Automating Software Documentation

submitted by
Karen Paffendorf
paffy@paffendorf.sj.ca.us

supervised by
Diane L. Souvaine

April 13, 1998
Technical Report dcs-tr-354*
Department of Computer Science
Rutgers University

Abstract

This paper describes automating the documentation of a software control structure. The code is preprocessed to take the form of directed graphs. The following tools are used for drawing the directed graphs: **DaVinci**, **Graphlet**, **Graphviz**, and **VCG**. The tools are described and compared.

1 INTRODUCTION:

During the early 1990's I worked for International Business Machines (IBM) on a hierarchical database, Information Management System (IMS), which was initially developed in the late 1960's. One of my responsibilities was to maintain the logger. The logger maintained copies of the information needed for database recovery in case of various assortments of hardware failure. Jack Howe implemented a system which reused blocks of code in different error scenarios for the logger. Rather than pay the time and space penalty of subroutines or the space penalty of macros, he and Ron Peterson developed a system for transferring control between the blocks of code. He documented this

*http://www.cs.rutgers.edu/pub/technical-reports/dcs-tr-354.ps.Z

system with a series of diagrams, one per scenario. Since updating this documentation was very error prone, automating the generation of this documentation became a goal for some of us who had to maintain this code.

I approached the problem by looking at the diagrams that I wanted to be produced. In a diagram, a block of code was represented by an oval containing the name of the block, and the transfer of control between blocks of code was represented by an arrow with a label describing the error condition that causes the particular transfer to occur. The diagrams are not totally independant from each other. This paper will not discuss the duplicated information which is used to trace the control flow from one diagram to another.

There are many tools available for presenting such information. Most require manually drawing the shapes, writing the text, and laying these objects out. [1] Although these tools make the diagrams look more polished than drawing them by hand, they require at least as much work as doing it manually. This paper is going to explore some tools which will accept textual information describing the content of the diagram and will create the entire diagram including layouts automatically.

When looking at the array of tools available, it becomes apparent that they use a different terminology to describe the diagram. A diagram is called a graph drawing. The oval and the associated label, representing a code block, is called a node. The arrow and its associated label, representing a transition, is called a directed edge. (See Figure 1.) It also becomes apparent that there are different classes of problems with this basic structure. Further examination of the data shows other classifications. In a non-connected graph, there are sections of the graph which do not share edges with other sections of the graph. (See Figure 2.) Additionally, the graphs may be cyclic (See Figure 3.). These graphs may also have multiple sources and multiple sinks, where a source has no incoming edges and a sink has no outgoing edges (See Figure 4). Multiple nodes in a diagram may have the same label (observe $S12$ in Figure 17 on page 25 for an example). See the Section 2.4 for more on how this situation can occur. This multiplicity requires the graph drawing tool to support nodes that have the same label and provide another method for unique identification. None of these diagrams have any self edges (edges that begin and end at the

[1] I used one of these tools, xfig, for drawing some examples.

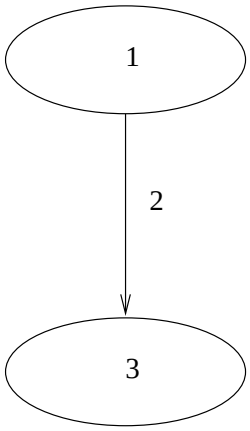

Figure 1: In this graph, 1 and 3 are nodes and 2 is a directed edge.

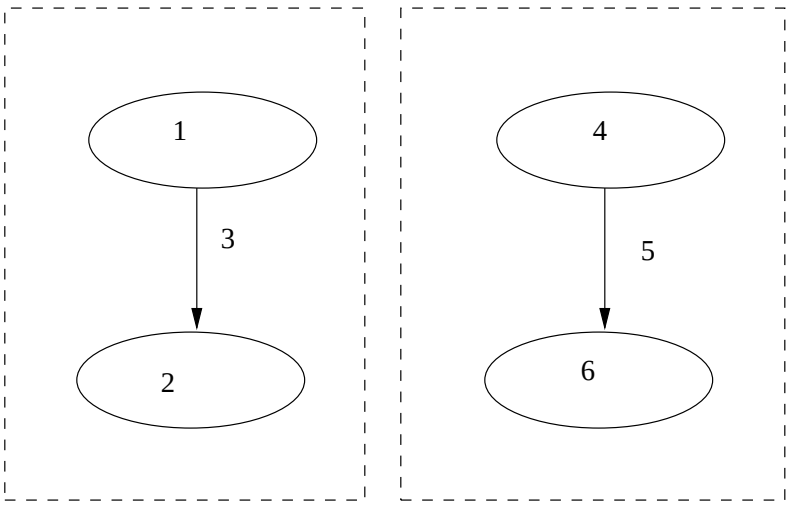

Figure 2: This graph has two non-connected components.

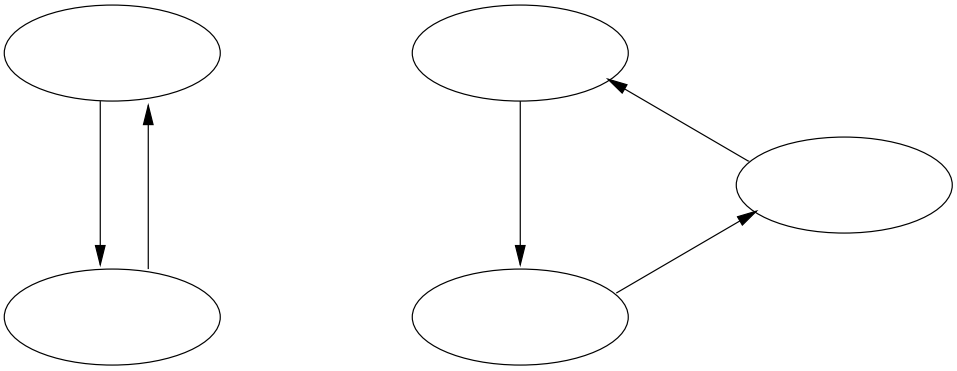

Figure 3: Two (non-connected) cycles in a graph

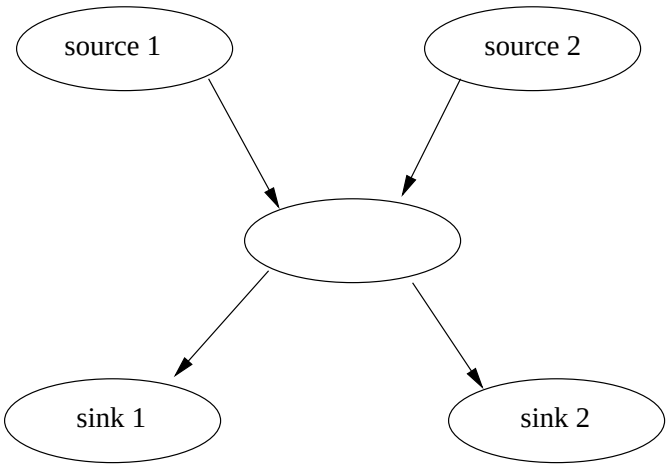

Figure 4: An example with two sources and two sinks

same node) in these diagrams. Each of these classifications narrows down the choice of tools.

Two main strategies are utilized by the current tools: hierarchical layout and spring-embedded layout. A hierarchical layout algorithm (See Figure 5) seems more appropriate for following flows than a spring-embedded one (See Figure 6). The basic hierarchical layout algorithm is described in Section 3.1. along with some of the specific variations used by some tools. A spring-embedded layout algorithm views the graph as a physical model where the nodes are objects subject to forces applied by springs between the nodes. A minimum energy level of the system is sought using numerical methods. Although the spring-embedded layout algorithm spaces nodes relatively evenly, no sense of flow is maintained, and the results may appear drastically different each time the algorithm is run.

This paper will discuss the preprocessing necessary to get from the assembler control structure to a graph definition in Section 2. It will also describe how the tools function and compare their results in Section 3. In Section 4, there is a brief evaluation and conclusion. Section 5 contains brief descriptions of a series of items to improve in automating the documentation and questions that arose while compiling this paper.

2 OBTAINING GRAPH INFORMATION:

To automate drawing graphs, I had to write a program to read the assembler code for the control structure and create inputs in formats comprehensible to each of the graph drawing programs. (See Section 3.6.2 for a comparison of graph formats.) This section will give a brief description of the control structure. It will then describe various functions of this preprocessor and its evolution in an effort to get the generated graphs to resemble the manually drawn diagrams.

2.1 Control Structure

The control structure gives an ordering of blocks of code. A state represents a block of code. The order is dependent on inputs to the system. These inputs can be results of calculations or responses received. An event represents the transition between states which may

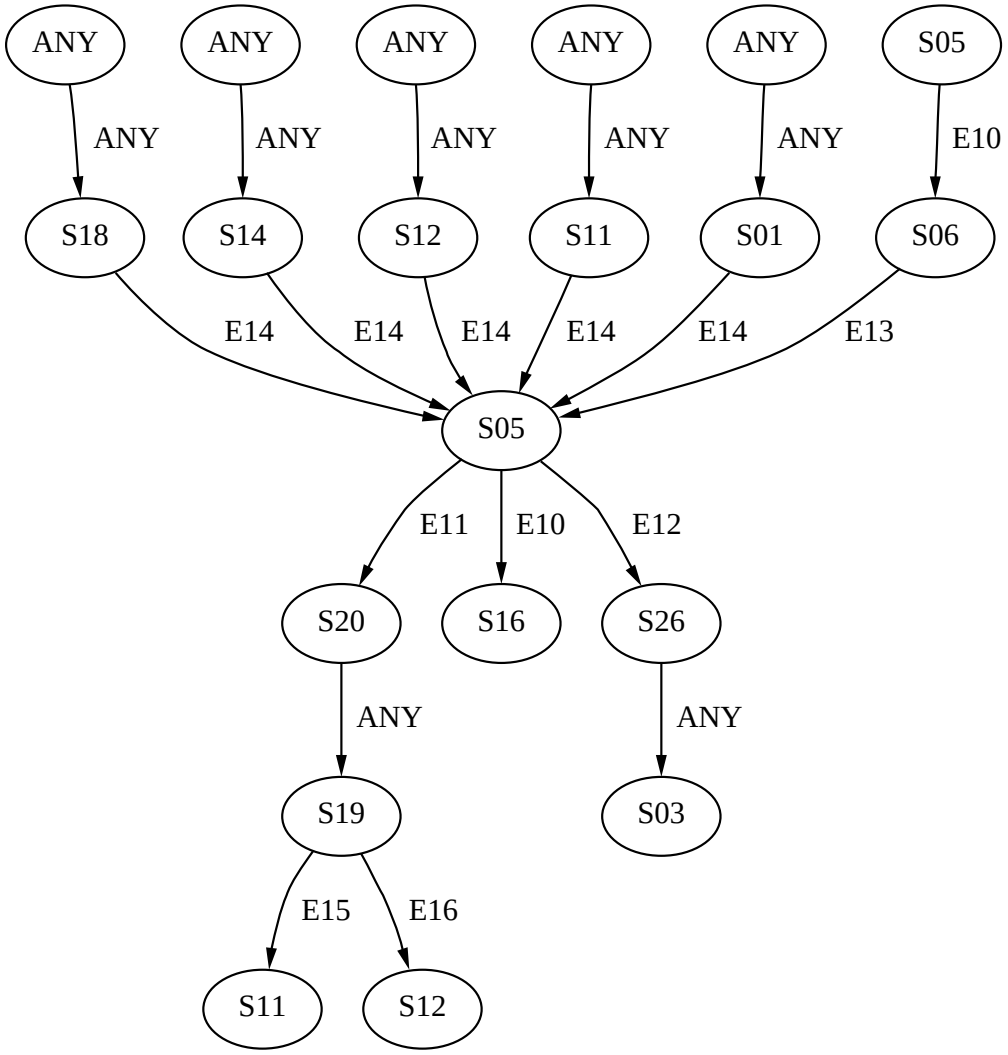

Figure 5: i3_10 drawn with **Graphviz**'s *dot*, which uses a *hierarchical* layout algorithm. Observe the flow from the top of the page towards the bottom of the page. Compare this graph drawing with the one in Figure 6.

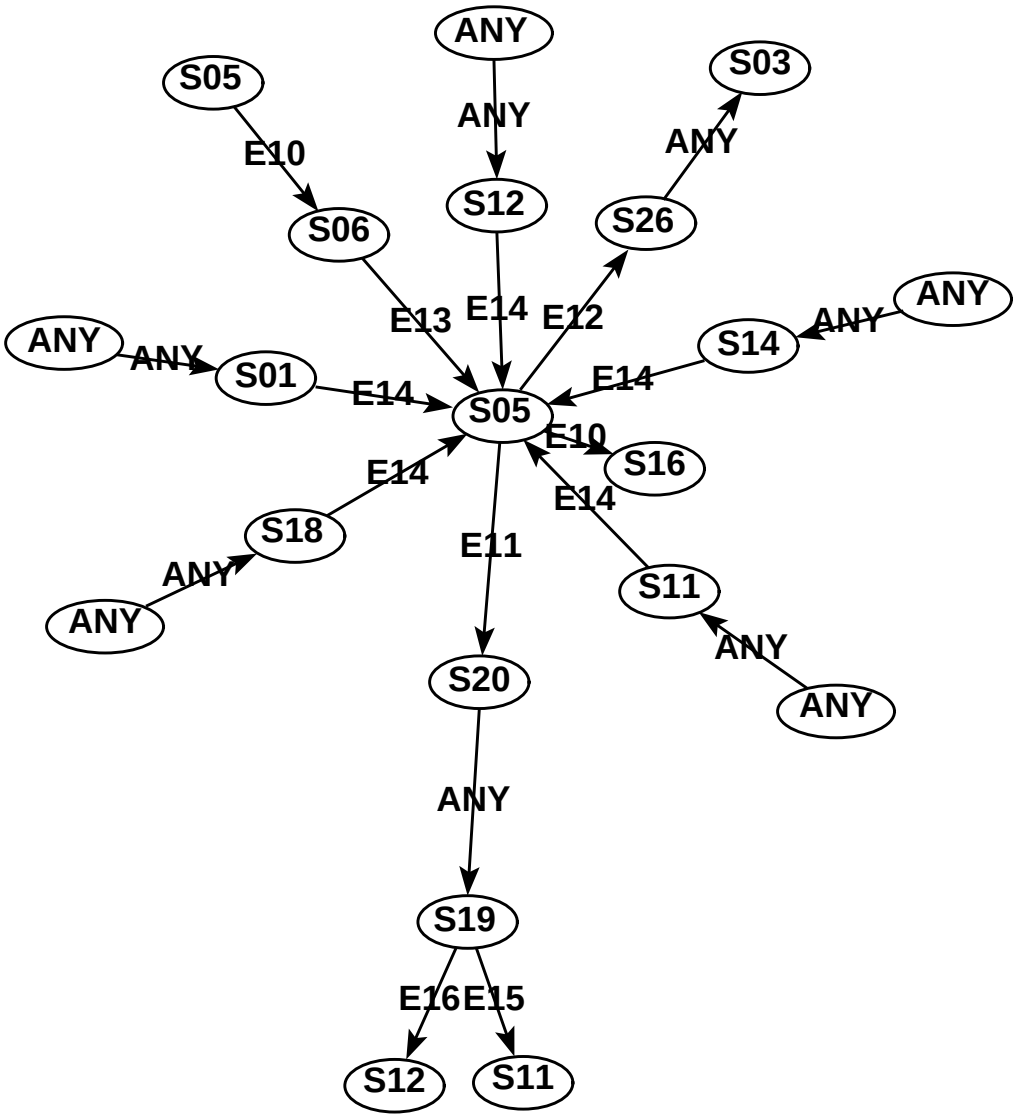

Figure 6: i3_10 drawn with **Graphlet** and one of the available *spring-embedded* layout algorithms. Differentiating sources and sinks takes an effort when reading this graph. It also takes an effort to identify which edges are entering $S05$ and which edges are leaving $S05$. Compare this graph drawing with the one in Figure 5.

be dependent on inputs. The terms "state" and "event" are borrowed from automata theory. In finite-state automata, the state and the event are sufficient to determine the next state, but in this system other information influences the next state. The first piece of information which influences the next state is which of three roles the system is performing. The actual roles are not relevant to this paper. Another piece of information which influences the next state is the error being processed. These two pieces of information are used to partition the system into manageable pieces - resulting in reduced complexity and better performance. A partition, described in the introduction as an error scenario, is named for these two pieces of information. For example, I2_3C is the name of a partition, where 2 is the number associated with the role and 3C is a hexadecimal number associated with the error. In this system, the prior state and prior event can influence the next state, permitting multiple occurrences of a state with different inputs and outputs. I used a *tuple* to store the information (prior state, prior event, current state, and current event) which is needed to determine the next state within a partition and next state itself. There is a graph for each partition with states represented as nodes and events represented as edges.

2.2 Assembler Code to Tuples

I wrote a very simple parser to read the records coding the control structure in assembler. (See the section 5.1.3 for possible improvements to the parser.) Records were grouped by the error being processed. The first record in the group specified the error for itself and any following records. A record contained prior state, prior event, current state, current event, and three next states, one for each system role. Three tuples were formed from each record. These tuples were in different partitions. A tuple contained prior state, prior event, current state, current event, and next state. This record structure simplified coding of those error scenarios that did not differ when the system role changed.

2.3 Traversing Tuples

In order for a graph to provide useful documentation, the tuples have to be linked together. Two tuples, A and B, can be linked with A preceding B, if the current state in A is the same as the prior state in

B, and the current event in A is the same as the prior event in B, and the next state in A is the same as the current state in B. When two tuples are linked, they will share two nodes and the adjoining edge (see Figure 7).

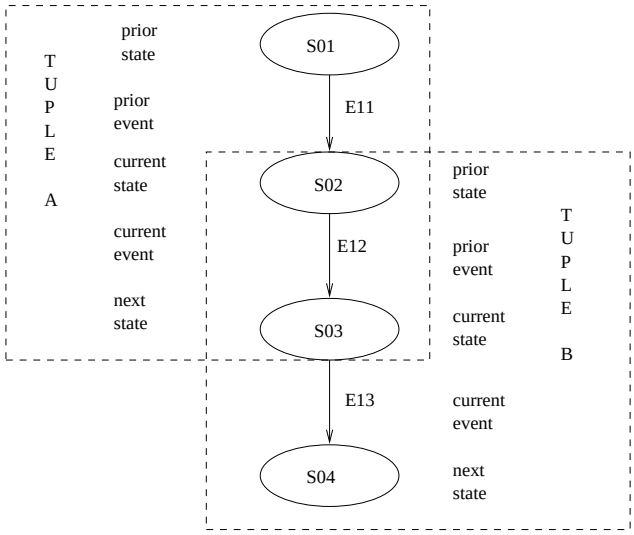

Figure 7: Graph of two linked tuples A and B, where A precedes B.

Since the tuples determine the components of a graph, it is possible for more than one node to have the same label when tuples mentioning the state can not be linked. See Figure 8 for an example. In the example, tuple *i* precedes both tuple *ii* and tuple *iii*, and tuple *iv* can not be linked with any of the other tuples.

2.4 Creating Inputs

Tuples without predecessors were used as starting points. A depth-first search was used to traverse the tuples from these starting points. Since all sources and sinks are actually transitions between graphs, any cycle that would not be traversed would also be unreachable code. The ancestors of a tuple are tracked for loop detection. The loop detection is necessary for termination and creating loops within the graph.

The resulting diagrams consisted of multiple trees possibly with cycles (See Figure 9). Although following through these non-connected graphs is correct, there can be a high quantity of duplication between

tuple id	*prior state*	*prior event*	*current state*	*current event*	*next state*
i	Z	4	A	1	B
ii	A	1	B	2	C
iii	A	1	B	3	D
iv	Y	4	A	1	X

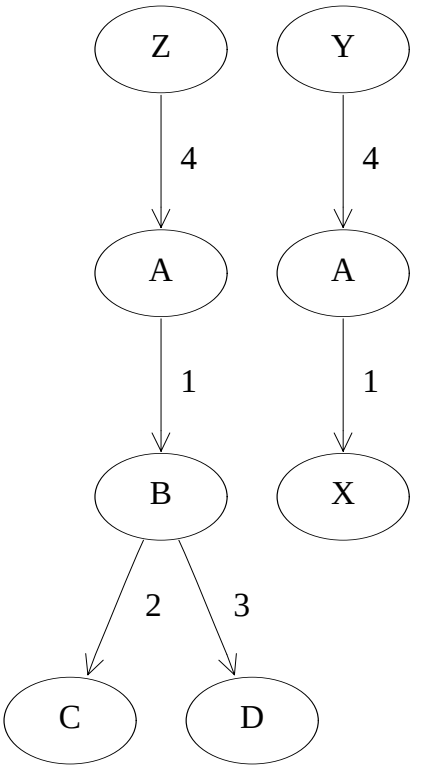

Figure 8: Graph created from list of tuples in the table.

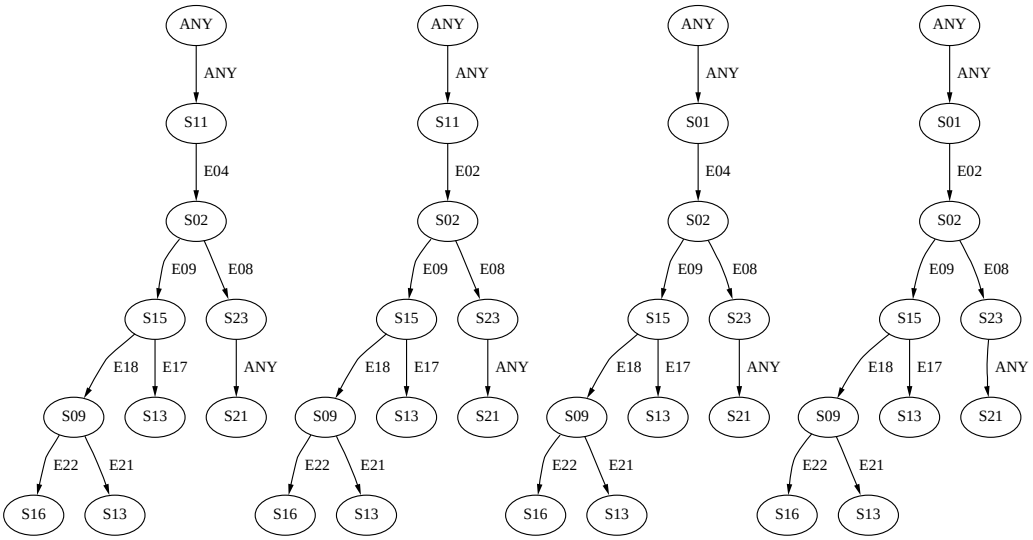

Figure 9: Graph, i1_08, before any merging. Notice the redundancy between non-connected components. Figures 10 and 11 show other representations of the same data with less redundancy.

"trees". This duplication can difficult for humans to see. It is far easier to see the commonalities if sections of the tree are merged.

A simple reduction of duplication is to merge equivalent initial edges. When this merge occurs, the same edge (and the nodes it attaches) is used for all tuples without predecessors which have the same values for prior state, prior event, and current state. (See Figure 10.)

A more complex reduction of duplication is to merge subgraphs that are the same except for the entering edges. (See Figure 11.) If one was working with only the graph, this would require searching the entire subgraph. However, since these graphs are mostly trees and each lower level node contains tuple information, comparing the next two edges and the tuples at the "grandchildren" nodes is sufficient to determine equality. This is actually done while the graph is being built thus avoiding the necessity of building this duplicate subgraph.

While comparing the generated graphs to the manually produced graphs for correctness and completeness of representation, the glaring difference was nodes that only appeared in one form of the graph. The nodes that appeared only in the generated graphs were "logic errors" that are checked for in the code, but are hopefully avoided in practice.

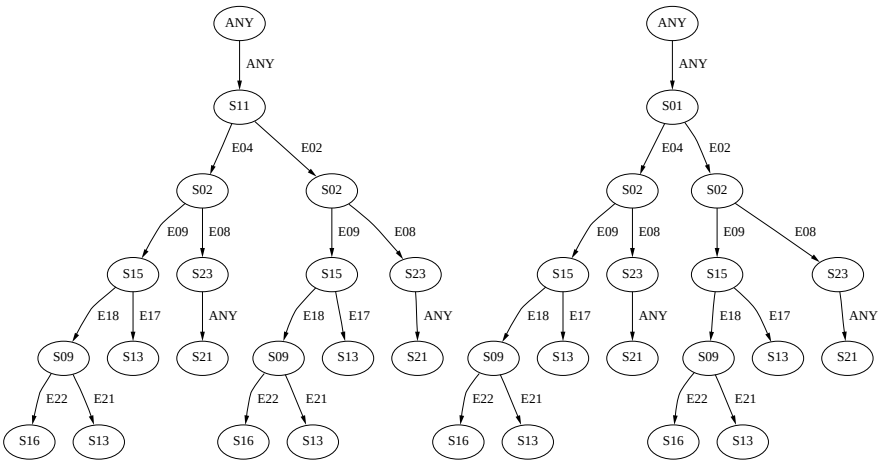

Figure 10: While this diagram represents the same information as the diagram in Figure 9, the merging of duplicate initial edges and the associated nodes, has eliminated four nodes and two edges. See Figure 11 for another representation of the same data with even less redundancy.

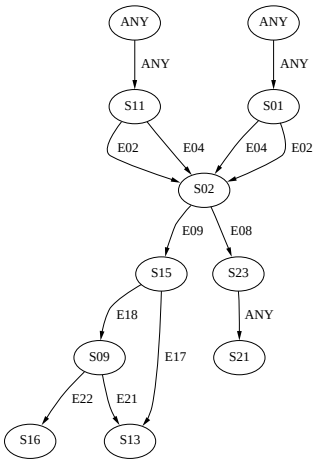

Figure 11: When comparing this diagram with the diagram in Figure 10, observe that subgraphs starting with $S02$ or $S13$ are merged. Notice the simplicity of this diagram with fewer nodes and edges.

These nodes were eliminated by ignoring all tuples with S99 as the next state. The nodes that do not appear in the generated graphs are not mentioned in control structure which was parsed, but are within the code which a node represents. (See the section 5.1.1 for a possible solution.) Some of these glaring differences appear to be that the manually drawn diagrams were out-of-date.[2]

Two out of fifty one diagrams[3] were different (See Section 5.1.1) and require further examination.

While some of the manually drawn graphs were not exactly the same as the generated graphs, they conveyed the same information.[4] This was usually caused by how nodes were merged. The most common merge difference was leaf nodes that were duplicated in the manual drawings, but existed only once in the automated drawings. The manual drawings were not consistent in the treatment of this situation.

2.5 Analysis of Preprocessing

Given r as the number of records and t as the number of tuples, $r = 3t$. The parsing of records to create tuples is implemented in $O(r)$ time with a space requirement of $O(t)$. The linking of tuples was implemented in $O(t^2)$ time and space. The actual space usage was not near this worst case bound. My tool to build the input for the graph drawing tools without any merging takes $O(t^2)$ time and with merges takes $O(t^4)$ time.

[2]Diagrams which appear out-of-date:

- i1_18 and i1_1C transitions from the node, S15, on the edge, E17
- i2_3C and i3_3C transitions from the node, S02, on the edge, E19
- i2_30 and i3_30 show an additional node, ANY, attached to another additional node, S33, which is attached to the node, S07, where the additional edges are both labelled ANY.

[3]Hand drawn versions of diagrams i2_28 (see Figure 22) and i3_28 differed from their generated counterparts.

[4]Graphs drawn differently: i1_0C, i1_28, i1_34, i1_3C, i2_08, i2_10, i2_18, i2_34, i2_3C, i3_08, i3_10, i3_18, i3_1C, i3_28, i3_34, and i3_3C

3 GRAPH DRAWING TOOLS

This section starts with an overview of the hierarchical layout algorithm. It then goes on to differentiate four tools which apply variations of this algorithm: **DaVinci**, **Graphlet**, **Graphviz**, and **VCG**. Finally it compares two aspects of the tools: the appearance of the resulting graph drawings and the usability of the tools.

3.1 Algorithm

This section starts with an overview of the hierarchical layout algorithm. Each of the tools examined determines the layout of the graph by examining the hierarchy of the graph. They each go through four basic phases:

1. Determine the relative rank of the nodes, where rank is used to determine the vertical placement of nodes.

   Dummy nodes are often added so that no edge crosses several levels.

2. Order the nodes from left to right within a rank to reduce the number of edge crossings in the layout.

   Minimizing the number of edge crossings is NP-Complete [PEW86]. A heuristic to avoid this complexity follows. Given an initial ordering within a rank, a sequence of iterations is performed to improve the orderings. An iteration traverses through the ranks from first to last or vice versa. Each vertex within a rank is assigned a weight based on the relative positions of incident vertices in the previous rank. This rank is used to reorder the vertices in the rank. Variations of the heuristic vary the number of iterations and the directions traversed.

   Two common weighting methods are the barycenter function [STT81] and the median function [EW86].

3. Position the nodes with the rank and order information.

   Vertical spacing is relatively trivial using the rank information. (In these diagrams the nodes are the same size, so all of the nodes in a rank have the same height.)

4. Add edges.

3.2 DaVinci[5]

The online documentation gives the following description.

> The graph layout algorithm of **DaVinci** is based on the heuristical approach of Sugiyama et al. ([STT81], [SM91]) and was refined by a non-deterministic component to get better results (discussed in [FW94]).

3.2.1 Determine Rank of Nodes

The rank appears to be inherent in the recursive nature of the input.

3.2.2 Node Ordering Within Ranks

The layout algorithm does not overcome a bad initial ordering of nodes near the top of the page. This appears to be caused by not doing a traversal of the ranks from the bottom up.

3.3 Graphlet[6]

Although DAG stands for directed acyclic graph, there are two layout options which appear applicable: DAG and Extended DAG. Apparently the cycles in a graph are eliminated (see Section 3.4.1 for a possible method). The documentation on these options, who wrote them and how they work, is sketchy at best. It appears that the Extended DAG is by Harald Mader [7]. One might determine how it works by looking at an animation of the layout algorithm. There are several levels of animation.

3.4 Graphviz (*dot*)[8]

Graphviz is actually a collection of tools. The two that are of interest for this problem are *dot*, which is useful for batch processing, and *dotty*,

[5] **DaVinci** V2.0.3 December 23, 1996 Michael Fröhlich, Mattias Werner, Sebastian Mangels http://www.informatik.uni-bremen.de/ davinci

[6] **Graphlet** Version 2.8.2-beta Universität Passau 1995–1998 http://www.fmi.uni-passau.de/Graphlet Upgraded from 2.2.0-final to be able to export the diagram from Graphlet.

[7] taken from the window to set its options

[8] John Ellson, Eleftherios Koutsofios, Stephen North. http://www.research.att.com/sw/tools/graphviz/

which provides a Graphical User Interface (GUI) for *dot*. The other tools within this package will not be addressed.

3.4.1 Determine Rank of Nodes

[GKNV93] breaks rank assignment into two major pieces: making the graph acyclic and applying the network simplex formulation [Chv83] to node ranking with a goal of short edges.

A depth-first search starting at either source or sink nodes partitions the edges into tree edges and non-tree edges. The tree defines a partial order on nodes. The non-tree edges can be further classified. A cross edge connects nodes that are unrelated in the partial order. A forward edge connects a node to a descendant. A back edge connects a descendant to an ancestor. A cycle can be broken by reversing a back edge. Determining a minimum set of edges to reverse is NP-complete [PEW86]. The heuristic used takes each non-trivial strongly connected component, counts the number of times each edge forms a cycle, and reverses the edge with the maximum count.

The initial step in ranking the acyclic graph is to determine a feasible spanning tree and a cut value for each edge. The cut value is defined as the sum of the weights of all edges in the tail component to the head component minus the weights of all edges in the head component to the tail component, where the components are determined by removing the edge from the tree and examining the direction of that edge. Any edges with negative cut values are replaced by other edges connecting the components. Nodes having multiple feasible ranks are assigned to a rank with the least nodes to reduce crowding.

3.4.2 Node Ordering Within Ranks

Dot uses a refinement of the median method. When there are two median values, it uses an interpolated value biased towards the side where vertices are more closely packed. It then reduces obvious crossings once the rank has been sorted. This ensures that a given ordering is locally optimal with respect to the adjacent vertices. It ignores the positioning of vertices without adjacent vertices in the previous rank. It attempts to get all flat edges from left to right. It flips nodes with equal values on alternate forward and backward traversal in an attempt to find symmetry.

3.4.3 Position Nodes

In determining the horizontal spacing of nodes, *dot* starts with all of the nodes spaced from the left margin with the minimal separating distance. It then sweeps the ranks and improves the placement. It improves the placement by using median locations of neighbors (either up or down). *Dot* attempts to straighten chains of virtual nodes and assign them the same X-coordinate. For each node, if the nodes to its left can be moved to the left without violating any positioning constraints, then *dot* shifts them.

3.4.4 Position Edges

A virtual box exists for each rank that an edge passes through. Its height is the average height of the nodes. Its width is the separation between the nodes. Vertically between the inter-rank boxes just described, are virtual node boxes. These are not constrained horizontally to permit the splines to be as smooth as possible. When the inter-rank space is adjacent to a termination point, if there are previously computed splines with the same endpoint, the inter-rank space is divided vertically, and each box is horizontally constrained by the adjacent splines at the division points.

When an edge has a section that is almost vertical, *dot* draws it vertically, adjusting the connecting sections accordingly. *Dot* draws edges within boxes using splines.

Labels on inter-rank edges are represented as off-center virtual nodes, which guarantees that they do not overlap other nodes, edges, or labels. Setting the minimum edge length to two and halving the separation between ranks compensates for the label nodes.

3.5 VCG[9]

3.5.1 Determine Rank of Nodes

VCG has many options for determining rank. I chose to use one based on calculating a minimum spanning tree using a depth-first search, because of how loops were treated. The node from which the loop was entered was at the top, with successive nodes lower than preceding

[9]VCG/XVCG - USAAR Visualization Tool V.1.3 Revision: 3.17 Date: 1995/02/08 11:11:14 Copyright (C) 1993-1995 I.Lemke/G.Sander and the Compare Consortium. http://www.cs.uni-sb.de:80/RW/users/sander/html/gsvcg1.html

ones. I found this flow easier to read even though it takes up more vertical space (see Figure 20).

3.5.2 Node Ordering Within Ranks

Although the documentation mentions calculating the number of crossings using a sweep-line algorithm, the tool includes both barycenter and median options.

3.6 COMPARE AND CONTRAST:

3.6.1 Appearance

Two graphs are used to demonstrate the tools.

- The graph, i3_10, has the most nodes and the most edges.

  Graphviz (see Figure 12) did not have any edge crossings, but the other tools did. The other tools had other problems as well. **DaVinci** (Figure 13) did not space the nodes very evenly. **Graphlet** (Figure 14) handled multi-edges and edge labeling poorly. Although these additional problems with **Graphlet** could be overcome (see Figure 15), there were still the edge crossings. **VCG** (see Figure 16) had edges that were pointing up, but were not looping back. The edges drawn by **VCG** had breaks for the labels. Both the edges and the node outlines were very thick in the drawings made by **VCG**, making them less attractive.

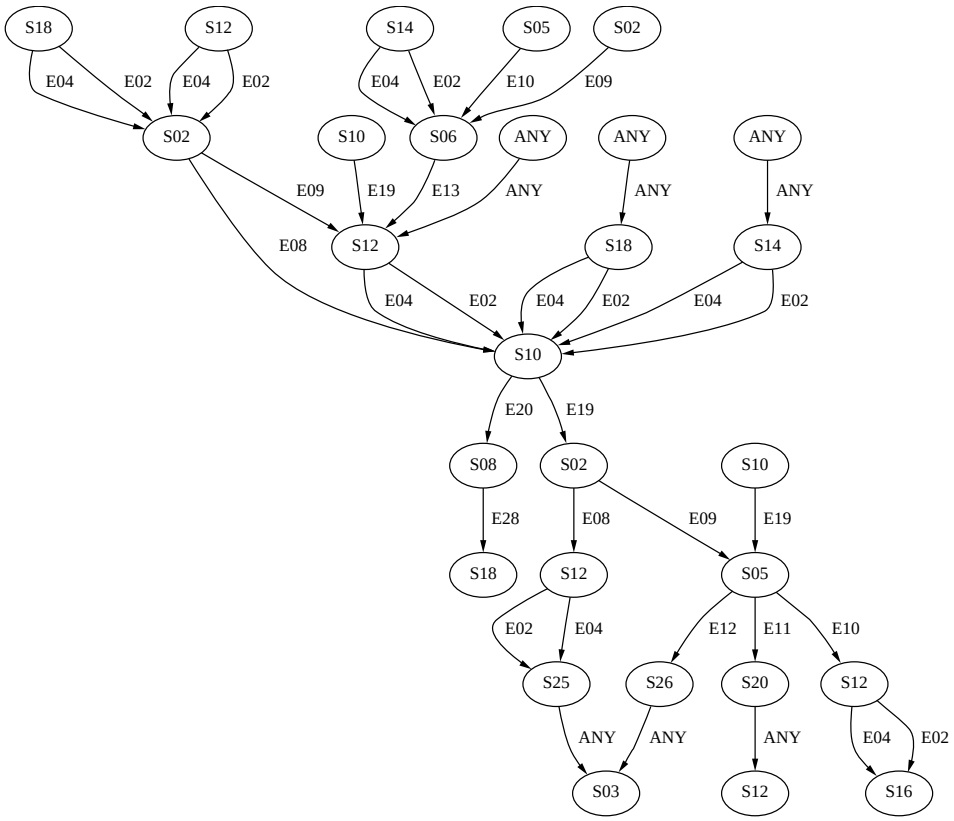

Figure 12: A graph with many nodes and edges, i3_10, drawn with **Graphviz**'s *dot*. Notice there are no edge crossings and the edges are continuous.

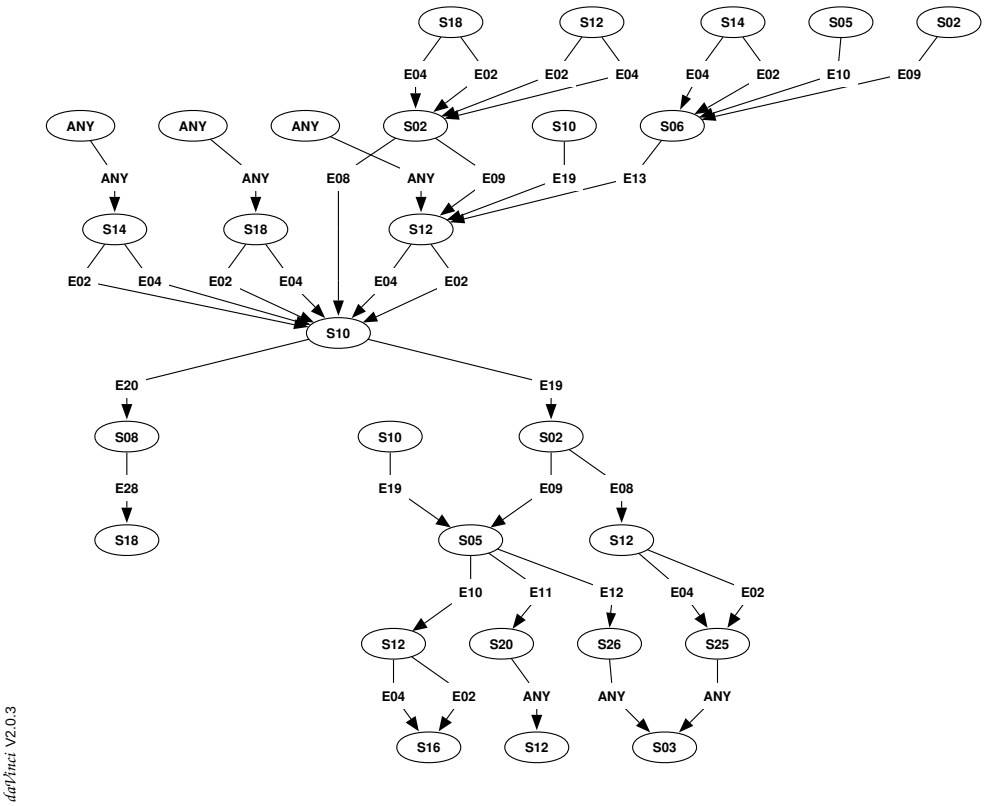

Figure 13: A graph with many nodes and edges, i3_10, drawn with **DaVinci**. Observe that the edges, $S02 \xrightarrow{E08} S10$ and $ANY \xrightarrow{ANY} S12$, cross. Also observe that the nodes preceding $S06$ are all to one side of that node.

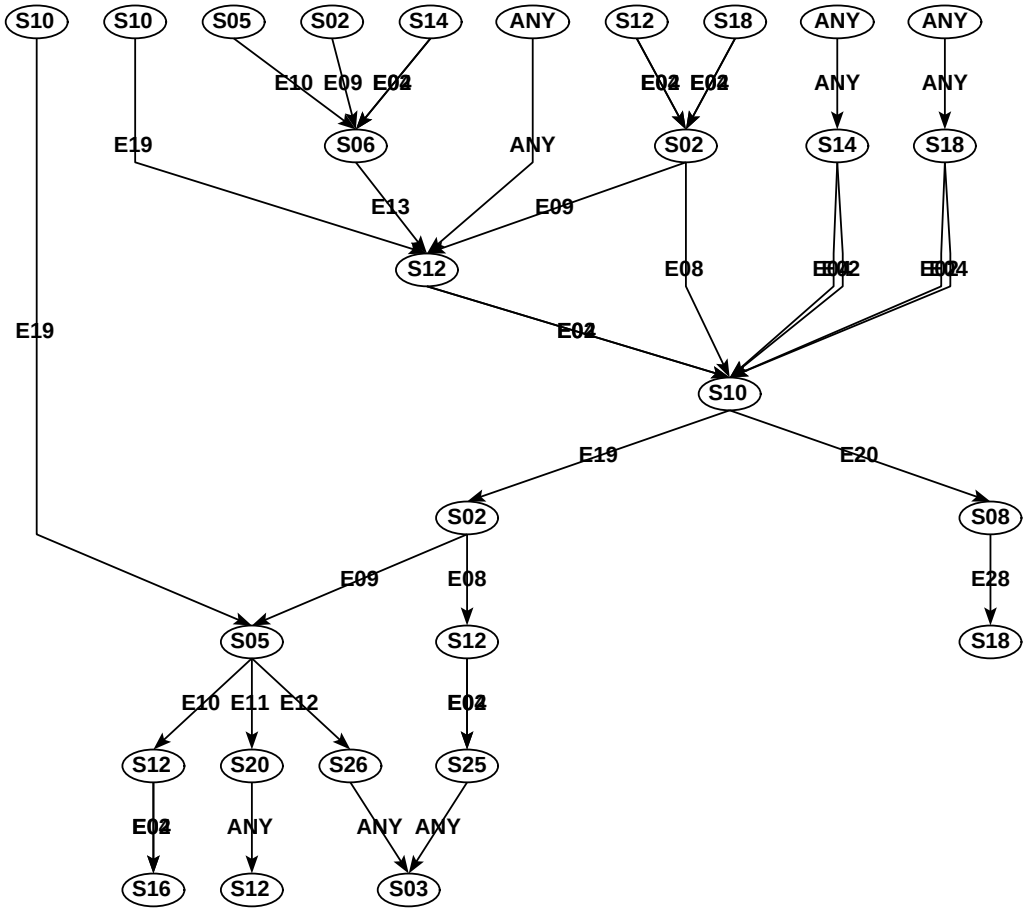

Figure 14: A graph with many nodes and edges, i3_10, drawn with **Graphlet** using edge labels. Observe the poor handling of multi-edges and edge label placement.

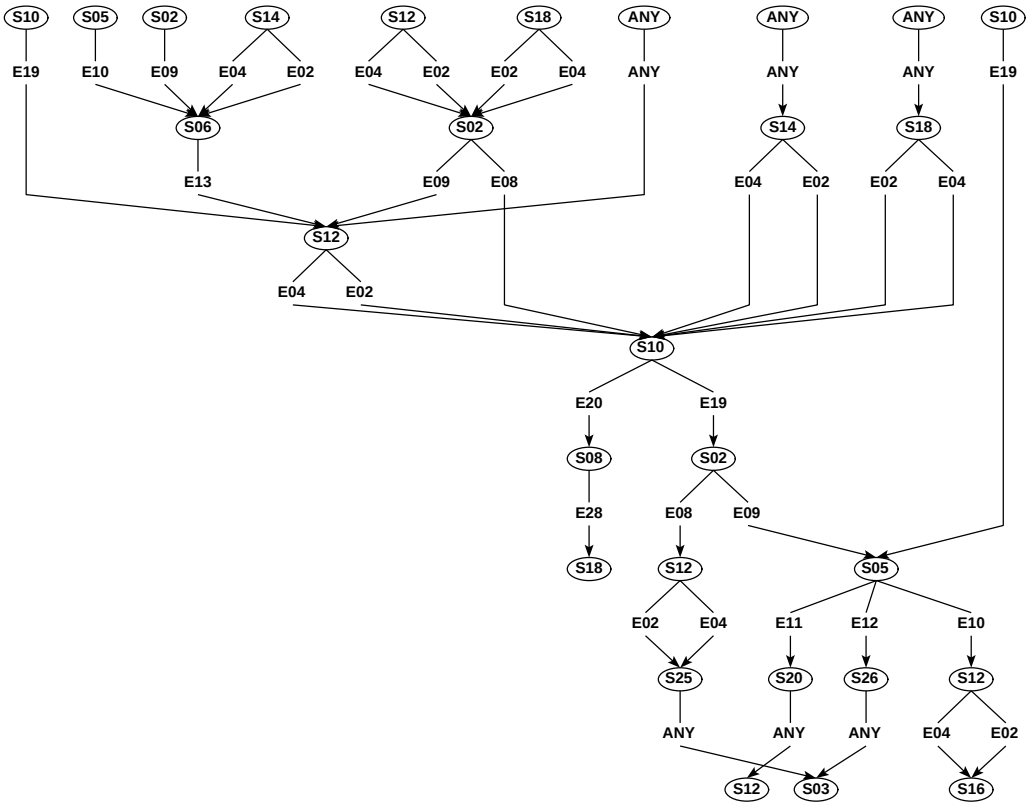

Figure 15: A graph with many nodes and edges, i3_10, drawn with **Graphlet** using nodes with different attributes to contain the edge labels rather than the built ability for edge labels. Also notice that two edges, $S20 \xrightarrow{ANY} S12$ and $S25 \xrightarrow{ANY} S03$, cross.

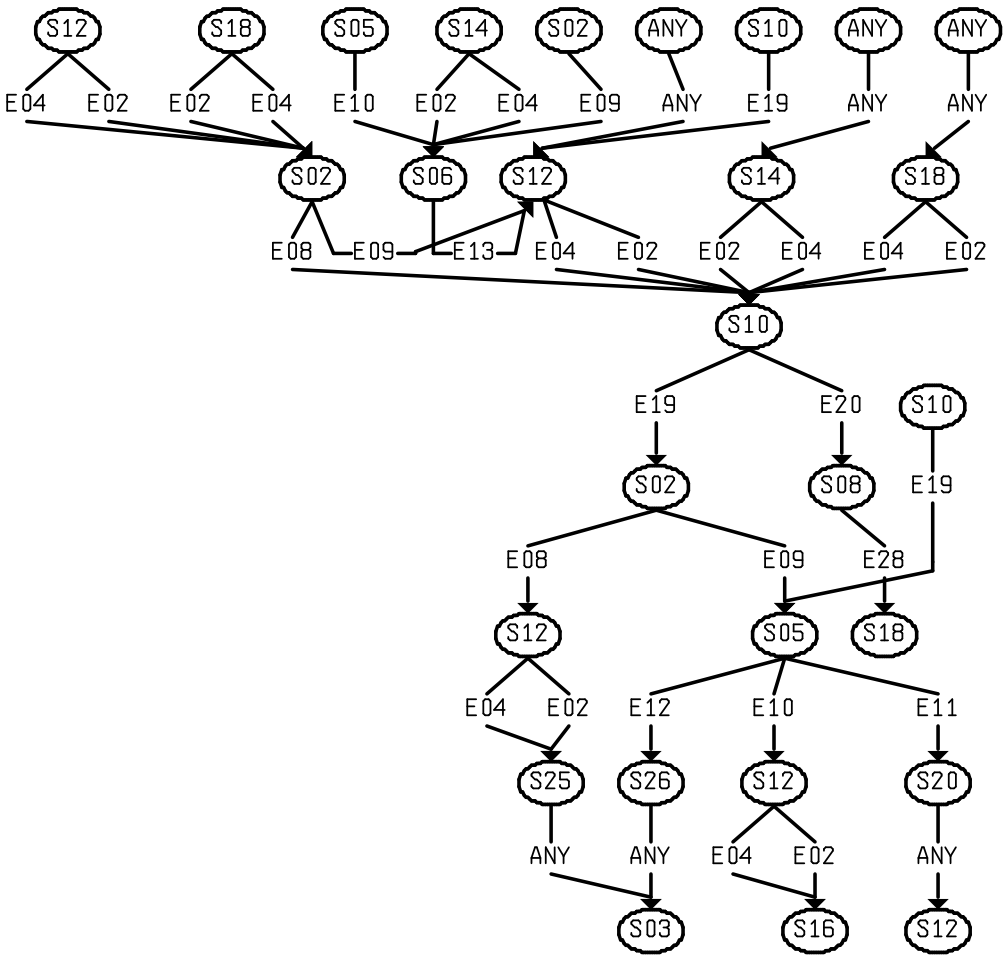

Figure 16: A graph with many nodes and edges, i3_10, drawn with **VCG**. Observe there is an edge crossing and that an edge that points up is not looping back (see $S02 \xrightarrow{E09} S12$).

- The graph, i2_10, contains a relatively large cycle.

  Although the flow of a loop is not top down in the diagram drawn with **Graphviz**'s *dot* (Figure 17) and it is with each of the other tools, *dot* still produces the best diagram. In the output of **DaVinci** (see Figure 18), the edge label, $E10$, is below the node, $S05$. This problem was not apparent in the diagram without a loop. This is caused by edge labels being arranged as nodes. The output of **Graphlet** (Figure 19), handles multi-edges or edge labeling extremely poorly. In the output of **VCG** (Figure 20), the edge between nodes $S26$ and $S03$ appears to be a loop back rather than a forward transition at first glance.

  The lack of top down flow in the diagram drawn with **Graphviz**'s *dot* could be caused by the choice of which edge was reversed or by the goal of keeping edges as short as possible.

Non-connected graphs are not compared in this paper, since these graphs are relatively simple and the results are comparable for all tools.

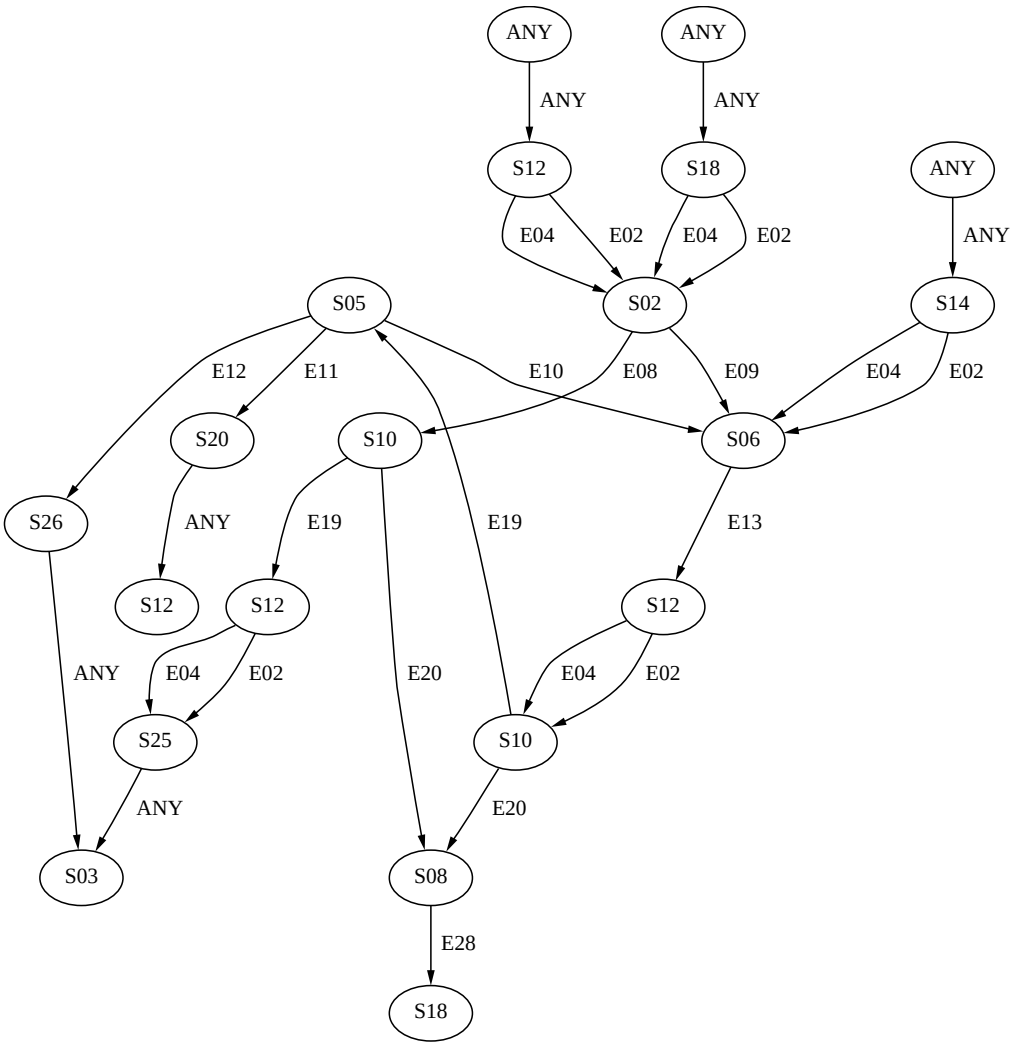

Figure 17: A graph containing a cycle, i2_10, drawn with **Graphviz**'s *dot*. Notice that a node in the loop, $S05$, is higher than the first node to be traversed in the loop, $S06$. There are are also unnecessary edge crossings. Despite these faults, this graph is legible.

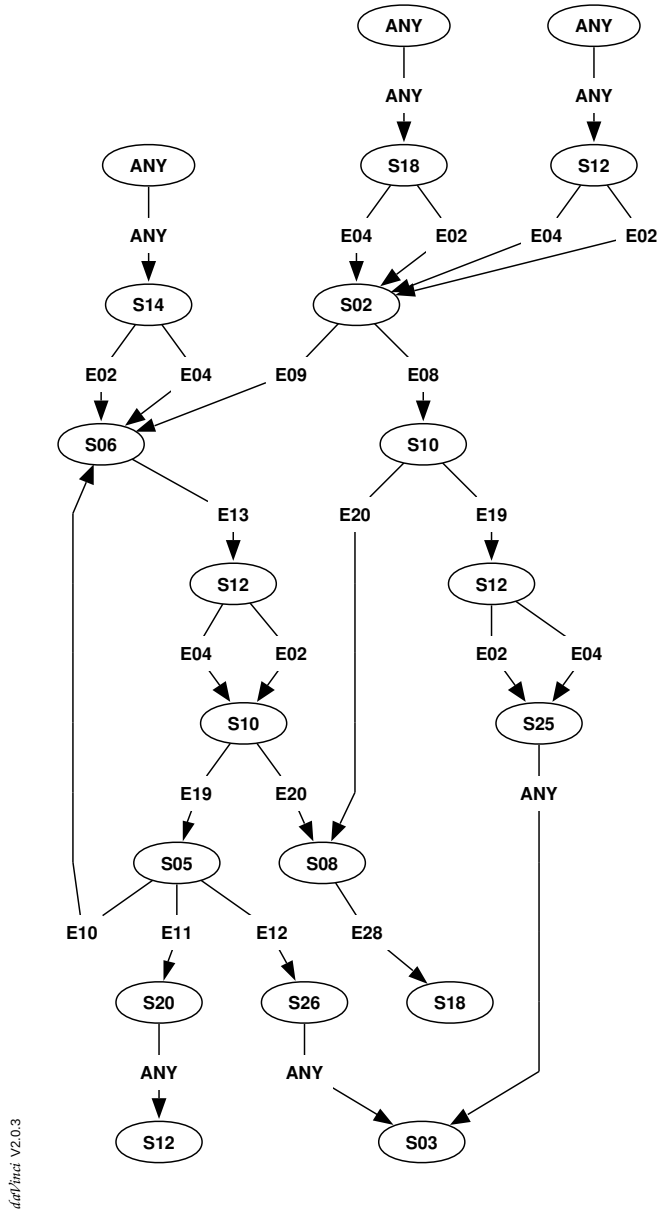

Figure 18: A graph containing a cycle, i2_10, drawn with **DaVinci**. Notice that the edge which loops back, $S05 \xrightarrow{E10} S06$, goes down to its label and then up to the node.

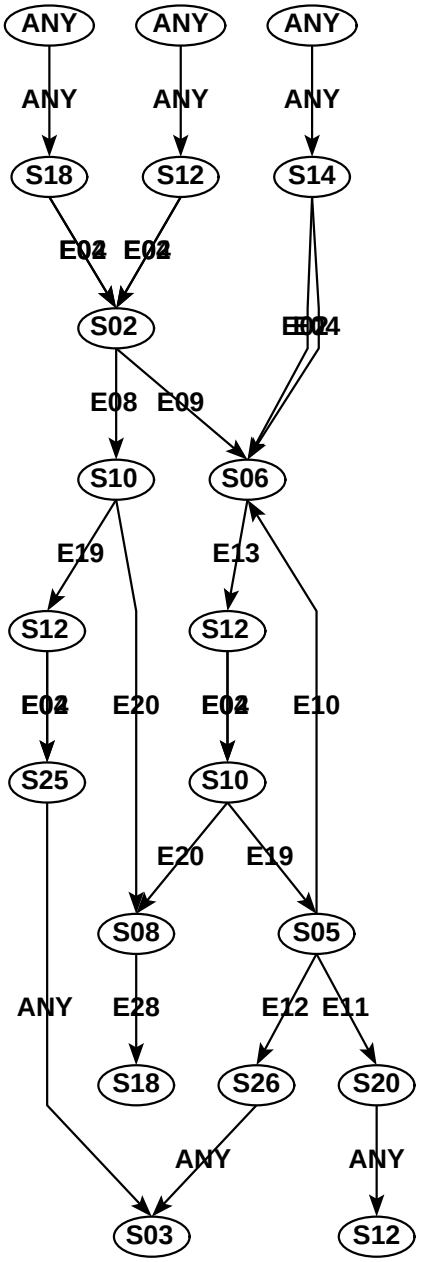

Figure 19: A graph containing a cycle, i2_10, drawn with **Graphlet**. The node placement and flows of the loop are clear, but the placement of multi-edges and edge labels renders the diagram illegible in places.

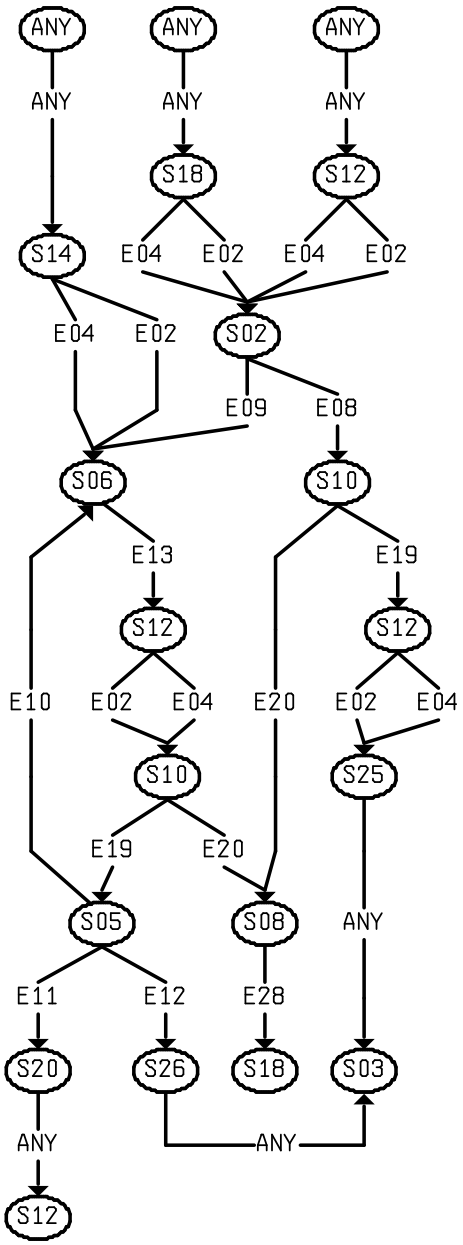

Figure 20: A graph containing a cycle, i2_10, drawn with **VCG**. The flow of the loop is clear. At first glance, it is not obvious that $S03$ is a sink.

3.6.2 Usability

Since my focus was on automatically generating graphs, I wanted to be able to generate a file that could be fed to the tool, run a layout algorithm, and output a file to print or include in a document. I only used the Graphical User Interface (GUI) for fine tuning which options I would use for my final results. This paper will not discuss the graph editors in any depth despite their usage in creating some figures in this paper.

		DaVinci	**Graphlet**	**Graphviz**	**VCG**
Input	format	recursive structure	adjacency list	adjacency list	adjacency list
	edge labels	no	yes	yes	yes
	additional features	none	none	none	layout options
Interface	command line	input file	input file	input file output options	input file layout options output options
	API	yes	yes	yes	no
	Graphical User Interface (GUI)	user friendly, well organized	user friendly, keystrokes non-functional	fast and easy, requires attribute knowledge	exists
Installation		easy	easy	minor fix necessary	need to change compile options

Figure 21: *Tool Usability Summary*: **Graphviz** (with its components *dot* and *dotty*) is clearly the best suited tool. *Dot* makes it possible to generate diagrams in batch mode. *Dotty* provides a fast, simple GUI for manipulating diagram elements.

Input Format Three of the tools (**Graphlet**, **Graphviz**, and **VCG**) use their own version of an adjacency list. An adjacency list may consist of a list of edges with attributes and the originating and terminating nodes, a list of nodes with attributes, and some attributes

associated with the graph. (See Appendix 5.2 for examples.) Alternatively, **DaVinci** required a more complex description of a graph . When a node is given, all edges starting at that node must be given. When an edge is given, the node to which it is going must also be given. This results in a recursive procedure. When a node is encountered again, a reference may be given.

DaVinci also does not handle labels on edges. The workaround is to place the label text in a node, use different display options for original nodes and added nodes, and replace the original edge with two edges. This same workaround was applied to the placement of edge labels and to multi–edges for **Graphlet**.

Automation All of the tools permit the name of the input file to be specified on the command line. Both **Graphviz**'s *dot* and **VCG** have command line arguments for specifying how the output should be formatted. This is critical in automating the process of having a tool read in a graph file, lay it out, and output a diagram. **Graphlet** and **DaVinci** have Application Programmer Interfaces (APIs), which might be a more complex alternative to using command line arguments. However, neither of them has a method of controlling the postscript options, such as page size, through the API.

Additionally, both **Graphviz**'s *dot* and **VCG** permit layout options to be specified with the graph. **VCG** also permits control of the ranking and node ordering from within the graph file or the command line.

Graphical User Interface (GUI) **DaVinci**'s GUI was fairly straight forward. A positive aspect of the interface is that the postscript options are available on the print menu.

Graphlet has a nicely defined interface for accessing many of its features. The ability of setting the layout options and apply them from the same place, is convenient for experimentation. Unfortunately the online documentation was not current with the software release. Not being able to save my printer options (letter sized paper, portrait, and fit page) was another annoyance. In **Graphlet**, the the commands on the menu bar do not seem to be accessible from the keyboard.

Dotty, a part of **Graphviz**, does not provide a GUI that guides a user. However, the accompanying documentation is reasonable, and once you know it, the interface seems fast and straight forward.

VCG had sliders that were not intuitive to me.

Installation The installations of two tools, **Davinci** and **Graphlet**, were problem free. After installing **Graphviz**, the script, *dotty*, needed minor syntactical modification. Installing **VCG** required changing the compiler options.[10]

4 CONCLUSION:

Initially, I thought that finding a good tool to draw the graphs was the main problem. While I learned about graph drawing for directed graphs and spent time and effort installing and using the tools, the harder problem was actually generating the input for the tool from the assembler control structure. I developed a viable method for generating the input. Section 2 describes the preprocessor that I built. Section 5 describes how the preprocessor could be improved or made obsolete.

Several interesting items surfaced while comparing graph drawing tools. First, hierarchical layout algorithms are far more appropriate for trying to follow a flow than spring-embedded algorithms. Force directed algorithms such as the spring-embedded one are actually good for node spacing within the hierarchical model. Second, most of the graphs associated with this problem have a relatively small number of nodes and edges. This simplicity means that all of the tools draw adequate pictures for most of the diagrams.

Graphviz, with its components *dot* and *dotty*, is currently the best tool to use for automating the documentation of the assembler control structure by drawing graphs illustrating the flow of control. An additional option to permit printing of a title on the graph would have been useful. (This was not provided by any of the tools.) I wrote a short script to find the title in the comments of the PostScript output and add a command to print the title on the graph. **Graphviz** does a reasonable job of node placement and elimination of edge crossings. **Graphviz** also does an outstanding job of drawing multi-edges and placement of edge labels. **Graphviz**'s input format of a modified adjacency list is relatively easy to produce. Automation is quick and easy with output options available on the command line.

[10]Thank you, Henry.

5 FUTURE WORK:

Although the preprocessing program described in Section 2 enabled the automation of documenting the control structure, there are improvements that could be made to the preprocessor. These are described in Section 5.1. Additionally, the following section outlines a possible way to eliminate the need for the preprocessor.

5.1 Data Preprocessing

Future work on the code described in Section 2 falls into the following categories: inconsistencies, readability, robustness, and efficiency.

5.1.1 Inconsistencies

One of the problems ignored in this paper is transitioning between partitions. The hand-drawn diagrams contain a line separating the sources, the edges attached, the nodes attached and sometimes additional edges and nodes from the rest of the nodes and edges. The separated information (including sources), was actually traversed while in a different partition but is needed in this partition to determine the appropriate next node. This piece of the problem can be broken into two parts: determining which nodes belong on which sides of the line and actually getting a graph drawing tool to draw the line. The hand-drawn diagrams also have additional nodes showing when partition changes occur or when the control structure is left. This often happens at the current sinks, but a partition change may also occur at other nodes. All of these additional nodes do not come from the control structure but rather from the code represented by the nodes. These blocks of code can be dependent on information from the control structure. Although the information for these additional nodes is obtainable and would be easy to add to the current graph inputs, automating this would provide another challenge.

Section 2.4 mentions that two diagrams are not consistent, i2_28 and i3_28. (See Figures 22 and 23 to see manual and generated versions of i2_28.) They both show similar inconsistencies. One major inconsistency is an "unexpected" side effect in the control flow[11]. Further investigation is required to determine whether the diagram is correct or there is an error in the rules used to link tuples and create graph inputs. Another inconsistency was the use of outside knowledge, such as

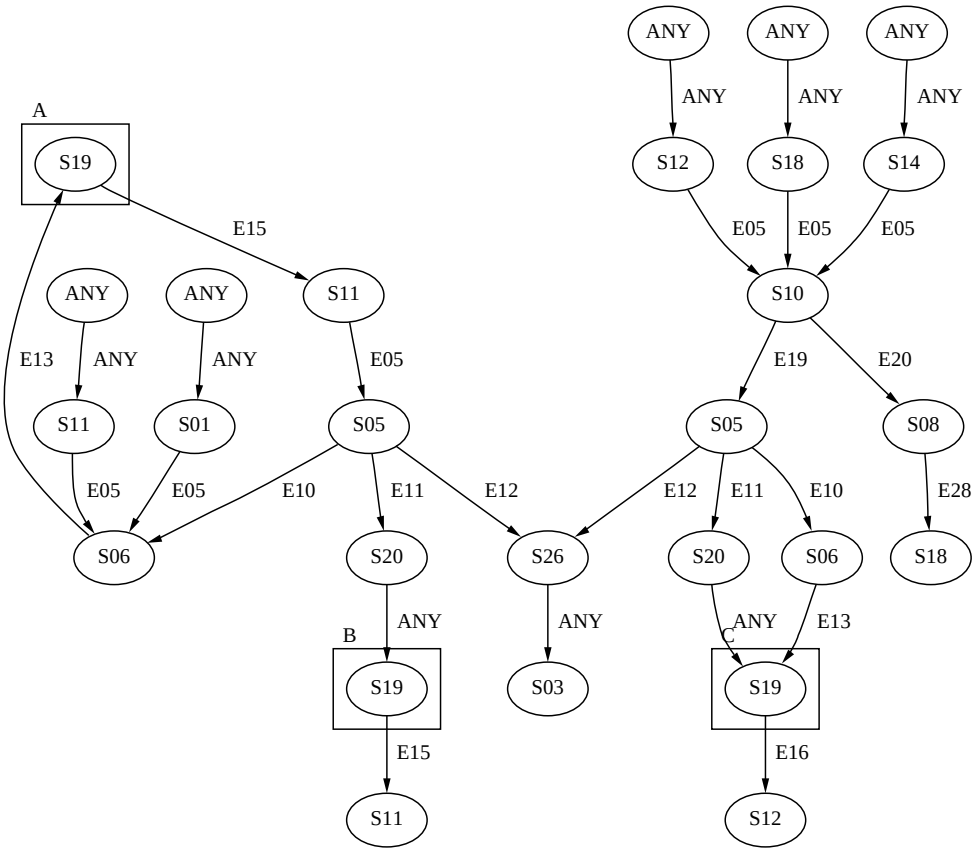

Figure 22: This version of graph i2_28 was drawn using a graphical editor, **Graphviz**'s *dotty*, to match the hand-drawn diagram. The graph input data was then modified by adding boxes to emphasize certain nodes. Compare it with the graph which was drawn with the generated input which contained fewer nodes due to merging (see Figure 23). Note the three instances of node $S19$ as well as the duplication of of the edge $S19 \xrightarrow{E15} S11$. Also note the duplication of node $S12$.

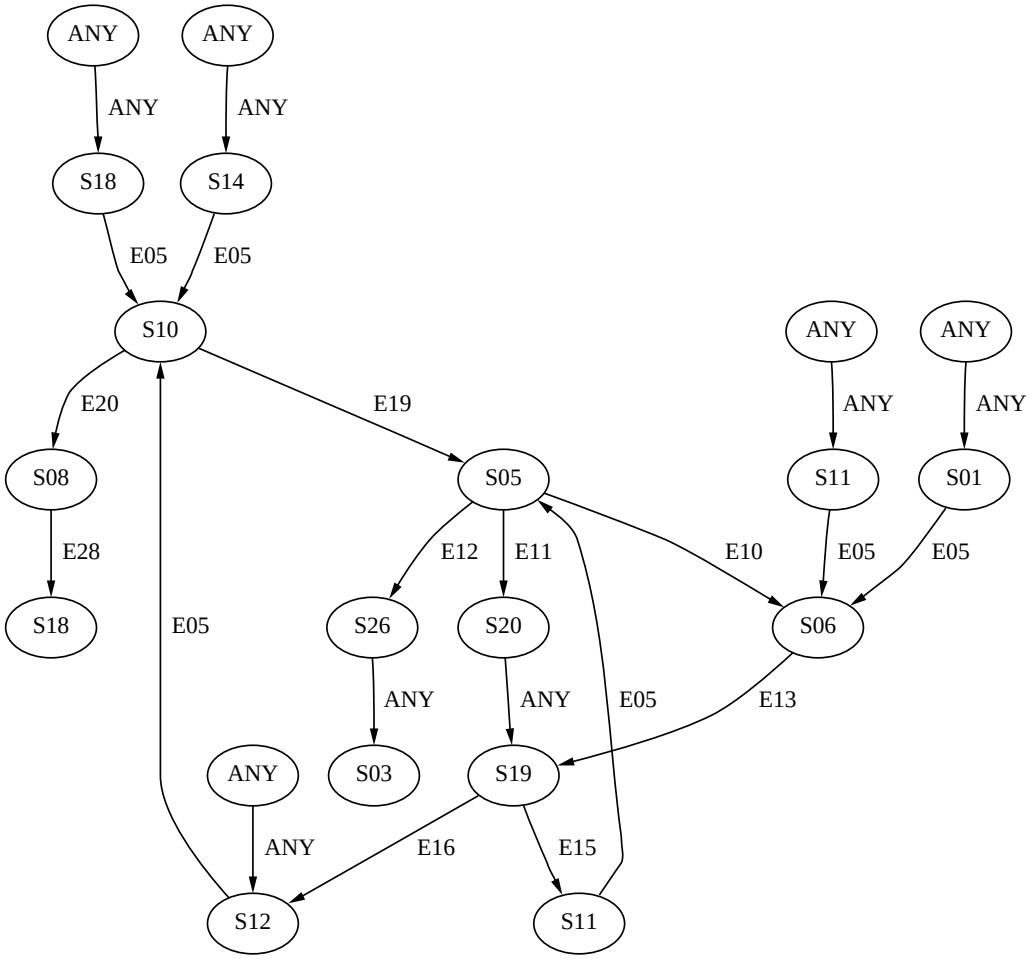

Figure 23: Compare this diagram of i2_28, which uses automatically generated input, with the same diagram as constructed manually (see Figure 22). The $S12$ nodes in the manual diagram were combined. This reflects a tuple ($S19$, $E16$, $S12$, $E05$, $S10$) that is not documented in the manual diagram. This may be an error in the manual diagram.

hardware configuration, in the graphs[12]. Including this information in the generated graphs is yet another challenge.

5.1.2 Readability

To improve readability in the diagrams, the preprocessor could merge multiple edges with the same sources and destinations into one edge, where the label reflects all of the edges (see Figure 24). This is

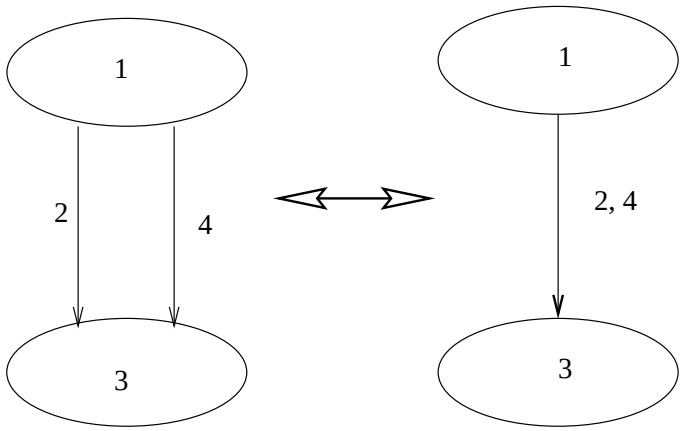

Figure 24: Edges 2 and 4 can be merged since they share the same preceding node, 1, and following node, 3.

done in the hand-drawn diagrams. This would be especially useful for **Graphlet**, which handles multi-edges poorly.

Another way to improve the readability of the diagrams is to merge those source nodes having the same name, so that the resulting node

[11]The edge $S19 \xrightarrow{E15} S11$ occurs in two places (see boxes A and B) in Figure 22. The preprocessor links the same tuples onto both of these edges where as the hand-drawn diagram shows an $S11$ as a sink. Further investigation is required to determine whether the generated diagram is showing an "unexpected" side effect in the control flow or an error in rules used to generate diagrams.

[12]The preprocessor would combine the $S19$ nodes in boxes A, B, and C in Figure 22, after examining the preceding edges. Although this is a correct interpretation of the rules given, the manual diagram reflects knowledge about the hardware configuration which is outside the knowledge used for control flow. This knowledge permits the elimination of certain events at certain nodes. Adding the outside knowledge to the preprocessor could be done in the future.

would have the edges from both originating nodes. However, this should not be applied to the nodes with the value *ANY* as it will change the meaning of the diagram. This will not affect many diagrams.

5.1.3 Robustness

Although the existing preprocessing code is sufficient for reading the current assembler file, improving the robustness of the code would decrease the likelihood of erroneous or non-functioning preprocessing code in the future. The preprocessor currently ignores comments and searches for new records. Any lines which the program does not know how to handle are ignored. Also, the record is expected to be broken up in a certain manner or else the preprocessing code terminates. The assembler is actually more lenient about how record content is spread across lines. To improve the parsing code, one could modify the preprocessing program so that it is aware of which lines to ignore and so that it accepts records in any format that the assembler would accept.

Another way of improving the robustness, involves determining to which node a loop goes back. The existing code correctly determines which tuples cycle. This correlates to a cycle in the graph. The current method of determining the loop-back node uses the name of the node, which may not be unique. Tuple information should be used instead.

5.1.4 Efficiency

An alternative method of building graphs from linked tuples may eliminate the need to merge trees and improve efficiency. The current system (see Section 2.4) processes a tuple and then recursively processes the tuples linked after that tuple. The alternative method would also traverse the tuples before that tuple. This approach requires special attention for graphs with cycles.

5.2 Code Generation from Graph Drawings

An improvement to the process would be to enable a user to generate the assembler code for a modified control structure by updating the diagrams with a graphical editor. Generating tuples for what is shown in the diagram would be fairly straight-forward, reversing the process used in preprocessing to create the graphs. The logic errors, which

are not on the diagrams, would have to be considered. These logic errors deal with the fact that the state ANY is treated like a wild card and restricts the possible actions. The tuples would then have to be ordered correctly. This would also require examining the respective graph editors.

A: EXAMPLES OF GRAPH DESCRIPTION FORMATS

See Section 3.6.2 for a brief verbal comparison. Figures 26, 27, 28, and 29 contain the information needed by **Davinci**, **Graphlet**, **Graphviz** (*dot*), and **VCG** respectively to create a graph drawing like that in Figure 25.

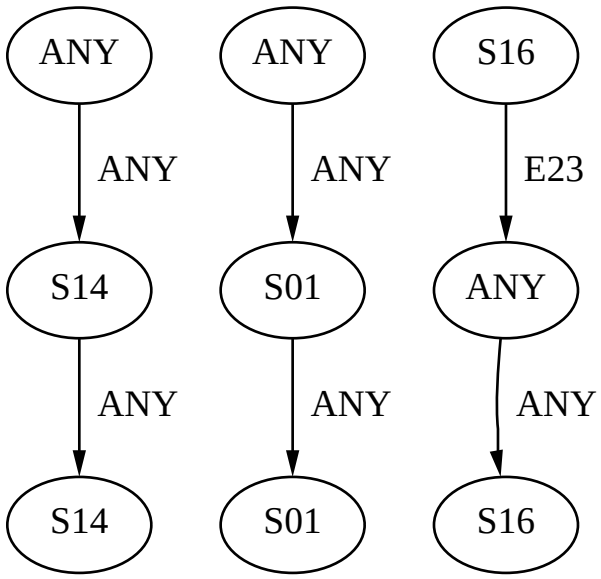

Figure 25: The graph, i1_00, associated with the examples of input text.

```
[
  l("3659", n("", [ a("OBJECT", "ANY"),a("_GO", "ellipse") ], [
    l("3659_3661", e("", [ a("_DIR", "none") ],
    l("3661", n("", [ a("OBJECT", "ANY"),a("_GO", "text") ], [
    l("3661_3660", e("", [],
      l("3660", n("", [ a("OBJECT", "S14"),a("_GO", "ellipse") ], [
        l("3660_3667", e("", [ a("_DIR", "none") ],
        l("3667", n("", [ a("OBJECT", "ANY"),a("_GO", "text") ], [
        l("3667_3666", e("", [],
          l("3666", n("", [ a("OBJECT", "S14"),a("_GO", "ellipse") ], []))
 )) ])) )) ])) )) ])) )) ]))
,
  l("3656", n("", [ a("OBJECT", "ANY"),a("_GO", "ellipse") ], [
    l("3656_3658", e("", [ a("_DIR", "none") ],
    l("3658", n("", [ a("OBJECT", "ANY"),a("_GO", "text") ], [
    l("3658_3657", e("", [],
      l("3657", n("", [ a("OBJECT", "S01"),a("_GO", "ellipse") ], [
        l("3657_3665", e("", [ a("_DIR", "none") ],
        l("3665", n("", [ a("OBJECT", "ANY"),a("_GO", "text") ], [
        l("3665_3664", e("", [],
          l("3664", n("", [ a("OBJECT", "S01"),a("_GO", "ellipse") ], []))
 )) ])) )) ])) )) ])) )) ]))
,
  l("3653", n("", [ a("OBJECT", "S16"),a("_GO", "ellipse") ], [
    l("3653_3655", e("", [ a("_DIR", "none") ],
    l("3655", n("", [ a("OBJECT", "E23"),a("_GO", "text") ], [
    l("3655_3654", e("", [],
      l("3654", n("", [ a("OBJECT", "ANY"),a("_GO", "ellipse") ], [
        l("3654_3663", e("", [ a("_DIR", "none") ],
        l("3663", n("", [ a("OBJECT", "ANY"),a("_GO", "text") ], [
        l("3663_3662", e("", [],
          l("3662", n("", [ a("OBJECT", "S16"),a("_GO", "ellipse") ], []))
  )) ])) )) ])))) ])) )) ]))
]
```

Figure 26: **DaVinci** mentions nodes and edges in a recursive manner for input.

```
graph [
   directed 1
   node [ id 3682 label "S14" graphics [  w 40.0  h 20.0  type "oval" ] ]
   node [ id 3680 label "S01" graphics [  w 40.0  h 20.0  type "oval" ] ]
   node [ id 3678 label "S16" graphics [  w 40.0  h 20.0  type "oval" ] ]
   node [ id 3676 label "S14" graphics [  w 40.0  h 20.0  type "oval" ] ]
   node [ id 3675 label "ANY" graphics [  w 40.0  h 20.0  type "oval" ] ]
   node [ id 3673 label "S01" graphics [  w 40.0  h 20.0  type "oval" ] ]
   node [ id 3672 label "ANY" graphics [  w 40.0  h 20.0  type "oval" ] ]
   node [ id 3670 label "ANY" graphics [  w 40.0  h 20.0  type "oval" ] ]
   node [ id 3669 label "S16" graphics [  w 40.0  h 20.0  type "oval" ] ]
   edge [ source 3669 target 3670 id 3671 label "E23" ]
   edge [ source 3672 target 3673 id 3674 label "ANY" ]
   edge [ source 3675 target 3676 id 3677 label "ANY" ]
   edge [ source 3670 target 3678 id 3679 label "ANY" ]
   edge [ source 3673 target 3680 id 3681 label "ANY" ]
   edge [ source 3676 target 3682 id 3683 label "ANY" ]
]
```

Figure 27: **Graphlet** uses a form of adjacency list for input.

```
digraph i1_00 {
n3666 [label = S14];
n3664 [label = S01];
n3662 [label = S16];
n3660 [label = S14];
n3659 [label = ANY];
n3657 [label = S01];
n3656 [label = ANY];
n3654 [label = ANY];
n3653 [label = S16];

n3653 -> n3654 [label = E23];
n3656 -> n3657 [label = ANY];
n3659 -> n3660 [label = ANY];
n3654 -> n3662 [label = ANY];
n3657 -> n3664 [label = ANY];
n3660 -> n3666 [label = ANY];
}
```

Figure 28: **Graphviz** uses a form of adjacency list for input.

```
graph: {
  title: "i1_00"
  layoutalgorithm: mindepthslow
  straight_phase: yes
  yspace: 20
  display_edge_labels: yes
  node.shape: ellipse
  node: { title: "3682" label: "S14" }
  node: { title: "3680" label: "S01" }
  node: { title: "3678" label: "S16" }
  node: { title: "3676" label: "S14" }
  node: { title: "3675" label: "ANY" }
  node: { title: "3673" label: "S01" }
  node: { title: "3672" label: "ANY" }
  node: { title: "3670" label: "ANY" }
  node: { title: "3669" label: "S16" }
  edge: { sourcename: "3669" targetname: "3670" label: "E23" }
  edge: { sourcename: "3672" targetname: "3673" label: "ANY" }
  edge: { sourcename: "3675" targetname: "3676" label: "ANY" }
  edge: { sourcename: "3670" targetname: "3678" label: "ANY" }
  edge: { sourcename: "3673" targetname: "3680" label: "ANY" }
  edge: { sourcename: "3676" targetname: "3682" label: "ANY" }
}
```

Figure 29: **VCG** uses a form of adjacency list for input. The input also includes information for guiding the layout algorithm.

B: ACKNOWLEDGEMENTS

Diane Souvaine guided and supported me. In addition, her enthusiasm and instruction encouraged me to explore technical areas, that I would have missed otherwise. Jack Howe provided the environment which prompted the research, taught me about it, and provided and verified information for the paper. Don Terry, a co-worker at IBM, pushed the idea of automating this process, and provided moral support throughout graduate school. Theresa Biedt, a fellow graduate student, gave me some guidance in the field of graph drawing. I would like to thank NEC for providing a working environment that nurtured my completing this paper. People at NEC who were especially helpful include Rich Kelsey, Steve Omohundro, Henry Cejtin, and Mary Zweibel.

References

[Chv83] V. Chvatal. *Linear Programming.* W. H. Freeman, New York, 1983.

[DETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994. file://ftp.cs.brown.edu/pub/papers/compgeo/gdbiblio.ps.gz.

[Ead84] P. Eades. A heuristic for graph drawing. *Congr. Numer.*, 42:149–160, 1984.

[EW86] P. Eades and N. Wormald. The median heuristic for drawing 2-layers networks. Technical Report 69, Department of Computer Science; University of Queensland, 1986.

[FW94] M. Frhlich and M. Werner. The graph visualization system davinci - a user interface for applications. Technical Report 5/94, Department of Computer Science; Universitt Bremen, September 1994. ftp://ftp.uni-bremen.de/pub/graphics/daVinci/papers/techrep0594.ps.gz.

[GKNV93] E. R. Gansner, Eleftherios Koutsofios, Stephen C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19:214–230, 1993. http://www.research.att.com/sw/tools/graphviz/TSE93.ps.gz.

[PEW86] B. McKay P. Eades and N. Wormald. On an edge crossing problem. In *Proceedings 9th Australian Computer Science Conference*, pages 175–198, 1986.

[San95] Georg Sander. Graph layout through the VCG tool. In Roberto Tamassia and Ioannis G. Tollis, editors, *Graph Drawing (Proceedings 1994)*, volume 894 of *Lecture Notes Computer Science*, pages 194–205. Springer-Verlag, 1995. ftp://ftp.cs.uni-sb.de/pub/graphics/vcg/doc/tr-A03-94.ps.gz.

[SM91] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-21(4):876–892, July 1991.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109–125, February 1981.

[Tam97] Roberto Tamassia. Graph drawing. In Jacob E. Goodman and Joseph O'Rourke, editors, *CRC Handbook of Discrete and Computational Geometry*. CRC Press, 1997. http://www.cs.brown.edu/cgc/papers/t-gd-97.ps.gz.

[War77] John Warfield. Crossing theory and hierarchy mapping. *IEEE Transactions on Software Engineering*, 7:502–523, 1977.