA.
- [9] Jeruchim, M., Techniques for Estimating the Bit Error Rate in the Simulation of Digital Communication Systems . IEEE J. of Selected Areas in Comm.,Jan 1984
- [10] Cobb, R., Use Statistics to Test Communications Systems Efficiently . EDN,Jan 1987
- [11] Lin, N.,et. al., Ring Network Reliability - The Probability that All Operative Nodes Can Communicate . IEEE Proc. 8th Symp. on Reliable Distributed Systems,Oct 1989
- [12] Lin, N.,et. al., A Reliability Comparison of Single and Double Rings . IEEE Proc. of InfoCom,Jun 1990
- [13] Boesch, F.,et. al., On Graphs of Invulnerable Communication Nets . IEEE Trans. on Circuit Theory,May 1970
- [14] Ed. Boesch, F.,et. al., Networks, An International Journal. New York: Wiley Journals, Vol. 15, Num. 2, Summer 1985
- [15] Ed. Boesch, F., Large Scale Networks, Theory and Design. New York: IEEE Press, 1976

APPENDIX A

NetStat, Version 1.2

User's Manual

G.S.Marzot

INTRODUCTION

The Network Survivable Topology Analytic Tool (NetStat) is a Macintosh application designed to aid a network architect or administrator in assessing the survivability/reliability of a given network topology. NetStat enables the user to graphically and interactively describe a network topology. The user is provided with several types of network elements(nodes,hubs,bypass-switches,unidirectional/bidirectional links) so that real-world topologies can be realized. After specifying destruction probabilities for each network element(individually or by type) the user can then run a Monte-Carlo simulation on the network to estimate certain connectivity metrics - the metrics available in this application are explained below. Tools are provided to edit, save and rearrange the topology so that a series of tradeoff scenarios may be investigated. In this way the user can allocate network resources in an intelligent way to maximize connectivity under a variety of conditions.

ABOUT BOX



Network Survivable Topology Analytic Tool (NetStat)
Version 1.2b by G. Marzot email:gmazot@linus.mitre.org

This Freeware application is an extension of the survivability analysis described in the MITRE document, MTR-10665

Aknowledgement to U.C.Georgopoulos and B.D.Metcalf

This application is also presented in partial fulfillment of my Tufts University MSEE project - Prof.Cheng, Advisor

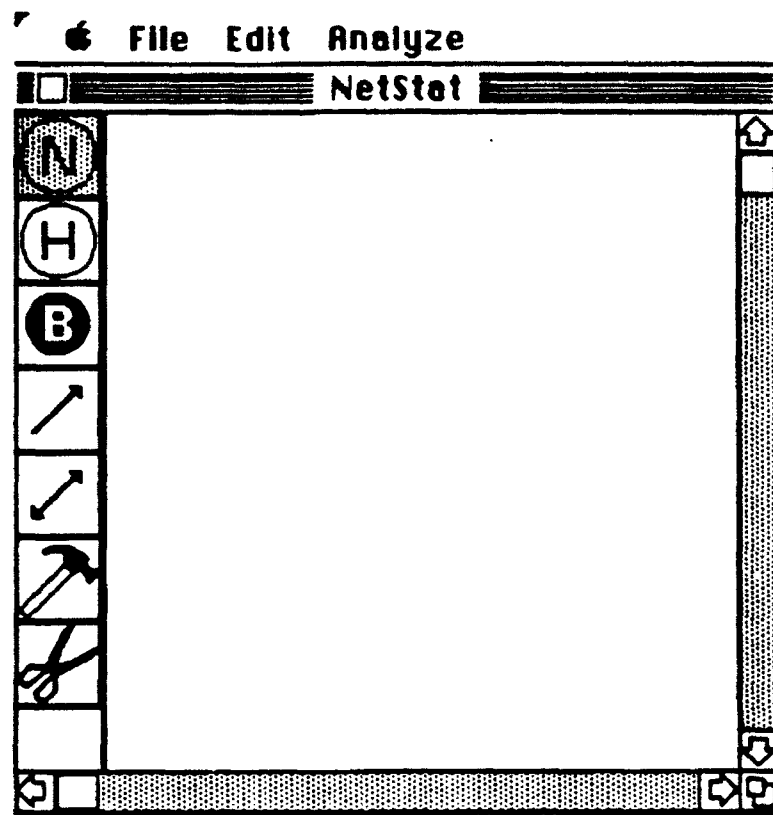
All Rights Reserved - 1990,1991

Portions of the Source Copyright© by the Symantec Corp.

OK

USER INTERFACE

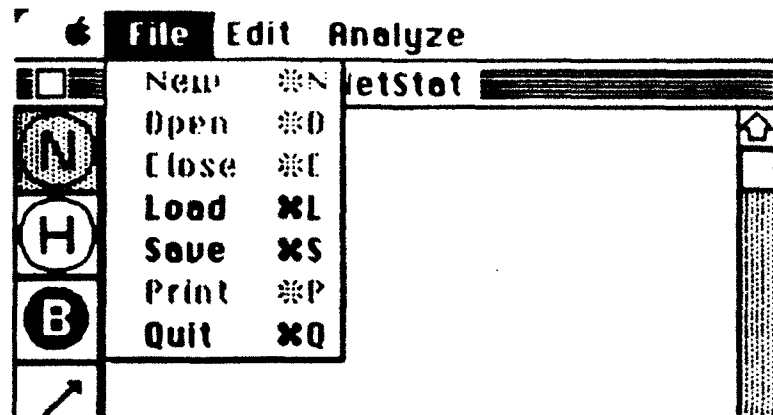
toolpalette:



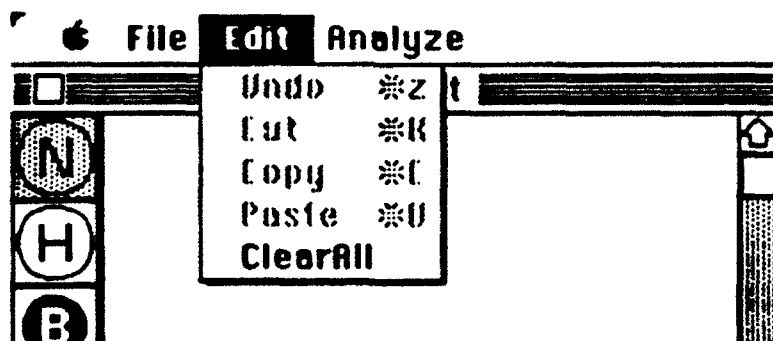
The network architectures are constructed in the application's drawing pad in the main part of the application window. In the left hand side of the application window a toolpalette is provided to assist in this operation. Each icon in the palette, when selected, corresponds to a mode that determines what will be drawn(or erased) in the drawing pad. A tool is selected by clicking on its icon in the palette. The icon remains selected (greyed) until the task associated with it is performed once then the tool becomes unselected and the cursor returns to an arrow. A tool may be indefinitely selected (locked/black) by double-clicking in the desired icon. The user may now perform the selected task a number of times. The user may change the tool by clicking in another icon or return to the default arrow tool by clicking in the blank area at the bottom of the palette. The tasks performed when each tool is selected are listed:

- Node Tool -** Add node to architecture by clicking in drawing pad, cursor is a square
- Hub Tool -** Add hub to architecture by clicking in drawing pad, cursor is a square
- Bypass Tool -** Add a bypass switch to architecture by clicking in drawing pad, cursor is a square
- Simplex Link Tool -** Join any two nodes(nodes,hubs or bypasses) by clicking in the source node then in the destination node(order determines direction), the cursor is a line with one arrow
- Duplex Link Tool -** Join any two nodes by clicking in the source node then in the destination node(order does not matter), the cursor is a line with two arrows
- Hammer Tool -** Remove any node by clicking in it, all links incident on the node are also removed, cursor is a hammer
- Scissor Tool -** Remove a link by clicking first in one terminal node then in the other(order does not matter) , the most recently added link will be removed, cursor is a pair of scissors
- Arrow Tool -** Default tool, Inspect the state of any network element (node or link) by clicking in(on) it and set the failure probability provided that option has been checked in the Simulation Parameters dialog box(see below), cursor is an arrow

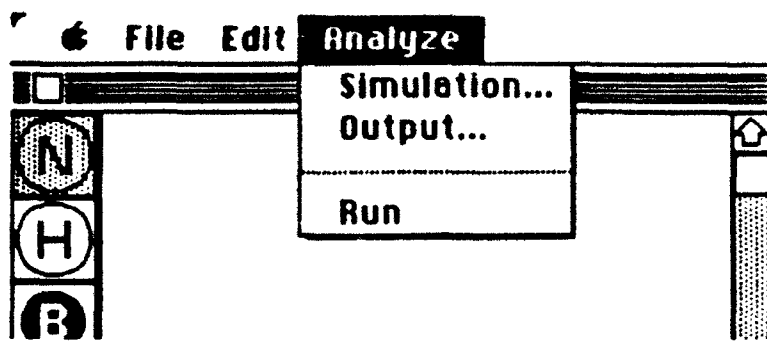
menus:



The **File** menu allows a preexistent architecture to be loaded from a file using the **Load** menu item. The architecture is usually stored in a file with the name "fname.arch". The **Save** menu item will allow a recently constructed topology to be saved to a file along with all present application settings. The **Quit** menu item quits the application. (note: clicking in the application close box also quits the application.) All other greyed items are unimplemented.



The **Edit** menu allows the user to remove all nodes and links in the drawing pad by selecting the **ClearAll** menu item. All other greyed items are unimplemented.



The **Analyze** menu allows the user to set parameters related to the simulation and the output format. By selecting the **Simulation...** menu item the Simulation Parameters and Settings dialog box is displayed(see below). By selecting the **Output...** menu item a dialog box displaying output specifications is displayed (see below). The **Run** menu item initiates a simulation in accordance with the present application settings.

dialogs:

Simulation Parameters and Settings

☒ **Monte Carlo** #of Monte Carlo States

☐ **Exhaustive** #of total system states 2^0

☐ **Individually Defined Destruction Probabilities**

☒ **Destruction Probabilities Set by Type (see below)**

				from	to	step size	
Node	Pd=	0.250	☒ or	0.000	0.500	0.100	☐
Hub	Pd=	0.250	☒ or	0.000	0.500	0.100	☐
Bypass	Pd=	0.250	☒ or	0.000	0.500	0.100	☐
Simplex Link	Pd=	0.125	☒ or	0.000	0.500	0.100	☐
Duplex Link	Pd=	0.125	☒ or	0.000	0.500	0.100	☐

The Simulation Parameters and Settings dialog box allows the user to specify the number of Monte-Carlo trials that will be generated in the simulation. As a reference the total number of system states, based on a binary state for each network element, is shown in a static text field. The exhaustive simulation is unimplemented. This would generate every realizable state for the system and would be unrealistic for most networks. A radio button is provided to choose between individually definable failure probabilities for the network elements or failure probabilities which are set by element type (i.e., node, hub, simplex link, etc.) . When the individually settable option is selected each element can be assigned a different Pd (see ToolPalette for details). When the "by type" option is selected all elements of the given type will have the same Pd. When the destruction probabilities are set by type there is a further option to make them constant or generate data over a range of destruction probabilities. (note: the choice of a ranges produces nested loops and will greatly increase simulation time, Best to run once with constant probabilities and then determine how many iterations are reasonable)

☒ Probability of segmentation	☐ Expected value of maximum segment length
☐ Average segment length	☐ Expected value of connected pairs
output filename: untitled 1	
OK Cancel	

The output settings dialog box determines which metrics will be generated and put out to a file when the Run menu item is selected. A check mark

next to the metric indicates that it will be calculated and dumped to the specified file in the **output filename** text field. A *special feature* is enabled when no metric is checked. The probability of segmentation will still be calculated when **Run** is selected but no file I/O will take place and the result will still be printed to the screen. The simulation will run up to *twice as fast* in this mode.

Element Record	
ID#:	1
TYPE:	NODE
Status:	ALIVE
Pd:	0.250
CANCEL OK	

Link Record	
ID#:	1
TYPE:	DPLX
Status:	ALIVE
Pd:	0.125
CANCEL OK	

These dialog boxes are displayed when the user clicks in or on any network element(node or link respectively) while the cursor is the default arrow. The Pd field may be edited to a probability between 0.0 and 1.0 inclusive provided the "individual setting" option has been checked in the Simulation Parameters dialog box.

CONNECTIVITY METRICS

When the Monte Carlo simulation is run an ensemble of the network states are generated and the connectivity metrics are computed over that ensemble. That means, that for one Monte-Carlo trial each network element may operational or non-operational, based on its Pd, resulting in a new network which is a subset of the original one. This resultant network can be analyzed in terms of its connectivity metrics (segmentation, segment size, connected pairs, etc.) and the result can be

stored. This process is repeated n times, where n is the number of Monte-Carlo trials, and the results are averaged over n producing estimates of the metrics.

Probability

of Segmentation - In graph theory jargon, this is the probability that a given graph will contain more than one component. Stated again, it is the probability that at least one surviving node will not have duplex connection with at least one other surviving node.

Average

Segment Length - Average segment length (normalized to the undamaged segment length)

Expected Value

of Connected Pairs - Connected pairs are defined as the number of pairs of surviving nodes that can communicate in a given graph. For each segment the number of connected pairs equals $N \text{ choose } 2$, or $N(N-1)/2$ where N is the number of nodes in that segment. (Normalized to undamaged number of connected pairs)

Expected Value of

Max. Segment Length - In each Monte-Carlo trial the size of the maximum segment is recorded and averaged over the ensemble of states. This is useful for investigating need for resource replication in distributed processing applications. (Normalized to undamaged segment length or total number of nodes)

FILE I/O

Output over ranges of input parameters can be computed and dumped to a file which is readable by CricketGraph™. The file is in tab delimited ASCII form and can be read by most graphing and plotting software.

Architectures can be saved to a file (Filename.arch) and reloaded later.

GRAPHING RESULTS W/ CricketGraph™

This is accomplished by selecting both the output file and Cricket Graph and then double-clicking on Cricket Graph. You may also read in the output file from CricketGraph by selecting the show text file option from within CricketGraphs **Open...** menu item dialog box. Appropriate column labels will be displayed.

LIMITATIONS/BUGS

The application is limited to 256 nodes with 256 links per node.

There is a limit of 4 links between any two nodes.

It is important to note that a 12 node system with 12 links simulating 10,000 Monte-Carlo states on a Mac IIcx takes approximately 10-15 seconds and that run time increases with the number of states and the number of total elements. Some simulations can run prohibitively long. Start with a small number of trials.

The application is designed to run on a Macintosh equipped with an 881/882 FPU, it will crash if one is not present.

The application will crash if you try to load an improper file - this bug will be fixed

GLOSSARY

Pd - the probability that a given network element will fail.

Nodes - Nodes represent network components such as terminals, modems, communication equipment or any similar items where users might reside or locations where messages are relayed. The term node is sometimes used to mean hubs and bypass switches as well.

Hubs - Hubs are the center nodes for star networks. Hubs are very similar to Nodes. The only difference is a conceptual one. The separate representation allows the individual setting of the hubs Pd as in a hardened hub star network where the hub has a Pd=0 and the nodes may have a non-zero Pd.

BypassSwitches - Bypass switches are usually three terminal components which can sense the loss of one connection and switch to the other. This allows nodes to be remoted from a network so that their loss has less of an impact on its connectivity. Bypasses are not counted in segment length, connected pairs, nor is a segment consisting solely of switches considered valid for the probability of segmentation. Nodes and Hubs are.

SimplexLinks - Simplex links are unidirectional communication links. This allows each path of a duplex link to be represented separately. Connection with simplex links is deemed valid only if a round trip path exists between the elements being considered.

Duplex Links - Duplex links provide bidirectional communication between the two nodes connected. Any two elements linked with an operational duplex link are considered connected.

APPENDIX B

NetStat, Version 1.2

Source Code

ConsVars Unit

unit ConsVars;

interface

const

 kSFSSaveDisk = \$214; {low memory nasties}
 kCurDirStore = \$398; { Negative of current volume refnum [WORD] }
 { DirID of current directory [LONG] }

 DEF_STATES = 1000; {default values for Simulation dialog box}
 DEF_NODE_PD = 0.25;
 DEF_LINK_PD = 0.125;
 DEF_FRM_PD = 0;
 DEF_TO_PD = 0.5;
 DEF_STP_SZ = 0.1;

 BASE_RES_ID = 400; {base resource ID for most res}

 NIL_POINTER = 0; {Parameters for various tool box calls}
 MOVE_TO_FRONT = -1;
 REMOVE_ALL_EVENTS = 0;
 SUSP_RES_MESS = 1;
 RES_MASK = 1;
 NO_EVENT = 0;

 ADD_CHECK_MARK = TRUE; {menu check parameters}
 REMOVE_CHECK_MARK = FALSE;

 DRAG_THRESHOLD = 10; {window parameters}
 SBarWidth = 15;
 TOOL_SIZE = 32;
 PAD_LIMIT = 1024;

 MAX_NEG = -32768; {integer range}
 MAX_POS = 32767 - 1;

 MIN_SLEEP = 0; {parameters for waitnextevent call/verify}
 NIL_MOUSE_REGION = 0;
 WNE_TRAP_NUM = \$60;
 UNIMPL_TRAP_NUM = \$9F;

 SIM_DLOG = BASE_RES_ID + 1; {dialog box res ID}
 OUT_DLOG = BASE_RES_ID + 2;
 ELEM_DLOG = BASE_RES_ID + 3;
 LINK_DLOG = BASE_RES_ID + 4;

 NEW_ITEM = 1; {file menu items}
 OPEN_ITEM = 2;
 CLOSE_ITEM = 3;
 LOAD_ITEM = 4;
 SAVE_ITEM = 5;
 PRINT_ITEM = 6;
 QUIT_ITEM = 7;

 UNDO_ITEM = 1; {edit menu items}

CUT_ITEM = 2;
COPY_ITEM = 3;
PASTE_ITEM = 4;
CLEAR_ITEM = 5;

SIM_ITEM = 1; {analyze menu items}
OUT_ITEM = 2;
RUN_ITEM = 4;

ABOUT_ITEM = 1; {apple menu item}

APPLE_MENU_ID = BASE_RES_ID; {menu res IDs}
FILE_MENU_ID = BASE_RES_ID + 1;
EDIT_MENU_ID = BASE_RES_ID + 2;
ANAL_MENU_ID = BASE_RES_ID + 3;

NODE_ICON = BASE_RES_ID; {tool/element icon res IDs}
HUB_ICON = BASE_RES_ID + 1;
BYP_ICON = BASE_RES_ID + 2;
SPLX_ICON = BASE_RES_ID + 3;
DPLX_ICON = BASE_RES_ID + 4;
CUTN_ICON = BASE_RES_ID + 5;
CUTL_ICON = BASE_RES_ID + 6;
NULL_ICON = BASE_RES_ID + 7;

ABOUT_ALERT = BASE_RES_ID; {alert res ID}

ELEM_CURS = BASE_RES_ID; {cursor res IDs}
CUT_NODE_CURS = BASE_RES_ID + 1;
ADD_SLNK_CURS = BASE_RES_ID + 2;
ADD_DLNK_CURS = BASE_RES_ID + 3;
CUT_LINK_CURS = BASE_RES_ID + 4;

DI_TOP = \$0050; {location for error alert for bad disk routine}
DI_LEFT = \$0070;

MAX_ITEMS = 127; {maximum nodes+bypass+hubs/links from one node}

ALINK = TRUE; {parameter for cutelem}
NOT_ALINK = FALSE;

PI = 3.141592654; { π }

type

PtrToLong = ^longint;
PtrToWord = ^integer;

{node/bypass/hub variables for set operations and bookkeeping}

NodeIDType = 0..255;
NodeSet = set of NodeIDType;
SetArrayType = array[0..127] of NodeSet;
SArrayPtr = ^SetArrayType;

{tool related types}

```

ToolStateType = (off, on, locked);
ComType = (AddNode, AddHub, AddByp, AddSplx, AddDplx, CutNode, CutLink, NullCom);
LinkComs = set of AddSplx..CutLink;
MapType = array[AddNode..NullCom] of Rect;

```

{network element types}

```

ElemType = (Node, Hub, Bypass, Dplx, Splx);
LinkElems = set of Dplx..Splx;

```

{Node like element record structure}

```

LinkPtr = ^Link;

ElementPtr = ^Element;
NodeArray = array[0..127] of ElementPtr;
LinkArray = array[0..127] of LinkPtr;
Element = record
  ID: NodeIDType;
  Kind: ElemType;
  Pd: Real;
  Loc: Rect;
  Alive: Boolean;
  LList: LinkArray
end;
FElement = record
  Kind: ElemType;
  Pd: Real;
  Loc: Rect;
end;

```

{link element record structure}

```

Link = record
  Kind: ElemType;
  End1ID, End2ID: integer;
  End1, End2: ElementPtr;
  HookPt1, A1pt1, A2pt1, HookPt2, A1pt2, A2pt2, Apt1, Apt2: Point;
  Alive: Boolean;
  Pd: real
end;
FLink = record
  Kind: ElemType;
  End1ID, End2ID: integer;
  Pd: real
end;

```

{Dialog element types and record}

```

DialogItemType = (Npd, Hpd, Bpd, Spd, Dpd, Nfr, Nto, Nst, Hfr, Hto, Hst, Bfr, Bto, Bst, Sfr, Sto, Sst, Dfr,
  Dto, {}
  Dst, Mst, Tst, Mon, Exh, Nc, Hc, Bc, Sc, Dc, Nr, Hr, Br, Sr, Dr, Ind, Typ, Paeg, Avel, Ecp, Emal);
DialogRec = record
  SimStrArr: array[Npd..Tst] of Str255;      {Npd=3, Sst=24}
  SimRadArr: array[Mon..Typ] of boolean;      {Mon=48, Dr=59, Ind=25,

```

```

    Typ=26,from/to/step=33-35)
    OutChkArr: array[Pseg..Emsl] of boolean;    {Pseg=5,Emsl=8}
    OutFileStr: Str255                          {  }
end;

TFilePtr = ^text;

ArchRec = record
    NodeArr: array[0..127] of FElement;
    LinkArr: array[0..127] of FLink;
    NumNodes: longint;
    NumLinks: longint;
    DlogRec: DialogRec
end;
ArchFile = file of ArchRec;
AFilePtr = ^ArchFile;

var
    gLinkSet: LinkElems;                        {set of link elements to distinguish from nodes}
    gLComSet: LinkComs;                        {set of link related commands to distinguish from node coms}
    gTState: ToolStateType;                    {present state of the tool panel}
    gPadWindow: WindowPtr;                     {global window storage}
    gSimDlogPtr, gOutDlogPtr, gElemDlogPtr, gLinkDlogPtr: DialogPtr;
                                                {Dialog Pointers to be loaded in initially and dereferenced
                                                later}
    gDlogRec: DialogRec;                        {record of dlog box item states}
    gDone,                                     {TRUE when time to leave app}
    gWNEImplemented,                           {TRUE when WaitNextEvent trap is implemented}
    gFirstClick,                               {TRUE for first click in content }
    gInBackground: Boolean;                     {TRUE when application is in background used for activate event}
    gTheEvent: EventRecord;                     {the event}
    gAppleMenu, gAnslMenu: MenuHandle; {menuhandles}
    gDragRect: Rect;                           {window drag limits}
    gVScroll, gHScroll: ControlHandle; {scroll bar handles}
    gNumSegs, gTotalSegs, gNumStates, gSegCount, gCurrentTime, gOldTime, gFirstID, gNumNodes,
    gNumLinks: longint;
    gFileName: Str255;
    gOrigin: Point;
    gPseg, gAvsl, gEcp, gEmsl, gPinch: extended;
    gUpdateRgn: RgnHandle;
    gMode: ComType;
    gTMap: MapType;
    gNodeArr: NodeArray;
    gNilLinkArr, gLinkArr: LinkArray;
    gMaxSegArr, gSegLnthArr: array[0..MAX_ITEMS] of longint;
    gWhenFirstClick: longint;
    gBypSet, gNodeSet: NodeSet;
    gWorld: SysEnvRec;
    gSFSaveDisk: PtrToWord;
    gCurDirStore: PtrToLong;
implementation
end.

```

Main

```

{$!+}
program Stat (input, output);
uses
  ConsVars, Engine;

```

```

procedure Grow (w: WindowPtr; p: Point);
forward;

```

```

(* PathNameFromDirID

```

```

.....)

```

```

{(*)
{(*) Given a DirID and real vRefnum, this routine will create and return the)
{(*) full pathname that corresponds to it. It does this by calling PBGetCatInfo)
{(*) for the given directory, and finding out its name and the DirID of its)
{(*) parent. It then performs the same operation on the parent, sticking ITS)
{(*) name onto the beginning of the first directory. This whole process is)
{(*) carried out until we have processed the root directory (identified with)
{(*) a DirID of 2.)
{(*)
{(*)}

```

```

function PathNameFromDirID (DirID: longint; vRefnum: integer): str255;

```

```

var

```

```

  Block: CInfoPBRec;
  directoryName, FullPathName: str255;
  err: OSerr;

```

```

begin

```

```

  FullPathName := '';

```

```

  with block do

```

```

    begin

```

```

      ioNamePtr := @directoryName;

```

```

      ioDrParID := DirID;

```

```

    end;

```

```

  repeat

```

```

    with block do

```

```

      begin

```

```

        ioVRefNum := vRefNum;

```

```

        ioFDirIndex := -1;

```

```

        ioDrDirID := block.ioDrParID;

```

```

      end;

```

```

      err := PBGetCatInfo(@Block, FALSE);

```

```

      directoryName := concat(directoryName, ':');

```

```

      fullPathName := concat(directoryName, fullPathName);

```

```

  until (block.ioDrDirID = 2);

```

```

  PathNameFromDirID := fullPathName;

```

```

end;

```

```

(* PathNameFromWD

```

```

.....)

```

```

(*)
{(*) Given an HFS working directory, this routine returns the full pathname}
{(*) that corresponds to it. It does this by calling PBGetWDInfo to get the}
{(*) VRefNum and DirID of the real directory. It then calls PathNameFromDirID,}
{(*) and returns its result.}
{(*)}
{(*) .....}
{(*) .....}

```

```
function PathNameFromWD (vRefNum: longint): str255;
```

```

var
  myBlock: WDPBRec;
  err: OSerr;
begin
  with myBlock do
    begin
      ioNamePtr := nil;
      ioVRefNum := vRefNum;
      ioWDIndex := 0;
      ioWDProcID := 0;
    end;

    { Change the Working Directory number in vRefnum into a real vRefnum }
    { and DirID. The real vRefnum is returned in ioVRefnum, and the real }
    { DirID is returned in ioWDDirID. }

    err := PBGetWDInfo(@myBlock, FALSE);

    with myBlock do
      PathNameFromWD := PathNameFromDirID(ioWDDirID, ioWDVRefnum)
    end;

```

```

{/..... Binomial ...../}
function Binomial (num: integer): integer;
var
  i, p: integer;
begin
  p := 1;
  if num > 1 then
    p := (((num - 1) * num) div 2)
  else
    p := 0;
  Binomial := p
end;

{/..... DoubleClick ...../}
function DoubleClick: boolean;
begin
  if TickCount > (gWhenFirstClick + GetDbtTime) then

```

```

begin
  gWhenFirstClick := TickCount;
  DoubleClick := FALSE
end
else
begin
  DoubleClick := TRUE;
  gWhenFirstClick := 0
end
end;

```

```

{ /***** RNumToString *****/ }

```

```

procedure RNumToString (num: real; var theString: Str255);
var
  myString1, myString2, myString3: Str255;
  i: integer;
begin
  NumToString(trunc(num), myString3);
  num := Abs(num - trunc(num));
  i := round(num * 1000);
  myString2 := '.';
  NumToString(i, myString1);
  if ((i < 100) and (i >= 10)) then
    myString1 := concat('0', myString1)
  else if i < 10 then
    myString1 := concat('00', myString1);
  theString := concat(myString3, myString2, myString1)
end;

```

```

{ /***** StringToRNum *****/ }

```

```

procedure StringToRnum (theString: Str255; var num: real);
var
  myString1, myString2: Str255;
  i, j, k: integer;
  n: longint;
begin
  i := pos('.', theString);
  if i = 0 then
    begin
      StringToNum(theString, n);
      num := n;
      Exit(StringToRNum)
    end;

  k := pos('-', theString);
  if k = 0 then
    begin
      myString1 := copy(theString, 1, i - 1);
      StringToNum(myString1, n);
      j := 1
    end

```

```

    else
    begin
        myString1 := copy(theString, k + 1, i - 1 - k);
        StringToNum(myString1, n);
        j := -1;
    end;

    num := n;
    myString1 := copy(theString, i + 1, 1);
    StringToNum(myString1, n);
    num := num + (n / 10);
    myString1 := copy(theString, i + 2, 1);
    StringToNum(myString1, n);
    num := num + (n / 100);
    myString1 := copy(theString, i + 3, 1);
    StringToNum(myString1, n);
    num := num + (n / 1000);
    num := num * j;
end;

{..... IsDAWindow .....}
function IsDAWindow (w: WindowPtr): Boolean;
begin
    if w = nil then
        IsDAWindow := FALSE
    else
        IsDAWindow := (WindowPeek(w)^.windowkind < 0)
    end;
end;

{..... IsAppWindow .....}
function IsAppWindow (w: WindowPtr): Boolean;
begin
    if w = nil then
        IsAppWindow := FALSE
    else
        IsAppWindow := (WindowPeek(w)^.windowkind = userkind)
    end;
end;

{..... SetScreen.....}
procedure SetScreen;
var
    theRect: Rect;
begin
    theRect := gPadWindow^.portRect;
    theRect.right := theRect.right - SBarWidth;
    theRect.bottom := theRect.bottom - SBarWidth;
    theRect.left := theRect.left + TOOL_SIZE + 2;
    OffsetRect(theRect, gOrigin.h, gOrigin.v);
    ClipRect(theRect);
    SetOrigin(gOrigin.h, gOrigin.v)
end;

{..... ResetScreen.....}
procedure ResetScreen;

```

```

begin
  SetOrigin(0, 0);
  ClipRect(screenBits.bounds)
end;

{/..... DrawArrows ...../}
procedure DrawArrows (theLink: LinkPtr);
var
  rise, run: integer;
  pt1, pt2: Point;
  ap, ep1, ep2: point;
begin
  with theLink^ do
    begin
      PenSize(2, 2);
      MoveTo(A1pt1.h, A1pt1.v);
      LineTo(Apt1.h, Apt1.v);
      MoveTo(A2pt1.h, A2pt1.v);
      LineTo(Apt1.h, Apt1.v);
      If kind = Dplx then
        begin
          MoveTo(A1pt2.h, A1pt2.v);
          LineTo(Apt2.h, Apt2.v);
          MoveTo(A2pt2.h, A2pt2.v);
          LineTo(Apt2.h, Apt2.v);
        end
      end;
      PenSize(1, 1)
    end;
end;

{/..... DrawCont ...../}

procedure DrawCont;
var
  theRect: Rect;
  myTimeString: Str255;
  ih: Handle;
  ph: PicHandle;
  i, j: integer;
begin
  for i := 1 to gNumNodes do
    begin
      j := 1;
      ih := GetIcon(NODE_ICON + ord(gNodeArr[i]^kind) - ord(Node));
      PlotIcon(gNodeArr[i]^Loc, ih);
      while gNodeArr[i]^LList[j] <> nil do
        begin
          MoveTo(gNodeArr[i]^LList[j]^HookPt1.h, gNodeArr[i]^LList[j]^HookPt1.v);
          LineTo(gNodeArr[i]^LList[j]^HookPt2.h, gNodeArr[i]^LList[j]^HookPt2.v);
          DrawArrows(gNodeArr[i]^LList[j]);
          j := j + 1
        end
      end
    end
  end
end

```

end
end;

```
{/..... GrayIcon ...../}
procedure GrayIcon (com: ComType);
var
  ih: Handle;
begin
  ih := GetIcon(NODE_ICON + ord(com) - ord(AddNode));
  PlotIcon(gTMap[com], ih);
  PenMode(PatOr);
  PenPat(!!Gray);
  PaintRect(gTMap[com]);
  PenPat(black);
  PenMode(patCopy)
end;
```

```
{/ ..... DrawTools ..... /}
```

```
procedure DrawTools;
var
  ih: Handle;
  tRect: Rect;
  com: ComType;
  prevBottom: integer;
begin
  PenSize(1, 1);
  prevBottom := gPadWindow^.portRect.top;
  for com := AddNode to CutLink do
    begin
      gTMap[com] := gPadWindow^.portRect;
      gTMap[com].top := prevBottom;
      gTMap[com].right := gTMap[com].left + TOOL_SIZE;
      gTMap[com].bottom := gTMap[com].top + TOOL_SIZE;
      prevBottom := gTMap[com].bottom;
      ih := GetIcon(NODE_ICON + ord(com) - ord(AddNode));
      PlotIcon(gTMap[com], ih);
      FrameRect(gTMap[com])
    end;
  SetRect(gTMap[NullCom], 0, 0, 0, 0);
  MoveTo(gTMap[CutLink].left, gTMap[CutLink].bottom);
  LineTo(gTMap[CutLink].right + 1, gTMap[CutLink].bottom);
  If gMode <> NullCom then
    case gTState of
      on:
        begin
          GrayIcon(gMode)
        end;
      locked:
        begin
          InvertRect(gTMap[gMode]);
        end;
    end;
  end;
```

```

    end;
  off:
  end;
  PenSize(2, 1);
  tRect := gPadWindow^.portRect;
  tRect.right := tRect.left + TOOL_SIZE;
  MoveTo(tRect.right, tRect.top);
  LineTo(tRect.right, tRect.bottom - SBarWidth);
  PenSize(1, 1)
end;

```

```

{ / ..... DrawWindow ..... }

```

```

procedure DrawWindow (theWindow: WindowPtr);
var
  theRect: Rect;
  myTimeString: Str255;
begin
  SetScreen;
  DrawCont;
  ResetScreen;
  DrawTools
end;

```

```

{ / ..... UpdateWindow ..... }

```

```

procedure UpdateWindow (theWindow: WindowPtr);
var
  savePort: GrafPtr;
begin
  GetPort(savePort);
  SetPort(theWindow);
  BeginUpdate(theWindow);
  If not EmptyRgn(theWindow^.visRgn) then
    begin
      EraseRect(theWindow^.portRect);
      DrawWindow(theWindow);
      DrawControls(theWindow);
      DrawGrowIcon(theWindow)
    end;
  EndUpdate(theWindow);
  SetPort(savePort)
end;

```

```

{ / ..... Grow ..... }

```

```

procedure Grow (w: WindowPtr; p: Point);
var
  savePort: GrafPtr;
  theResult: longint;
  oScroll: integer;

```

```

    r, oView, viewRect: Rect;
begin
    GetPort(savePort);
    SetPort(w);
    SetRect(r, TOOL_SIZE + 18, 7 * TOOL_SIZE + 18, screenBits.bounds.right, screenBits.bounds.bottom);
    theResult := GrowWindow(w, p, r);
    If (theResult <> 0) then
        begin
            viewRect := w^.portRect;
            InvalRect(gVScroll^^.ctrlRect);
            InvalRect(gHScroll^^.ctrlRect);
            EraseRect(gVScroll^^.ctrlRect);
            EraseRect(gHScroll^^.ctrlRect);
            viewRect.left := viewRect.right - SBarWidth;
            viewRect.top := viewRect.bottom - SBarWidth;
            InvalRect(viewRect);
            EraseRect(viewRect);
            SizeWindow(w, LoWord(theResult), HiWord(theResult), TRUE);
            HidePen;
            MoveControl(gVScroll, w^.portRect.right - SBarWidth, w^.portRect.top - 1);
            SizeControl(gVScroll, SBarWidth + 1, w^.portRect.bottom - w^.portRect.top - (SBarWidth - 2));
            MoveControl(gHScroll, w^.portRect.left - 1, w^.portRect.bottom - SBarWidth);
            SizeControl(gHScroll, w^.portRect.right - w^.portRect.left - (SBarWidth - 2), SBarWidth + 1);
            ShowPen;
            UpdateWindow(w);
            SetPort(savePort)
        end
    end;
end;

```

```

{ / ..... HandleNull ..... / }

```

```

{/..... SaveSettings ...../}

```

```

procedure SaveSettings (theDlg: DialogPtr);

```

```

var

```

```

    itemType: integer;

```

```

    i: DialogItemType;

```

```

    itemHandle: Handle;

```

```

    itemRect: rect;

```

```

begin

```

```

    with gDlgRec do

```

```

        begin

```

```

            if theDlg = gSimDlgPtr then

```

```

                begin

```

```

                    for i := Npd to Mst do

```

```

                        begin

```

```

                            GetDItem(gSimDlgPtr, (ord(i) - ord(Npd) + 3), itemType, itemHandle, itemRect);

```

```

                            GetIText(itemHandle, SimStrArr[i]);

```

```

                        end;

```

```

                    for i := Mnn to Typ do

```

```

                        begin

```

```

                            if i < Ind then

```

```

                                begin

```

```

        GetDItem(gSimDlogPtr, (ord(i) - ord(Mon) + 48), itemType, itemHandle, itemRect);
        SimRadArr[i] := Boolean(GetCtlValue(ControlHandle(itemHandle)))
    end
  else
    begin
      GetDItem(gSimDlogPtr, (ord(i) - ord(Ind) + 25), itemType, itemHandle, itemRect);
      SimRadArr[i] := Boolean(GetCtlValue(ControlHandle(itemHandle)))
    end;
  end
end
end
else
  begin
    for i := Pseg to Emsl do
      begin
        GetDItem(gOutDlogPtr, (ord(i) - ord(Pseg) + 5), itemType, itemHandle, itemRect);
        OutChkArr[i] := Boolean(GetCtlValue(ControlHandle(itemHandle)))
      end;
      GetDItem(gOutDlogPtr, 3, itemType, itemHandle, itemRect);
      GetIText(itemHandle, OutFileStr)
    end
  end
end;
end;

```

{/..... Restore Settings/}

```

procedure RestoreSettings (theDlog: DialogPtr);

```

```

var

```

```

  i: DialogItemType;
  itemType: integer;
  itemHandle: Handle;
  itemRect: rect;

```

```

begin

```

```

  with gDlogRec do

```

```

    begin

```

```

      BringToFront(theDlog);

```

```

      If theDlog = gSimDlogPtr then

```

```

        begin

```

```

          GetDItem(gSimDlogPtr, 25, itemType, itemHandle, itemRect);

```

```

          SetCtlValue(ControlHandle(itemHandle), Integer(SimRadArr[Ind]));

```

```

          GetDItem(gSimDlogPtr, 26, itemType, itemHandle, itemRect);

```

```

          SetCtlValue(ControlHandle(itemHandle), Integer(SimRadArr[Typ]));

```

```

          for i := Npd to Mst do

```

```

            begin

```

```

              GetDItem(gSimDlogPtr, (ord(i) - ord(Npd) + 3), itemType, itemHandle, itemRect);

```

```

              SetIText(itemHandle, SimStrArr[i]);

```

```

            end;

```

```

          for i := Mon to Dr do

```

```

            begin

```

```

              GetDItem(gSimDlogPtr, (ord(i) - ord(Mon) + 48), itemType, itemHandle, itemRect);

```

```

              SetCtlValue(ControlHandle(itemHandle), Integer(SimRadArr[i]));

```

```

            end;

```

```

        end

```

```

    else
    begin
        for i := Pseg to Emsl do
        begin
            GetDItem(gOutDialogPtr, (ord(i) - ord(Pseg) + 5), itemType, itemHandle, itemRect);
            SetCtlValue(ControlHandle(itemHandle), Integer(OutChkArr[i]));
        end;
        GetDItem(gOutDialogPtr, 3, itemType, itemHandle, itemRect);
        SetIText(itemHandle, OutFileStr)
    end
end
end;

{/ ..... HandleAppleChoice ..... /}

procedure HandleAppleChoice (theItem: integer);
var
    accName: Str255;
    n, accNumber: integer;
    itemNumber: Byte;
    AboutDialog: DialogPtr;
begin
    case (theItem) of
        ABOUT_ITEM:
            n := NoteAlert(ABOUT_ALERT, nil);
        otherwise
            begin
                GetItem(gAppleMenu, theItem, accName);
                accNumber := OpenDeskAcc(accName)
            end
    end
end;

{/..... HiliTool ...../}
procedure HiliTool (com: ComType);
var
    theRect: Rect;
    ih: Handle;
    dbl: boolean;
begin
    dbl := DoubleClick;
    If (gMode = com) then
    begin
        If not dbl then
        begin
            If gTState = on then
            begin
                gMode := NullCom;
                gTState := off;
                gFirstClick := TRUE;
                gFirstID := 0;
                ih := GetIcon(NODE_ICON + ord(com) - ord(AddNode));
                PlotIcon(gTMap[com], ih)
            end
        end
    end
end;

```

```

        end
    else
        begin
            If com <> NullCom then
                begin
                    gTState := on;
                    GrayIcon(com)
                end
            end
        end
    else
        begin
            If com <> NullCom then
                begin
                    gTState := locked;
                    ih := GetIcon(NODE_ICON + ord(com) - ord(AddNode));
                    PlotIcon(gTMap[com], ih);
                    InvertRect(gTMap[com])
                end
            end
        end
    else
        begin
            If com <> NullCom then
                begin
                    gTState := on;
                    ih := GetIcon(NODE_ICON + ord(gMode) - ord(AddNode));
                    PlotIcon(gTMap[gMode], ih);
                    gMode := com;
                    GrayIcon(com)
                end
            else
                begin
                    ih := GetIcon(NODE_ICON + ord(gMode) - ord(AddNode));
                    PlotIcon(gTMap[gMode], ih);
                    gMode := com;
                    gTState := off
                end
            end
        end
    end;
end;

```

```

{***** FindElem *****/}

```

```

function FindElem (pt: point): integer;

```

```

var

```

```

    p: point;

```

```

    done: boolean;

```

```

    Elem: integer;

```

```

begin

```

```

    p.h := pt.h + gOrigin.h;

```

```

    p.v := pt.v + gOrigin.v;

```

```

    Elem := 0;

```

```

    done := FALSE;

```

```

    If gNumNodes > 0 then

```

```

repeat
  Elem := Elem + 1;
  done := PtInRect(p, gNodeArr[Elem]^Loc);
until (done or (Elem = gNumNodes));
if not done then
  Elem := 0;
FinJElem := Elem
end;

```

```

(/..... GetLinkNum ...../)

```

```

function GetLinkNum (pt: point): integer;

```

```

var

```

```

  p: point;

```

```

  horiz, vert, done: boolean;

```

```

  rise1, rise2, run1, run2, link: integer;

```

```

begin

```

```

  p.h := pt.h + gOrigin.h;

```

```

  p.v := pt.v + gOrigin.v;

```

```

  link := 0;

```

```

  done := FALSE;

```

```

  if gNumLinks > 0 then

```

```

    repeat

```

```

      link := link + 1;

```

```

      with gLinkArr[link]^ do

```

```

        begin

```

```

          vert := (HookPt1.h = HookPt2.h);

```

```

          horiz := (HookPt1.v = HookPt2.v);

```

```

          if ((p.h <= HookPt1.h) and (p.h >= HookPt2.h)) or ((p.h <= HookPt2.h) and (p.h >= HookPt1.h)) then

```

```

            if ((p.v <= HookPt1.v) and (p.v >= HookPt2.v)) or ((p.v <= HookPt2.v) and (p.v >= HookPt1.v))

```

```

          then

```

```

            begin

```

```

              run1 := HookPt1.h - HookPt2.h;

```

```

              rise1 := HookPt1.v - HookPt2.v;

```

```

              run2 := HookPt1.h - p.h;

```

```

              rise2 := HookPt1.v - p.v;

```

```

              if (horiz and (abs(p.v - HookPt1.v) = 0)) then

```

```

                done := TRUE

```

```

              else if (vert and (abs(p.h - HookPt1.h) = 0)) then

```

```

                done := TRUE

```

```

              else if (abs(round((rise1 / run1) * run2) - rise2) <= 1) then

```

```

                done := TRUE

```

```

              else if (abs(round((run1 / rise1) * rise2) - run2) <= 1) then

```

```

                done := TRUE;

```

```

            end;

```

```

        end;

```

```

      until (done or (link = gNumLinks));

```

```

    if not done then

```

```
link := 0;  
GetLinkNum := link  
end;
```

```
{/..... PopElemDLOG ...../}  
procedure PopElemDLOG (theElem: integer);  
var  
  dun: boolean;  
  IDStr: Str255;  
  TypeStr: Str255;  
  StatusStr: Str255;  
  PdStr: Str255;  
  theNewPd: real;  
  itemType: integer;  
  itemHandle: Handle;  
  itemRect: rect;  
  item: integer;  
  result: integer;  
begin  
  BringToFront(gElemDlogPtr);  
  
  NumToString(theElem, IDStr);  
  GetDItem(gElemDlogPtr, 4, itemType, itemHandle, itemRect);  
  SetIText(itemHandle, IDStr);  
  
  case gNodeArr[theElem]^Kind of  
    Hub:  
      TypeStr := 'HUB';  
    Node:  
      TypeStr := 'NODE';  
    Bypass:  
      TypeStr := 'BYPASS';  
    otherwise  
  end;  
  GetDItem(gElemDlogPtr, 6, itemType, itemHandle, itemRect);  
  SetIText(itemHandle, TypeStr);  
  
  If gNodeArr[theElem]^Alive then  
    StatusStr := 'ALIVE'  
  else  
    StatusStr := 'DEAD';  
  GetDItem(gElemDlogPtr, 8, itemType, itemHandle, itemRect);  
  SetIText(itemHandle, StatusStr);  
  
  RNumToString(gNodeArr[theElem]^Pd, PdStr);  
  GetDItem(gElemDlogPtr, 10, itemType, itemHandle, itemRect);  
  SetIText(itemHandle, PdStr);  
  
  ShowWindow(gElemDlogPtr);  
  while not dun do  
    begin  
      ModalDialog(nil, item);  
      case item of
```

```

1:
begin
  GetDItem(gElemDlogPtr, 10, ItemType, ItemHandle, ItemRect);
  GetIText(itemHandle, PdStr);
  StringToRNum(PdStr, theNewPd);
  If (theNewPd <= 1.0) and (theNewPd >= 0) then
    begin
      If (theNewPd <> gNodeArr[theElem]^Pd) then
        begin
          If (gDlogRec.SimRadArr[Ind]) then
            begin
              gNodeArr[theElem]^Pd := theNewPd;
              Hidewindow(gElemDlogPtr);
              dun := TRUE;
            end
          else
            result := NoteAlert(402, nil);
          end
        else
          begin
            Hidewindow(gElemDlogPtr);
            dun := TRUE;
          end;
        end
      else
        begin
          RNumToString(gNodeArr[theElem]^Pd, PdStr);
          GetDItem(gElemDlogPtr, 10, ItemType, ItemHandle, ItemRect);
          SetIText(itemHandle, PdStr);
          result := NoteAlert(401, nil);
        end;
      end;
    end;
  2:
  begin
    HideWindow(gElemDlogPtr);
    dun := TRUE
  end;
end;
end;
end;

{..... PopLinkDLOG .....}
procedure PopLinkDLOG (theLink: integer);
var
  dun: boolean;
  IDStr: Str255;
  TypeStr: Str255;
  StatusStr: Str255;
  PdStr: Str255;
  theNewPd: real;
  ItemType: integer;
  ItemHandle: Handle;
  ItemRect: rect;

```

```
    Item: integer;
    result: integer;
begin
    BringToFront(gLinkDlogPtr);

    NumToString(theLink, IDStr);
    GetDItem(gLinkDlogPtr, 4, ItemType, ItemHandle, ItemRect);
    SetIText(itemHandle, IDStr);

    case gLinkArr[theLink]^Kind of
        Dplx:
            TypeStr := 'DPLX';
        Splx:
            TypeStr := 'SPLX';
        otherwise
    end;
    GetDItem(gLinkDlogPtr, 6, ItemType, ItemHandle, ItemRect);
    SetIText(itemHandle, TypeStr);

    If gLinkArr[theLink]^Alive then
        StatusStr := 'ALIVE'
    else
        StatusStr := 'DEAD';
    GetDItem(gLinkDlogPtr, 8, ItemType, ItemHandle, ItemRect);
    SetIText(itemHandle, StatusStr);

    RNumToString(gLinkArr[theLink]^Pd, PdStr);
    GetDItem(gLinkDlogPtr, 10, ItemType, ItemHandle, ItemRect);
    SetIText(itemHandle, PdStr);

    ShowWindow(gLinkDlogPtr);
    while not dun do
        begin
            ModalDialog(nil, Item);
            case Item of
                1:
                    begin
                        GetDItem(gLinkDlogPtr, 10, ItemType, ItemHandle, ItemRect);
                        GetIText(itemHandle, PdStr);
                        StringToRNum(PdStr, theNewPd);
                        If (theNewPd <= 1.0) and (theNewPd >= 0) then
                            begin
                                If (theNewPd <> gLinkArr[theLink]^Pd) then
                                    begin
                                        If (gDlogRec.SimRadArr[Ind]) then
                                            begin
                                                gLinkArr[theLink]^Pd := theNewPd;
                                                HideWindow(gLinkDlogPtr);
                                                dun := TRUE;
                                            end
                                        else
                                            result := NoteAlert(402, nil);
                                        end
                                    end
                                end
                            end
                        end
                    end
            end
        end
```

```

        else
        begin
            HideWindow(gLinkDlogPtr);
            dun := TRUE;
        end;
    end
else
begin
    RNumToString(gLinkArr[theLink]^Pd, PdStr);
    GetDitem(gLinkDlogPtr, 10, ItemType, ItemHandle, ItemRect);
    SetIText(itemHandle, PdStr);
    result := NoteAlert(401, nil);
end;
end;
2:
begin
    HideWindow(gLinkDlogPtr);
    dun := TRUE;
end;
end;
end;
end;

{***** NumLinks *****/}
function NumLinks (ind1, ind2: integer): integer;

var
    i, j, k: integer;
begin
    j := 1;
    k := 0;
    while gNodeArr[ind1]^LList[j] <> nil do
        begin
            if gNodeArr[ind1]^LList[j]^End2 = gNodeArr[ind2] then
                k := k + 1;
            j := j + 1;
        end;
    j := 1;
    while gNodeArr[ind2]^LList[j] <> nil do
        begin
            if gNodeArr[ind2]^LList[j]^End2 = gNodeArr[ind1] then
                k := k + 1;
            j := j + 1;
        end;
    NumLinks := k;
end;

{***** AssignHooks *****/}
procedure AssignHooks (ID1, ID2: integer; var q1, h1, q2, h2: integer);
var
    rect1, rect2: Rect;
    rise, run: Real;

```

```

    linkNum, slope: integer;
begin
    rect1 := gNodeArr[ID1]^Loc;
    rect2 := gNodeArr[ID2]^Loc;
    rise := rect2.top - rect1.top;
    run := rect2.left - rect1.left;
    If abs(run) > abs(rise) then
    begin
        If run > 0 then
        begin
            q1 := 1;
            q2 := 2;
        end
        else
        begin
            q1 := 2;
            q2 := 1;
        end
    end
    else
    begin
        If rise > 0 then
        begin
            q1 := 4;
            q2 := 3;
        end
        else
        begin
            q1 := 3;
            q2 := 4;
        end
    end;
    h1 := NumLinks(ID1, ID2) + 1;
    h2 := h1;
    If h1 > 4 then
        q1 := 0;

end;

{/..... GetHookPt ...../}
function HookPt (tRect: Rect; q, h: integer): Point;
var
    p: Point;
    mult: integer;
begin
    If odd(h) then
        mult := -1;
    else
        mult := 1;
    case q of
        1:
            begin
                p.h := tRect.right - 1;

```

```

    p.v := tRect.top + (mult * (h div 2) + 1) * 5
end;
2:
begin
    p.h := tRect.left;
    p.v := tRect.top + (mult * (h div 2) + 1) * 5
end;
3:
begin
    p.v := tRect.top;
    p.h := tRect.left + (mult * (h div 2) + 1) * 5
end;
4:
begin
    p.v := tRect.bottom - 1;
    p.h := tRect.left + (mult * (h div 2) + 1) * 5
end;
otherwise
end;
HookPt := p
end;

{..... GetAPoints .....}
procedure GetAPoints (pt1, pt2: Point; var e1, e2, sp: Point);
var
    rise, run, slope, theta, deltas, deltac, sx, sy, ex1, ey1, ex2, ey2: real;
begin
    run := pt2.h - pt1.h;
    rise := pt2.v - pt1.v;
    if run < 0 then
        begin
            slope := rise / run;
            theta := -arctan(slope)
        end
    else if rise < 0 then
        theta := Pi / 2
    else
        theta := 3 * Pi / 2;

    sx := pt1.h + 2 * run / 3;
    sp.h := round(sx);
    sy := pt1.v + 2 * rise / 3;
    sp.v := round(sy);
    deltac := 3.8 * cos(Pi / 4.7 - theta);
    deltas := 3.8 * sin(Pi / 4.7 - theta);
    if run >= 0 then
        begin
            ex1 := sx - deltac;
            ey1 := sy - deltas;
            ex2 := sx - deltas;
            ey2 := sy + deltac
        end
    else

```

```

begin
  ex1 := sx + deltac;
  ey1 := sy + deltas;
  ex2 := sx + deltas;
  ey2 := sy - deltac
end;
e1.h := round(ex1);
e1.v := round(ey1);
e2.h := round(ex2);
e2.v := round(ey2);
end;

{..... AddElem .....}
procedure AddElem (tkind: ElemType; thePoint: Point);
var
  theElement: ElementPtr;
  theLElement: LinkPtr;
  theRect: rect;
  ph: PicHandle;
  ih: Handle;
  erro: boolean;
  result, i, j, id, Q1, H1, Q2, H2: integer;
  pt1, pt2: Point;
begin
  i := 0;
  j := 0;
  erro := FALSE;
  If tkind In gLinkSet then
    begin
      If gFirstClick then
        begin
          gFirstID := FindElem(thePoint);
          If gFirstID <> 0 then
            gFirstClick := False
          end
        end
      else
        begin
          Q1 := 0;
          id := FindElem(thePoint);
          If ((id <> 0) and (id <> gFirstID)) then
            begin
              gFirstClick := TRUE;
              New(theLElement);
              with theLElement^ do
                begin
                  kind := tkind;
                  Alive := TRUE;
                  End1ID := gFirstID;
                  End2ID := id;
                  End1 := gNodeArr[gFirstID];
                  End2 := gNodeArr[id]
                end;
            end;
          end;
        end
      end;
    end;
  end;
end;

```

```

repeat
  i := i + 1
until (gNodeArr[gFirstID]^LList[i] = nil) or (i = MAX_ITEMS);
if i = MAX_ITEMS then
  erro := TRUE;
if kind = Splx then
  StringToRNum(gDlogRec.SimStrArr[Spd], theLElement^.Pd)
else
  StringToRNum(gDlogRec.SimStrArr[Dpd], theLElement^.Pd);
repeat
  j := j + 1
until (gNodeArr[id]^LList[j] = nil) or (j = MAX_ITEMS);
if j >= MAX_ITEMS then
  erro := TRUE;
AssignHooks(gFirstID, id, Q1, H1, Q2, H2)
end;
if Q1 = 0 then
  erro := TRUE;
if not erro then
  begin
    with theLElement^ do
      begin
        HookPt1 := HookPt(gNodeArr[gFirstID]^Loc, Q1, H1);
        HookPt2 := HookPt(gNodeArr[id]^Loc, Q2, H2);
        GetAPoints(HookPt1, HookPt2, A1pt1, A2pt1, Apt1);
        GetAPoints(HookPt2, HookPt1, A1pt2, A2pt2, Apt2);
        SetScreen;
        MoveTo(HookPt1.h, HookPt1.v);
        LineTo(HookPt2.h, HookPt2.v);
        if kind = Dplx then
          gNodeArr[id]^LList[j] := theLElement;
        end;
        DrawArrows(theLElement);
        ResetScreen;
        gNodeArr[gFirstID]^LList[i] := theLElement;
        gNumLinks := gNumLinks + 1;
        gLinkArr[gNumLinks] := theLElement
      end
    end
  end
end
else
  begin
    New(theElement);
    theElement^.Loc.left := thePoint.h + gOrigin.h - 8;
    theElement^.Loc.top := thePoint.v + gOrigin.v - 8;
    theElement^.Loc.right := thePoint.h + gOrigin.h + 8;
    theElement^.Loc.bottom := thePoint.v + gOrigin.v + 8;
    theElement^.Kind := kind;
    case kind of
      Node:
        begin
          StringToRNum(gDlogRec.SimStrArr[Npd], theElement^.Pd);

```

```

    ih := GetIcon(NODE_ICON)
  end;
Hub:
  begin
    StringToRNum(gDlogRec.SimStrArr[Hpd], theElement^.Pd);
    ih := GetIcon(HUB_ICON)
  end;
Bypass:
  begin
    StringToRNum(gDlogRec.SimStrArr[Bpd], theElement^.Pd);
    ih := GetIcon(BYP_ICON)
  end;
end;
theElement^.ID := gNumNodes + 1;
theElement^.Alive := TRUE;
for i := 0 to MAX_ITEMS do
  theElement^.LList[i] := nil;
thePoint := theElement^.Loc.topLeft;
thePoint.h := thePoint.h - gOrigin.h;
thePoint.v := thePoint.v - gOrigin.v;
If findElem(thePoint) <> 0 then
  erro := TRUE;
thePoint := theElement^.Loc.botRight;
thePoint.h := thePoint.h - gOrigin.h;
thePoint.v := thePoint.v - gOrigin.v;
If findElem(thePoint) <> 0 then
  erro := TRUE;
thePoint.h := theElement^.Loc.right - gOrigin.h;
thePoint.v := theElement^.Loc.top - gOrigin.v;
If FindElem(thePoint) <> 0 then
  erro := TRUE;
thePoint.h := theElement^.Loc.Left - gOrigin.h;
thePoint.v := theElement^.Loc.Bottom - gOrigin.v;
If FindElem(thePoint) <> 0 then
  erro := TRUE;
If thePoint.h < (gPadWindow^.portRect.left + TOOL_SIZE) then
  erro := TRUE;
If theElement^.Loc.top < (gPadWindow^.portRect.top) then
  erro := TRUE;
If (gNumNodes < MAX_ITEMS) and (not erro) then
  begin
    gNumNodes := gNumNodes + 1;
    gNodeArr[gNumNodes] := theElement;
    SetScreen;
    PlotIcon(theElement^.Loc, ih);
    ResetScreen
  end
else
  dispose(theElement)
end
end;
{/..... FindLink ...../}

```

```

function findLink (thisNode, thatNode: ElementPtr): LinkPtr;
var
  i: integer;
begin
  i := 1;
  findLink := nil;
  while thisNode^.LList[i] <> nil do
    begin
      If thisNode^.LList[i]^End2 = thatNode then
        findLink := thisNode^.LList[i];
        i := i + 1
      end
    end;
end;

```

```

{ /***** ElimLink *****/ }

```

```

procedure ElimLink (theLink: LinkPtr);
var
  i: integer;
  Flag: boolean;
  thisLink: LinkPtr;
begin
  i := 1;
  Flag := FALSE;
  while i <= gNumLinks do
    begin
      If ((not Flag) and (gLinkArr[i] = theLink)) then
        begin
          thisLink := gLinkArr[i];
          Flag := TRUE
        end;
      If Flag then
        gLinkArr[i] := gLinkArr[i + 1];
        i := i + 1
      end;
      If flag then
        Dispose(theLink);
        gNumLinks := gNumLinks - 1
    end;
end;

```

```

{ /***** DetachLink *****/ }

```

```

procedure DetachLink (theLink: LinkPtr; theNode: ElementPtr);
var
  i: integer;
  flag: boolean;
begin
  i := 1;
  flag := FALSE;
  while theNode^.LList[i] <> nil do
    begin
      If theNode^.LList[i] = theLink then
        Flag := TRUE;
      If flag then

```

```

    theNode^.LList[i] := theNode^.LList[i + 1];
    i := i + 1
  end
end;

{***** EraseLink *****/}
procedure EraseLink (theLink: LinkPtr; thisNode, thatNode: ElementPtr);
var
  ih: Handle;
begin
  SetScreen;
  PenPat(white);
  DrawArrows(theLink);
  MoveTo(theLink^.HookPt1.h, theLink^.HookPt1.v);
  LineTo(theLink^.HookPt2.h, theLink^.HookPt2.v);
  ih := GetIcon(NODE_ICON + ord(thisNode^.kind) - ord(Node));
  PlotIcon(thisNode^.Loc, ih);
  ih := GetIcon(NODE_ICON + ord(thatNode^.kind) - ord(Node));
  PlotIcon(thatNode^.Loc, ih);
  ResetScreen;
  PenNormal
end;

{***** CutElem *****/}
procedure CutElem (link: boolean; ID1, ID2: integer);
var
  pt1, pt2: Point;
  i, j, k: integer;
  anLink: LinkPtr;
begin
  If link then
    begin
      anLink := nil;
      anLink := findLink(gNodeArr[ID1], gNodeArr[ID2]);
      If anLink = nil then
        anLink := findLink(gNodeArr[ID2], gNodeArr[ID1]);
      If anLink <> nil then
        begin
          DetachLink(anLink, gNodeArr[ID1]);
          DetachLink(anLink, gNodeArr[ID2]);
          EraseLink(anLink, gNodeArr[ID1], gNodeArr[ID2]);
          ElimLink(anLink)
        end
      end
    else
      begin
        If ID1 < 0 then
          begin
            for i := 1 to gNumNodes do
              begin
                j := 0;
                If i < ID1 then

```

```

        j := NumLinks(i, ID1);
        for k := 1 to j do
            CutElem(ALINK, ID1, i)
        end;
        SetScreen;
        EraseRect(gNodeArr[ID1]^Loc);
        ResetScreen;
        Dispose(gNodeArr[ID1]);
        for i := ID1 to gNumNodes + 1 do
            begin
                gNodeArr[i] := gNodeArr[i + 1];
                If gNodeArr[i] <> nil then
                    gNodeArr[i]^ID := i
                end;
            end;
        gNumNodes := gNumNodes - 1
    end
end
end;

{r . . . . . HandleFileChoice . . . . . }

procedure HandleFileChoice (theltem: integer);
var
    reply: SFReply;           { used in all SF samples }
    fptr: AFilePtr;
    f: ArchFile;
    r: ArchRec;
    n: ElementPtr;
    l: LinkPtr;
    k: ElemType;
    p: Point;
    typeList: SFTYPEList;     { typelist for all SF samples }
    nl, nn: longint;
    i: integer;
begin
    case (theltem) of
        LOAD_ITEM:
            begin
                SFGGetFile(Point($00400040), 'Space for Rent', nil, -1, typeList, nil, reply);
                {location}
                {vestigial string}
                {fileFilter}
                {numtypes; -1 means all}
                {array to types to show}
                {dlgHook}
                {record for returned values}

                If reply.good then
                    begin
                        Reset(f, concat(PathnameFromWD(reply.vRefNum), reply.fName));
                        r := f^;
                    end;
            end;
    end;
    {get(f);}
    Close(f);
    nn := gNumNodes;

```

```

for i := nn downto 1 do
  CutElem(NOT_ALINK, i, 0);    {destroy old arch}
gOrigin.h := 0;
gOrigin.v := 0;
SetCtlValue(gHScroll, 0);
SetCtlValue(gVScroll, 0);

nn := r.NumNodes;
nl := r.NumLinks;
gDlogRec := r.DlogRec;
RestoreSettings(gSimDlogPtr);
RestoreSettings(gOutDlogPtr);

for i := 1 to nn do
  begin
    p := r.NodeArr[i].Loc.topLeft;
    p.h := p.h + 8;
    p.v := p.v + 8;
    AddElem(r.NodeArr[i].Kind, p);
    gNodeArr[i]^Pd := r.NodeArr[i].Pd;
  end;
for i := 1 to nl do
  begin
    k := r.LinkArr[i].kind;
    p := gNodeArr[r.LinkArr[i].End1ID]^Loc.topLeft;
    AddElem(k, p);
    p := gNodeArr[r.LinkArr[i].End2ID]^Loc.topLeft;
    AddElem(k, p);
    gLinkArr[i]^Pd := r.LinkArr[i].Pd;
  end;

  UpdateWindow(gPadWindow)
end
else
  SysBeep(1)

end;
SAVE_ITEM:
begin
  SFPutFile(Point($00400040), 'Save Topology as:', 'untitled.arch', nil, reply);  {location}
{prompt string}
      (original name)
      (dlgHook)
      {record for returned values}
If reply.good then
  begin
    for i := 1 to gNumNodes do
      begin
        r.NodeArr[i].Kind := gNodeArr[i]^Kind;
        r.NodeArr[i].Loc := gNodeArr[i]^Loc;
        r.NodeArr[i].Pd := gNodeArr[i]^Pd;
      end;

```

```

    for i := 1 to gNumLinks do
    begin
        r.LinkArr[i].Kind := gLinkArr[i]^Kind;
        r.LinkArr[i].End1ID := gLinkArr[i]^End1ID;
        r.LinkArr[i].End2ID := gLinkArr[i]^End2ID;
        r.LinkArr[i].Pd := gLinkArr[i]^Pd;
    end;
    r.NumNodes := gNumNodes;
    r.NumLinks := gNumLinks;
    r.DlogRec := gDlogRec;
    f^ := r;
    rewrite(f, concat(PathnameFromWD(reply.vRefNum), reply.fName));
    put(f);
    close(f)
end
else
    SysBeep(1)
end;
QUIT_ITEM:
gDone := TRUE;
end
end;

{ /***** Update Arch *****/}
procedure UpdateArch;
var
    i: integer;
begin
    if gDlogRec.SimRadArr[Typ] then
    begin
        for i := 1 to gNumNodes do
            case gNodeArr[i]^kind of
                Node:
                    StringToRNum(gDlogRec.SimStrArr[Npd], gNodeArr[i]^Pd);
                Hub:
                    StringToRNum(gDlogRec.SimStrArr[Hpd], gNodeArr[i]^Pd);
                Bypass:
                    StringToRNum(gDlogRec.SimStrArr[Bpd], gNodeArr[i]^Pd)
            end;
        for i := 1 to gNumLinks do
            case gLinkArr[i]^kind of
                Dplx:
                    StringToRNum(gDlogRec.SimStrArr[Dpd], gLinkArr[i]^Pd);
                Splx:
                    StringToRNum(gDlogRec.SimStrArr[Spd], gLinkArr[i]^Pd)
            end
        end
    end
end;

{ /***** Bypass Set *****/}
function BypassSet (var nset: NodeSet): NodeSet;
var
    i: integer;

```

```

theSet: NodeSet;
begin
theSet := [];
nset := [];
for i := 1 to gNumNodes do
  If gNodeArr[i]^kind = Bypass then
    theSet := theSet + [gNodeArr[i]^ID]
  else
    nset := nset + [gNodeArr[i]^ID];
  BypassSet := theSet
end;

{ /..... Start ..... / }
function Start (FrItem, RngItem: DlogItem): integer;
var
  r: real;
begin
  with gDlogRec do
    begin
      If SimRadArr[RngItem] then
        begin
          StringToRNum(SimStrArr[FrItem], r);
          If r = 0 then
            Start := 0
          else
            Start := 1
          end
        end
      else
        Start := 1
      end
    end
end;

{ /..... Fin ..... / }
function Fin (FrItem, RngItem: DlogItem): integer;
var
  i: integer;
  r: real;
begin
  with gDlogRec do
    begin
      If SimRadArr[RngItem] then
        begin
          StringToRNum(SimStrArr[succ(FrItem)], r);
          i := round(r * 1000);
          StringToRNum(SimStrArr[FrItem], r);
          i := i - round(r * 1000);
          StringToRNum(SimStrArr[succ(succ(FrItem))], r);
          If r <> 0 then
            i := i div (round(r * 1000))
          else
            i := 0
          end
        end
      else
        i := 0
      end
    end
end;

```

```

    i := 1;
    Fin := i
  end
end;

{ /..... StepSize ..... / }
function StepSize (FrItem, RngItem, PdItem: DialogItemType): real;
var
  r: real;
begin
  with gDialogRec do
    begin
      If SimRadArr[RngItem] then
        StringToRNum(SimStrArr[succ(succ(FrItem))], r)
      else
        StringToRNum(SimStrArr[PdItem], r);
      StepSize := r
    end
  end;
end;

{ /..... WriteHeader ..... / }
procedure WriteHeader (f: TFilePtr);
begin
  with gDialogRec do
    begin
      rewrite(f^, concat(PathnameFromDirID(gCurDirStore^, -(gSFSSaveDisk^)), OutFileStr));
      writeln(f^, '');
      write(f^, 'NPd');
      If SimRadArr[Hr] then
        write(f^, chr(9), 'HPd');
      If SimRadArr[Br] then
        write(f^, chr(9), 'BPd');
      If SimRadArr[Sr] then
        write(f^, chr(9), 'SPd');
      If SimRadArr[Dr] then
        write(f^, chr(9), 'DPd');
      If OutChkArr[Pseg] then
        write(f^, chr(9), 'Pseg');
      If OutChkArr[Avsl] then
        write(f^, chr(9), 'Avsl');
      If OutChkArr[Ecp] then
        write(f^, chr(9), 'Ecp');
      If OutChkArr[Emsl] then
        write(f^, chr(9), 'Emsl');
      writeln(f^);

    end
  end;
end;

{ /..... AdjustDialogRec ..... / }
procedure AdjustDialogRec (Np, Hp, Bp, Sp, Dp: real);
begin
  with gDialogRec do

```

```

begin
  RNumToString(Np, SimStrArr[Npd]);
  RNumToString(Hp, SimStrArr[Hpd]);
  RNumToString(Bp, SimStrArr[Bpd]);
  RNumToString(Sp, SimStrArr[Spd]);
  RNumToString(Dp, SimStrArr[Dpd]);
end
end;

{ / . . . . . HandleAnalChoice . . . . . / }
procedure HandleAnalChoice (theItem: integer);
var
  f: text;
  n, h, b, s, d, l, max, j, k, Item: integer;
  dItem: DialogItemType;
  i, Norm: longint;
  ItemType: integer;
  itemHandle: Handle;
  theRect, itemRect: rect;
  dun, Turbo: Boolean;
  myString1, myString2: Str255;
  Ns, Hs, Bs, Ss, Ds, Psg: real;
begin
  dun := FALSE;
  case theItem of
    SIM_ITEM:
      begin
        BringToFront(gSimDlogPtr);
        NumToString(gNumNodes + gNumLinks, myString1);
        GetDItem(gSimDlogPtr, 24, ItemType, itemHandle, itemRect);
        SetIText(itemHandle, myString1);
        ShowWindow(gSimDlogPtr);
        while not dun do
          begin
            ModalDialog(nil, Item);
            case Item of
              1:
                begin
                  HideWindow(gSimDlogPtr);
                  SaveSettings(gSimDlogPtr);
                  UpdateArch;
                  dun := TRUE
                end;
              2:
                begin
                  HideWindow(gSimDlogPtr);
                  RestoreSettings(gSimDlogPtr);
                  dun := TRUE
                end;
              25:
                begin
                  GetDItem(gSimDlogPtr, Item, ItemType, itemHandle, itemRect);

```

```
    If GetCtlValue(ControlHandle(itemHandle)) <> 1 then
    begin
        SetCtlValue(ControlHandle(itemHandle), 1);
        GetDItem(gSimDlogPtr, Item + 1, itemType, itemHandle, itemRect);
        SetCtlValue(ControlHandle(itemHandle), 0);

    end
end;
26:
begin
    GetDItem(gSimDlogPtr, Item, itemType, itemHandle, itemRect);
    If GetCtlValue(ControlHandle(itemHandle)) <> 1 then
    begin
        SetCtlValue(ControlHandle(itemHandle), 1);
        GetDItem(gSimDlogPtr, Item - 1, itemType, itemHandle, itemRect);
        SetCtlValue(ControlHandle(itemHandle), 0);

    end
end;
48..49:
begin
    GetDItem(gSimDlogPtr, Item, itemType, itemHandle, itemRect);
    If GetCtlValue(ControlHandle(itemHandle)) <> 1 then
    begin
        SetCtlValue(ControlHandle(itemHandle), 1);
        GetDItem(gSimDlogPtr, Item + (+1 - (Item mod 48) * 2), itemType, itemHandle, itemRect);
        SetCtlValue(ControlHandle(itemHandle), 0)
    end
end;
50..54:
begin
    GetDItem(gSimDlogPtr, Item, itemType, itemHandle, itemRect);
    If GetCtlValue(ControlHandle(itemHandle)) <> 1 then
    begin
        SetCtlValue(ControlHandle(itemHandle), 1);
        GetDItem(gSimDlogPtr, Item + 5, itemType, itemHandle, itemRect);
        SetCtlValue(ControlHandle(itemHandle), 0)
    end
end;
55..59:
begin
    GetDItem(gSimDlogPtr, Item, itemType, itemHandle, itemRect);
    If GetCtlValue(ControlHandle(itemHandle)) <> 1 then
    begin
        SetCtlValue(ControlHandle(itemHandle), 1);
        GetDItem(gSimDlogPtr, Item - 5, itemType, itemHandle, itemRect);
        SetCtlValue(ControlHandle(itemHandle), 0)
    end
end;
otherwise
end
end
end;
```

OUT_ITEM:

begin

BringToFront(gOutDlogPtr);

ShowWindow(gOutDlogPtr);

while not dun do

begin

ModalDialog(nil, Item);

case Item of

1:

begin

HideWindow(gOutDlogPtr);

SaveSettings(gOutDlogPtr);

dun := TRUE

end;

2:

begin

HideWindow(gOutDlogPtr);

RestoreSettings(gOutDlogPtr);

dun := TRUE

end;

5..8:

begin

GetDItem(gOutDlogPtr, Item, itemType, itemHandle, itemRect);

SetCtlValue(ControlHandle(itemHandle), Bitxor(GetCtlValue(ControlHandle(itemHandle)), 1));

end;

otherwise

end

end

end;

RUN_ITEM:

begin

with gDlogRec do

begin

SetCursor(GetCursor(watchCursor)^^);

gBypSet := BypassSet(gNodeSet);

StringToNum(SimStrArr[Mst], gNumStates);

Turbo := not (OutChkArr[Pseg] or OutChkArr[Avsl] or OutChkArr[Ecp] or OutChkArr[Emsl]);

end;

If Turbo then

begin

gSegCount := 0;

randSeed := TickCount;

TurboLoop;

Pag := gSegCount / gNumStates;

SetRect(theRect, 35, 3, 123, 20);

EraseRect(theRect);

FrameRect(theRect);

MoveTo(37, 15);

RNumToString(Pag, myString1);

DrawString(concat('Pseg=', myString1));

ShowCursor

end

else

```

begin
WriteHeader(@f);
Ns := StepSize(Nfr, Nr, Npd);
Hs := StepSize(Hfr, Hr, Hpd);
Bs := StepSize(Bfr, Br, Bpd);
Ss := StepSize(Sfr, Sr, Spd);
Ds := StepSize(Dfr, Dr, Dpd);
for n := Start(Nfr, Nr) to Fin(Nfr, Nr) do
  for h := Start(Hfr, Hr) to Fin(Hfr, Hr) do
    for b := Start(Bfr, Br) to Fin(Bfr, Br) do
      for s := Start(Sfr, Sr) to Fin(Sfr, Sr) do
        for d := Start(Dfr, Dr) to Fin(Dfr, Dr) do
          begin
            AdjustDlgRec(n * Ns, h * Hs, b * Bs, s * Ss, d * Ds);
            UpdateArch;
            gSegCount := 0;
            gAvsl := 0;
            gEcp := 0;
            gEmsl := 0;
            gTotalSegs := 0;
            max := 0;
            randSeed := TickCount;
            for i := 1 to gNumNodes do
              begin
                gSegLnthArr[i] := 0;
                gMaxSegArr[i] := 0
              end;
            (for i to gNumNodes)
            for i := 1 to gNumStates do
              begin
                GetMonteState;
                Analyze;

                If gNumSegs > 1 then
                  gSegCount := gSegCount + 1;
                  max := 0;
                  for j := 1 to gNumSegs do
                    begin
                      l := SetSize(gSegArr[j]);
                      gSegLnthArr[l] := gSegLnthArr[l] + 1;
                      If l > max then
                        max := l
                    end;
                  (for j)
                  gTotalSegs := gTotalSegs + gNumSegs;
                  gMaxSegArr[max] := gMaxSegArr[max] + 1
                end;
              (for i to gNumStates)

            Pag := gSegCount / gNumStates;
            for i := 1 to gNumNodes do
              begin
                gAvsl := gAvsl + i * gSegLnthArr[i];

```

```

    gEcp := gEcp + Binomial(i) * gSegLnthArr[i];
    gEmsl := gEmsl + i * gMaxSegArr[i]
  end;
  {for}

  Norm := (gTotalSegs * SetSize(gNodeSet));
  If Norm <> 0 then
    gAvsl := gAvsl / Norm
  else
    gAvsl := 0;
  Norm := (gTotalSegs * Binomial(SetSize(gNodeSet)));
  If Norm <> 0 then
    gEcp := gEcp / Norm
  else
    gEcp := 0;
  Norm := (gNumStates * SetSize(gNodeSet));
  If Norm <> 0 then
    gEmsl := gEmsl / Norm
  else
    gEmsl := 0;

  with gDlogRec do
  begin
    write(f, n * Ns);
    If SimRadArr[Hr] then
      write(f, chr(9), (h * Hs) : 6 : 3);
    If SimRadArr[Br] then
      write(f, chr(9), (b * Bs) : 6 : 3);
    If SimRadArr[Sr] then
      write(f, chr(9), (s * Ss) : 6 : 3);
    If SimRadArr[Dr] then
      write(f, chr(9), (d * Ds) : 6 : 3);
    If OutChkArr[Pseg] then
      write(f, chr(9), (Psg) : 6 : 3);
    If OutChkArr[Avsl] then
      write(f, chr(9), (gAvsl) : 6 : 3);
    If OutChkArr[Ecp] then
      write(f, chr(9), (gEcp) : 6 : 3);
    If OutChkArr[Emsl] then
      write(f, chr(9), (gEmsl) : 6 : 3);
    writeln(f)
  end;
  {with}

  SetRect(theRect, 35, 3, 123, 20);
  EraseRect(theRect);
  FrameRect(theRect);
  MoveTo(37, 15);
  RNumToString(Psg, myString1);
  DrawString(concat("Pseg=", myString1));
  ShowCursor;
end;
{multi for loop}

close(f)
end;
{else}
SetCursor(arrow)

```

```

    end
  end
end;
(RUN)
(case)
(procedure)

```

```

{ / . . . . . HandleMenuChoice . . . . . }

```

```

procedure HandleMenuChoice (menuChoice: longint);
var
  i, nn, theMenu, theItem: integer;
begin
  if (menuChoice <> 0) then
    begin
      theMenu := HiWord(menuChoice);
      theItem := LoWord(menuChoice);
      case (theMenu) of
        APPLE_MENU_ID:
          HandleAppleChoice(theItem);
        FILE_MENU_ID:
          HandleFileChoice(theItem);
        EDIT_MENU_ID:
          begin
            case theItem of
              CLEAR_ITEM:
                begin
                  nn := gNumNodes;
                  for i := nn downto 1 do
                    CutElem(NOT_ALINK, i, 0);    (destroy old arch)
                  gOrigin.h := 0;
                  gOrigin.v := 0;
                  SetCtlValue(gHScroll, 0);
                  SetCtlValue(gVScroll, 0);
                end;
            end;
          end;
        ANAL_MENU_ID:
          HandleAnalChoice(theItem);
      end;
      HighlightMenu(0)
    end
  end;
end;

```

```

{ / . . . . . Scroll . . . . . }

```

```

procedure Scroll;
var
  oldOrigin: Point;
  dh, dv: integer;
  tRect: Rect;
begin

```

```

oldOrigin := gOrigin;
gOrigin.h := 1 * GetCtlValue(gHScroll);
gOrigin.v := 1 * GetCtlValue(gVScroll);
dh := oldOrigin.h - gOrigin.h;
dv := oldOrigin.v - gOrigin.v;
gUpdateRgn := NewRgn;
tRect := gPadWindow^.portRect;
tRect.right := tRect.right - SBarWidth;
tRect.bottom := tRect.bottom - SBarWidth;
tRect.left := tRect.left + TOOL_SIZE + 2;
ScrollRect(tRect, dh, dv, gUpdateRgn);

```

```

(/ * Have scrolled in junk .. need to redraw * /)

```

```

SetOrigin(gOrigin.h, gOrigin.v);
OffsetRect(gUpdateRgn^.rgnBBox, gOrigin.h, gOrigin.v);
ClipRect(gUpdateRgn^.rgnBBox);
DrawCont;
DisposeRgn(gUpdateRgn);
SetOrigin(0, 0);
ClipRect(screenBits.bounds)
end;

```

```

(/ * * * * * scrollbarproc * * * * * /)

```

```

procedure ScrollProc (theControl: ControlHandle; theCode: integer);
var
  pageSize: integer;
  scrollAmt: integer;
begin
  if (theCode <> 0) then
    begin
      if (theControl = gVScroll) then
        pageSize := (gPadWindow^.portRect.bottom - gPadWindow^.portRect.top) div 4
      else
        pageSize := (gPadWindow^.portRect.right - gPadWindow^.portRect.left) div 4;

      case (theCode) of
        inUpButton:
          scrollAmt := -3;
        inDownButton:
          scrollAmt := 3;
        inPageUp:
          scrollAmt := -pageSize;
        inPageDown:
          scrollAmt := pageSize
      end;
      SetCtlValue(theControl, GetCtlValue(theControl) + scrollAmt);
      Scroll
    end
  end;
end;

```

```

{ / . . . . . HandleControl . . . . . /
  procedure HandleControl (cntrl: ControlHandle; part: integer; pnt: Point);

```

```

  begin

```

```

    if ((cntrl = gVScroll) or (cntrl = gHScroll)) then

```

```

      if (part = inThumb) then

```

```

        begin

```

```

          part := TrackControl(cntrl, pnt, nil);

```

```

          Scroll

```

```

        end

```

```

      else

```

```

        part := TrackControl(cntrl, pnt, @ScrollProc)

```

```

    end;

```

```

{ / . . . . . HandleScreen . . . . . /
  procedure HandleScreen (w: WindowPtr; p: Point);

```

```

    var

```

```

      thePoint: Point;

```

```

      theRect: Rect;

```

```

      thePart, ElemID1, ElemID2: integer;

```

```

      theControl: ControlHandle;

```

```

      savePort: GrafPtr;

```

```

      theElement: ElementPtr;

```

```

  begin

```

```

    GetPort(savePort);

```

```

    SetPort(w);

```

```

    thePoint := p;

```

```

    GlobalToLocal(thePoint);

```

```

    thePart := FindControl(thePoint, w, theControl);

```

```

    ElemID1 := 0;

```

```

    ElemID2 := 0;

```

```

    theRect := gPadWindow^.portRect;

```

```

    theRect.right := theRect.left + TOOL_SIZE;

```

```

  if (thePart = 0) then

```

```

    begin

```

```

      if (PtInRect(thePoint, gTMap[CutNode])) then

```

```

        begin

```

```

          HiliTool(CutNode)

```

```

        end

```

```

      else if (PtInRect(thePoint, gTMap[CutLink])) then

```

```

        begin

```

```

          HiliTool(CutLink)

```

```

        end

```

```

      else if (PtInRect(thePoint, gTMap[AddDplx])) then

```

```

        begin

```

```

    HiliTool(AddDplx)
  end
else if (PtInRect(thePoint, gTMap[AddSplx])) then
  begin
    HiliTool(AddSplx)
  end
else if (PtInRect(thePoint, gTMap[AddByp])) then
  begin
    HiliTool(AddByp)
  end
else if (PtInRect(thePoint, gTMap[AddHub])) then
  begin
    HiliTool(AddHub)
  end
else if (PtInRect(thePoint, gTMap[AddNode])) then
  begin
    HiliTool(AddNode)
  end
else if PtInRect(thePoint, theRect) then
  begin
    HiliTool(NullCom);
    gFirstClick := True;
    gFirstID := 0
  end
else
  begin
    gWhenFirstClick := 0;
    case gMode of
      NullCom:
        begin
          ElemID1 := FindElem(thePoint);
          if ElemID1 <> 0 then
            PopElemDLOG(ElemID1)
          else
            begin
              ElemID1 := GetLinkNum(thePoint);
              if ElemID1 <> 0 then
                PopLinkDLOG(ElemID1)
            end
          end;
        end;
      AddNode:
        begin
          AddElem(Node, thePoint);
          if gTState <> locked then
            HiliTool(AddNode);
          end;
        end;
      AddHub:
        begin
          AddElem(Hub, thePoint);
          if gTState <> locked then
            HiliTool(AddHub);
          end;
        end;
      AddByp:

```

```
begin
  AddElem(Bypass, thePoint);
  If gTState <> locked then
    HiliTool(AddByp);
  end;
AddSplx:
begin
  AddElem(Splx, thePoint);
  If ((gTState <> locked) and gFirstClick) then
    HiliTool(AddSplx);
  end;
AddDplx:
begin
  AddElem(Dplx, thePoint);
  If ((gTState <> locked) and gFirstClick) then
    HiliTool(AddDplx);
  end;
CutNode:
begin
  ElemID1 := FindElem(thePoint);
  If ElemID1 <> 0 then
    CutElem(NOT_ALINK, ElemID1, ElemID2);
  If gTState <> locked then
    HiliTool(CutNode)
  end;
CutLink:
begin
  If gFirstClick then
    begin
      gFirstID := FindElem(thePoint);
      If gFirstID <> 0 then
        gFirstClick := FALSE
      end
    else
      begin
        ElemID2 := FindElem(thePoint);
        If ElemID2 <> 0 then
          begin
            If ElemID2 = gFirstID then
              begin
                gFirstID := 0;
                gFirstClick := True
              end
            else
              begin
                CutElem(ALINK, gFirstID, ElemID2);
                gFirstClick := True
              end
            end;
          end;
        end;
      If ((gTState <> locked) and gFirstClick) then
        HiliTool(CutLink);
      end
    end
```

```

        end (case)
    end
end
else
    HandleControl(theControl, thePart, thePoint);

    SetPort(savePort)

end;

{/ ..... HandleMouseDown ..... /}

procedure HandleMouseDown;
var
    whichWindow: WindowPtr;
    thePart: Integer;
    menuChoice, windSize: longint;
    thePoint: Point;
begin
    thePart := FindWindow(gTheEvent.where, whichWindow);
    case (thePart) of
        inDesk:
            SysBeep(1);

        inMenuBar:
            begin
                menuChoice := MenuSelect(gTheEvent.where);
                HandleMenuChoice(menuChoice)
            end;

        inSysWindow:
            SystemClick(gTheEvent, whichWindow);

        inDrag:
            DragWindow(whichWindow, gTheEvent.where, gDragRect);

        inGoAway:
            If (TrackGoAway(whichWindow, gTheEvent.where)) then
                gDone := TRUE;

        inGrow:
            Grow(whichWindow, gTheEvent.where);

        inContent:
            If whichWindow <> frontwindow then
                SelectWindow(whichW. dow)
            else
                HandleScreen(whichWindow, gTheEvent.where)
            end
    end;
end;

```

```

{ /***** Handle Activ*****/ }
procedure HandleActivate (w: WindowPtr; goingActive: Boolean);
begin
  if w = gPadWindow then
    begin
      SetPort(w);
      DrawGrowIcon(w);
    end;
  if goingActive then
    begin
      ShowControl(gHScroll);
      ShowControl(gVScroll)
    end
  else
    begin
      HideControl(gHScroll);
      HideControl(gVScroll)
    end
  end;
end;

{ /***** Handle Event*****/ }
procedure HandleEvent;
var
  aPoint: Point;
  err: integer;
begin
  case (gTheEvent.what) of

    nullEvent:
      ;
      (* HandleNull ;*)

    mouseDown:
      begin
        HandleMouseDown
      end;
    autoKey:
      ;
    keyDown:
      begin
        gTheEvent.Message := BitAnd(gTheEvent.Message, charCodeMask);
        if BitAnd(gTheEvent.Modifiers, CmdKey) = CmdKey then
          begin
            HandleMenuChoice(MenuKey(Chr(gTheEvent.Message)))
          end
        end;
      end;
    updateEvt:
      if (WindowPtr(gTheEvent.message) = gPadWindow) then
        UpdateWindow(gPadWindow)
      else
        begin

```

```

    BeginUpdate(WindowPtr(gTheEvent.Message));
    EndUpdate(WindowPtr(gTheEvent.Message))
end;

```

```
activateEvt:
```

```
begin
```

```
    HandleActivate(gPadWindow, BAND(gTheEvent.Modifiers, activeFlag) = 1)
```

```
end;
```

```
diskEvt:
```

```
begin
```

```
    If HiWrd(gTheEvent.message) <> noErr then
```

```
        begin
```

```
            SetPt(aPoint, DI_LEFT, DI_TOP);
```

```
            err := DIBadMount(aPoint, gTheEvent.Message)
```

```
        end
```

```
    end;
```

```
app4Evt:
```

```
begin
```

```
    case BAND(BROTL(gTheEvent.message, 8), $FF) of
```

```
        SUSP_RES_MESS:
```

```
            begin
```

```
                glnBackground := BAND(gTheEvent.Message, RES_MASK) = 0;
```

```
                HandleActivate(gPadWindow, not glnBackground)
```

```
            end;
```

```
        otherwise
```

```
    end
```

```
end;
```

```
otherwise
```

```
end
```

```
end;
```

```
{/..... GetGlobMowse ...../}
```

```
procedure GetGlobMowse (var pt: point);
```

```
var
```

```
    event: EventRecord;
```

```
begin
```

```
{if OSEventAvail(0, event) then}
```

```
{pt := event.where}
```

```
{else}
```

```
{pt := event.where}
```

```
    GetMouse(pt);
```

```
    LocalToGlobal(pt)
```

```
end;
```

```
{/..... AdjustCurs...../}
```

```
procedure AdjustCurs (pt: point; var crgn: RgnHandle);
```

```
var
```

```
    window: WindowPtr;
```

```
    arrowRgn, activeRgn: RgnHandle;
```

```
    pRect: Rect;
```

```
    CursID: Integer;
```

```
begin
```

```
    window := FrontWindow;
```

```

If (not gInBackground) and (not IsDAWindow(window)) then
begin

```

```

  arrowRgn := NewRgn;

```

```

  activeRgn := NewRgn;

```

```

  SetRectRgn(arrowRgn, MAX_NEG, MAX_NEG, MAX_POS, MAX_POS);

```

```

  If ((gMode <> NullCom)) then

```

```

  begin

```

```

    If IsAppWindow(window) then

```

```

    begin

```

```

      SetPort(window);

```

```

      SetOrigin(-window^.portBits.bounds.left, -window^.portBits.bounds.top);

```

```

      pRect := window^.portRect;

```

```

      pRect.left := pRect.left + TOOL_SIZE;

```

```

      pRect.right := pRect.right - SBarWidth;

```

```

      pRect.bottom := pRect.bottom - SBarWidth;

```

```

      RectRgn(activeRgn, pRect);

```

```

      SectRgn(activeRgn, window^.visRgn, activeRgn);

```

```

      case gMode of

```

```

        AddNode..AddByp:

```

```

          CursID := ELEM_CURS;

```

```

        CutNode:

```

```

          CursID := CUT_NODE_CURS;

```

```

        AddSplx:

```

```

          CursID := ADD_SLNK_CURS;

```

```

        AddDplx:

```

```

          CursID := ADD_DLNK_CURS;

```

```

        CutLink:

```

```

          CursID := CUT_LINK_CURS;

```

```

        otherwise

```

```

      end;

```

```

      SetOrigin(0, 0)

```

```

    end

```

```

  end;

```

```

  DiffRgn(arrowRgn, activeRgn, arrowRgn);

```

```

  If PtInRgn(pt, activeRgn) then

```

```

  begin

```

```

    SetCursor(GetCursor(CursID)^);

```

```

    CopyRgn(activeRgn, crgn)

```

```

  end

```

```

  else

```

```

  begin

```

```

    SetCursor(arrow);

```

```

    CopyRgn(arrowRgn, crgn)

```

```

  end;

```

```

  DisposeRgn(arrowRgn);

```

```

  DisposeRgn(activeRgn)

```

```

end

```

```

end;

```

```

{ ..... MainLoop ..... }

```

```

procedure MainLoop;

```

```

var

```

```

    gotEvent: boolean;
    cursRgn: RgnHandle;
    mowse: point;
begin
    mowse.h := 0;
    mowse.v := 0;
    cursRgn := NewRgn;
    FlushEvents(everyEvent, 0);
    gWNEImplemented := (NGetTrapAddress(WNE_TRAP_NUM, TooITrap) <>
        NGetTrapAddress(UNIMPL_TRAP_NUM, TooITrap));
    while (gDone = FALSE) do
        begin
            If (gWNEImplemented) then
                begin
                    GetGlobMowse(mowse);
                    AdjustCurs(mowse, cursRgn);

                    gotEvent := WaitNextEvent(everyEvent, gTheEvent, MAXLONGINT, cursRgn)
                end
            else
                begin
                    SystemTask;
                    gotEvent := GetNextEvent(everyEvent, gTheEvent)
                end;
            If gotEvent then
                begin
                    AdjustCurs(gTheEvent.where, cursRgn);
                    HandleEvent
                end
            end
        end;
end;

{ /***** MiscInit *****/}
procedure MiscInit;
var
    i: integer;
begin
    gDone := FALSE;
    gWhenFirstClick := 0;
    gInBackground := FALSE;
    gMode := NullCom;
    for i := 0 to MAX_ITEMS do
        begin
            gNodeArr[i] := nil;
            gLinkArr[i] := nil;
            gNilLinkArr[i] := nil
        end;
    gTState := off;
    gFirstClick := TRUE;
    gFirstID := 0;
    gNumNodes := 0;
    GNumLinks := 0;
    SetCursor(arrow);

```

```

gLinkSet := {Splx, Dplx};
gLComSet := {AddDplx, AddSplx, CutLink};
randSeed := TickCount;
gSegCount := 0;
i := SysEnvirons(1, gWorld);
gSFSSaveDisk := PtrToWord(kSFSSaveDisk);
gCurDirStore := PtrToLong(kCurDirStore)
end;

```

```

{.....DialogInit .....}
procedure DialogInit;
var
  i: DialogItemType;
  itemType: integer;
  itemRect: rect;
  itemHandle: Handle;
begin
  with gDlogRec do
    begin
      gSimDlogPtr := GetNewDialog(SIM_DLOG, nil, POINTER(MOVE_TC_FRONT));
      gOutDlogPtr := GetNewDialog(OUT_DLOG, nil, POINTER(MOVE_TO_FRONT));
      gElemDlogPtr := GetNewDialog(ELEM_DLOG, nil, POINTER(MOVE_TO_FRONT));
      gLinkDlogPtr := GetNewDialog(LINK_DLOG, nil, POINTER(MOVE_TO_FRONT));
      for i := Npd to Emsl do
        case i of
          Mon:
            SimRadArr[i] := TRUE;
          Exh:
            SimRadArr[i] := FALSE;
          Nc..Dc:
            SimRadArr[i] := TRUE;
          Nr..Dr:
            SimRadArr[i] := FALSE;
          Ind:
            SimRadArr[i] := FALSE;
          Typ:
            SimRadArr[i] := TRUE;
          Npd..Bpd:
            RNumToString(DEF_NODE_PD, SimStrArr[i]);
          Spd, Dpd:
            RNumToString(DEF_LINK_PD, SimStrArr[i]);
          Nfr..Dst:
            case ((ord(i) - ord(Npd)) mod 3) of
              0:
                RNumToString(DEF_TO_PD, SimStrArr[i]);
              1:
                RNumToString(DEF_STP_SZ, SimStrArr[i]);
              2:
                RNumToString(DEF_FRM_PD, SimStrArr[i]);
            end;
          Mst:
            NumToString(DEF_STATES, SimStrArr[i]);
          Tst:

```

```

    NumToString((gNumLinks + gNumNodes), SimStrArr[i]);
Pseg..Emsl:
    If i = Pseg then
        OutChkArr[i] := TRUE
    else
        OutChkArr[i] := FALSE;
    otherwise
end;
OutFileStr := 'untitled 1';
end;
RestoreSettings(gSimDlogPtr);
RestoreSettings(gOutDlogPtr);
GetDItem(gSimDlogPtr, 49, ItemType, itemHandle, itemRect);
HiliteControl(ControlHandle(itemHandle), 255);

end;

{/ ..... WindowInit ..... /}

procedure WindowInit;
var
    padRect, vScrollRect, hScrollRect, viewRect: Rect;
    cRgn: RgnHandle;
begin
    gOrigin.h := 0;
    gOrigin.v := 0;
    gPadWindow := GetNewWindow(BASE_RES_ID, nil, POINTER(MOVE_TO_FRONT));
    SetPort(gPadWindow);
    vScrollRect := gPadWindow^.portRect;
    vScrollRect.left := vScrollRect.right - 15;
    vScrollRect.right := vScrollRect.right + 1;
    vScrollRect.bottom := vScrollRect.bottom - 14;
    vScrollRect.top := vScrollRect.top - 1;
    gVScroll := NewControl(gPadWindow, vScrollRect, 'h', TRUE, 0, 0, PAD_LIMIT, scrollBarProc, 0);
    hScrollRect := gPadWindow^.portRect;
    hScrollRect.right := hScrollRect.right - 14;
    hScrollRect.left := hScrollRect.left - 1;
    hScrollRect.bottom := hScrollRect.bottom + 1;
    hScrollRect.top := hScrollRect.bottom - 16;
    gHScroll := NewControl(gPadWindow, hScrollRect, 'HO', TRUE, 0, 0, PAD_LIMIT, scrollBarProc, 0);
    gDragRect := screenBits.bounds;
    gDragRect.left := gDragRect.left + DRAG_THRESHOLD;
    gDragRect.right := gDragRect.right - DRAG_THRESHOLD;
    gDragRect.bottom := gDragRect.bottom - DRAG_THRESHOLD;
end;

{/ ..... MenuBarInit ..... /}

procedure MenuBarInit;
var
    myMenuBar: Handle;
begin
    myMenuBar := GetNewMBar(BASE_RES_ID);

```

```
SetMenuBar(myMenuBar);  
gAppleMenu := GetMHandle(APPLE_MENU_ID);  
AddResMenu(gAppleMenu, 'DRVR');  
DrawMenuBar  
end;
```

```
begin  
  MiscInit;  
  WindowInit;  
  MenuBarInit;  
  DialogInit;  
  MainLoop  
end.
```

MultiEngine Unit

unit Engine;

Interface

uses

ConsVars;

var

gSegArr: array[0..MAX_ITEMS] of NodeSet;

procedure GetMonteState;

procedure Analyze;

procedure TurboAnalyze;

function SetSize (nset: NodeSet): integer;

procedure TurboLoop;

Implementation

procedure TurboLoop;

var

i: longint;

begin

for i := 1 to gNumStates do

begin

GetMonteState;

TurboAnalyze;

end;

end;

function SetSize (nset: NodeSet): integer;

var

i, j: integer;

begin

j := 0;

for i := 1 to gNumNodes do

if ((i in nset) and (i in gNodeSet)) then

j := j + 1;

SetSize := j

end;

procedure GetMonteState;

var

rnd: real;

i: integer;

begin

for i := 1 to gNumNodes do

begin

rnd := random;

rnd := abs(rnd);

rnd := rnd / 32767.001;

if rnd < gNodeArr[i]^Pd then

gNodeArr[i]^Alive := FALSE

```

    else
      gNodeArr[i]^Alive := TRUE
    end;
  for i := 1 to gNumLinks do
    begin
      rnd := Abs(Random) / 32767.001;
      If rnd < gLinkArr[i]^Pd then
        gLinkArr[i]^Alive := FALSE
      else
        gLinkArr[i]^Alive := TRUE
      end
    end;
  end;
procedure Analyze;
var
  ParentSet, DoneSet, SegSet: NodeSet;
  ind, SegNum: integer;
  ParentArr: SetArrayType;
  flag: boolean;

procedure inspectseg (var pset, lset: NodeSet; thisNode: ElementPtr);
var
  i: integer;
  loopset: NodeSet;
  NextNode: ElementPtr;
begin
  ParentArr[thisNode^.ID] := pset;
  pset := pset + [thisNode^.ID];
  lset := lset + [thisNode^.ID];
  i := 1;
  while (thisNode^.LList[i] <> nil) do
    begin
      If (thisNode^.LList[i]^Alive) then
        begin
          If thisNode = thisNode^.LList[i]^end1 then
            NextNode := thisNode^.LList[i]^end2
          else
            NextNode := thisNode^.LList[i]^end1;
          If NextNode^.Alive then
            begin
              If NextNode^.ID In pset then
                lset := lset + (pset - ParentArr[NextNode^.ID])
              else
                begin
                  If ((thisNode^.LList[i]^kind = Splx)) then
                    begin
                      loopset := [];
                      inspectseg(pset, loopset, NextNode);
                      If lset * loopset = [] then
                        begin
                          If (((loopset - gBypSet) <> []) and ((loopset * DoneSet) = [])) then
                            begin
                              gNumSegs := gNumSegs + 1;
                              gSegArr[gNumSegs] := loopset
                            end
                          end
                        end
                    end
                end
            end
          end
        end
      end
    end
  end

```

```

        end;
        pset := pset - loopset;
        DoneSet := DoneSet + loopset
      end
    else
      lset := lset + loopset
    end
  else
    begin
      inspectseg(pset, lset, NextNode)
    end
  end
end
end;
i := i + 1
end
end;
begin
  Doneset := [];
  ind := 1;
  gNumSegs := 0;
  for ind := 1 to gNumNodes do
    begin
      if (gNodeArr[ind]^Alive and not (gNodeArr[ind]^ID In DoneSet) and (gNodeArr[ind]^kind <> Bypass))
      then {substitute doneset for segset}
      begin
        ParentSet := [];
        SegSet := [];
        SegNum := gNumSegs + 1;
        gNumSegs := SegNum;
        inspectSeg(ParentSet, SegSet, gNodeArr[ind]);
        gSegArr[SegNum] := SegSet;
        DoneSet := DoneSet + SegSet
      end
    end
  end;

  procedure TurboAnalyze;
  label
  1;
  var
    ParentSet, SegSet: NodeSet;
    ind: integer;
    ParentArr: SetArrayType;
    flag: boolean;
  procedure Tinspectseg (var pset, lset: NodeSet; thisNode: ElementPtr);
  var
    i: integer;
    loopset: NodeSet;
    NextNode: ElementPtr;
  begin
    ParentArr[thisNode^.ID] := pset;
    pset := pset + [thisNode^.ID];

```

```

lset := lset + [thisNode^.ID];
i := 1;
while (thisNode^.LList[i] <> nil) do
begin
  if (thisNode^.LList[i]^Alive) then
  begin
    if thisNode = thisNode^.LList[i]^end1 then
      NextNode := thisNode^.LList[i]^end2
    else
      NextNode := thisNode^.LList[i]^end1;
    if NextNode^.Alive then
    begin
      if NextNode^.ID in pset then
        lset := lset + (pset - ParentArr[NextNode^.ID])
      else
        begin
          if ((thisNode^.LList[i]^kind = Splx)) then
          begin
            loopset := [];
            Tinspectseg(pset, loopset, NextNode);
            if lset * loopset = [] then
            begin
              if (loopset - gBypSet) <> [] then
              begin
                goto 1
              end;
              pset := pset - loopset
            end
            else
              lset := lset + loopset
            end
            else
            begin
              Tinspectseg(pset, lset, NextNode);
            end
          end
        end
      end
    end;
    i := i + 1
  end
end;
begin
  SegSet := [];
  flag := FALSE;
  for ind := 1 to gNumNodes do
  begin
    if ((gNodeArr[ind]^Alive) and not (gNodeArr[ind]^ID in SegSet) and (gNodeArr[ind]^kind <>
      Bypass)) then
    begin
      if Flag then
        goto 1;
      ParentSet := [];
      TinspectSeg(ParentSet, SegSet, gNodeArr[ind]);
    end
  end
end

```

```
    Flag := True;
  end
end;
Exit(TurboAnalyze);
1:
  gSegCount := gSegCount + 1
end;

end.
```