Swan — A Data Structure Visualization System

by

Jun Yang

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

Dr. Clifford A. Shaffer, Chairman

Dr. Lenwood S. Heath     Dr. Edward A. Fox

May, 1995

Blacksburg, Virginia
Swan — A Data Structure Visualization System

by

Jun Yang

Committee Chairman: Dr. Clifford A. Shaffer

Computer Science

(ABSTRACT)

Swan is a data structure visualization system. Using Swan, a C or C++ program can be annotated to provide multiple views of the data structures used in the program so that they can be better understood by the viewer. Swan allows visualization to be a two-way communication process between the viewer and the program. Several automatic graph layout algorithms are implemented in Swan for layout of various data structures. Automatic layout allows the annotator to concentrate on the logical views of data structures without worrying about their graphical display. Swan serves as a data structure visualization system, a graphical debugging tool, and a platform for experimenting with graph layout algorithms.
ACKNOWLEDGEMENTS

The majority of my thanks go to Dr. Clifford A. Shaffer, who not only suggested this project but also helped explore solutions to the problems during the development process of Swan. I also appreciate his general support during my study. I extend my thanks to Dr. Lenwood S. Heath for his helpful suggestions and thoughtful criticisms to the design and implementation of Swan. I thank Dr. Edward A. Fox for his guidance, advice, and help in general.

I owe my gratitude to my colleague, David Hines, who helped me debug and test the system. I want to thank all the students participated in the Swan usability testing which makes me confident in the future of Swan. I also thank all the members in Project GeoSim for their excellent product Gil on which the graphical user interface of Swan is based.

Finally, I thank my family and friends. This research would not have been possible without their support and encouragement.

This thesis is dedicated to my parents, Deyuan Yang and Qiming Wang.
# TABLE OF CONTENTS

1 Introduction ........................................... 1
   1.1 Program Visualization ............................ 1
   1.2 Swan ........................................... 3
   1.3 Related Work ................................... 7

2 System Design ......................................... 14
   2.1 Overview ....................................... 14
   2.2 The Swan Kernel ................................. 16
   2.3 Process Control ................................ 20

3 The Swan Viewer Interface (SVI) ..................... 24
   3.1 Overview ....................................... 24
   3.2 SVI Functionalities and Implementations ....... 24

4 Swan Annotation Interface Library (SAIL) ........ 28
   4.1 Basic Elements ................................ 29
   4.2 SAIL Functions ................................ 30
   4.3 Annotation Techniques ......................... 35

5 System Implementation ............................... 41
   5.1 Overview ....................................... 41
   5.2 Fundamental Classes ............................ 42
   5.3 Graph Layouts .................................. 53
   5.4 Graph Display .................................. 53
   5.5 Control of Graphical Attributes ............... 54
6 Graph Layout Algorithms
   6.1 Introduction .................................................. 57
   6.2 Swan’s Graph Layout Algorithms .......................... 58

7 Usability Testing ............................................. 72

8 Conclusion ..................................................... 75

A Test Questions ................................................. 77

B Questionnaire and Collected Data .......................... 78

References .......................................................... 81
# List of Figures

1.1 A program visualization model ............................................. 2
1.2 Two views of a graph created in an annotated minimum spanning tree algorithm 4
1.3 The Swan visualization model ............................................. 6
1.4 Genealogy of program visualization systems .......................... 11

2.1 Three main components of Swan ......................................... 14
2.2 The architecture of Swan .................................................. 16
2.3 The state diagram of Swan annotated program ....................... 21
2.4 The tri-state transitions in Swan ........................................ 22

3.1 Modifying graphical attributes in Swan .............................. 27

5.1 The global graph list gg1 .................................................. 42
5.2 The structure of class swGraph .......................................... 44
5.3 Structure nb-node .......................................................... 45
5.4 The structure of class swNode ............................................ 46
5.5 The structure of class swEdge ............................................ 47
5.6 A graph $G$ and its representation in Swan ......................... 48
5.7 The layout component structure in Swan .............................. 51
5.8 The structure of class swLC .............................................. 52

6.1 A hierarchy of linked lists created in Swan ......................... 60
6.2 A binary search tree ..................................................... 63
6.3 A layout of a graph by using Kamada and Kawai's algorithm .... 64
6.4 A hierarchical layout of a directed graph .......................... 71
Chapter 1

Introduction

1.1 Program Visualization

Computer program listings are often difficult to understand. Computers equipped with powerful graphics capabilities have made it possible to generate and view graphical representations of programs, through program visualization [41]. Essentially, program visualization illustrates certain aspects of a program's static structure or its run-time execution using computer graphics techniques.

Several successful program visualization systems exist, including BALSA [8], Pavane [40], PROVIDE [28], TANGO [44], and ZEUS [11]. Most of them are used for teaching, presentation, and debugging purposes. In the context of advanced computer systems, program visualization is also used for performance monitoring and tuning [48].

1.1.1 Taxonomies

Different taxonomies of program visualization systems have been proposed [41]. Shu [42] focuses on the increasing complexity of program visualization, from pretty-printing to algorithm animation. Myers [34] classifies systems according to the program structure that is illustrated (code, data, algorithm) and the display style (static or dynamic). Stasko and Patterson [45] give a four-dimensional classification: aspect, abstractions, animation, and automation.

A comprehensive and systematic taxonomy is proposed by Roman and Cox [41]. They define program visualization as a mapping, or transformation, of a program to a graphical representation. Their definition suggests the model shown in Figure 1.1 in which program
CHAPTER 1. INTRODUCTION

visualization is the result of interactions among three participants: the programmer who develops the original program, the animator who defines and constructs the mapping, and the viewer who observes the graphical representation.

![Diagram](image.png)

Figure 1.1: A program visualization model

According to their model, five criteria are selected to categorize a program visualization system:

Scope

The aspect of a program being visualized: code, data, control states, and execution behavior. Visualization systems often limit their scope to a subset of these program aspects.

Abstraction

Information conveyed by the visualization. Abstraction levels of the concepts presented in the graphical form may vary from straight-forward representation, such as highlighted code, to a more abstract level, such as a control flow graph.

Specification Method

The method used by the animator to construct the visualization. Some systems provide “hard-wired” mapping; others allow arbitrary definition of the mapping. Some require modification of code; others do not.
CHAPTER 1. INTRODUCTION

Interface

Facilities provided by the system for visual presentation of information. They include tools used by the animator as well as the interface used by the viewer to investigate the presented information.

Presentation

Mechanisms and heuristics used by the animator to express the semantics of a visualization.

1.2 Swan

Swan is a data structure visualization system. Its main purpose is to allow the user to visualize the data structures and the basic execution process of a C/C++ program. Swan is specially designed to support visualization of programs implementing graph algorithms, including trees, lists and arrays.

As a part of the NSF CISE Institutional (Education) Infrastructure (Grant CDA-9312611) project at Virginia Tech, Swan will be used in two ways: by instructors as a teaching tool for data structures, and by students to animate their own programs and to understand how and why their programs do or do not work.

To use Swan, a program must first be annotated, then compiled and linked with the Swan Annotation Interface Library (SAIL). Then the viewer can run the annotated program.

To annotate a program, the annotator should have a clear understanding of its data structures. Different views (i.e., graphs) of the data structure can be constructed by calling functions in SAIL. The annotator is responsible for specifying the logical structure and the semantics of these graphs; Swan does not assume any responsibility for analyzing any specific data structure of the annotated program. Swan only accepts requests to create, display and maintain graphs. The annotator does not need to control the graphical display
CHAPTER 1. INTRODUCTION

of these graphs although he has full power to decide most visual attributes of the graphs.

To run an annotated program, the viewer simply runs the program and investigates the views rendered in the Swan display window. The viewer has the capability to modify not only the graphical attributes of the graphs, but also the logical structure of the graphs when such modifications have been supported by the annotator.

SAIL is a library of functions which can be used to add visualization to any C/C++ program. SVI is a window environment in which the viewer can see the resulting visualizations (Figure 1.2).

Figure 1.2: Two views of a graph created in an annotated minimum spanning tree algorithm

The programmer, annotator and viewer all play separate roles in this visualization process. In practice, they may all be the same person, which makes Swan a potential tool for
CHAPTER 1. INTRODUCTION

high-level visualized debugging.

Currently, two versions of **Swan** are available: one for UNIX systems with the X Window system installed and one for MS-DOS.

1.2.1 Swan’s Visualization Model

If we apply Roman and Cox’s taxonomy [41] to **Swan**, its general characteristics are:

**Scope:** data structures

**Abstraction:** graphs

**Specification:** annotation

**Interface:** SAIL for the annotator to design the views and SVI for the viewer to investigate the annotated program

**Presentation:** textual interpretation and visual events

Although **Swan**’s visualization model is based on Roman and Cox’s model, it possesses some special properties such as the emphasis on the close cooperation among the participants and a two-way communication mechanism between the annotator and the viewer as shown in Figure 1.3.

In the general model, the three participants play different roles to make the visualization process complete. Most program visualization systems regard the participants as different people and they do not have direct interactions. Programs and their graphical representations link them together. Usually, the consequence is that the animation mechanisms become so complicated that only the animator is interested in and able to use it. Of course, it does not hurt if both the programmer and the viewer are not involved in the animation process at all.

Because of **Swan**’s educational purpose, users of **Swan** are likely to play all three parts themselves. This is reflected in Figure 1.3 as three overlapping circles. For example, a student writes a program, uses **Swan** to animate it and checks whether it works as expected.
CHAPTER 1. INTRODUCTION

Figure 1.3: The Swan visualization model

In the general model, program visualization is a one-way information passing process: a program is written by the programmer, then animated by the animator, and finally graphical representations are viewed by the viewer. In Swan, the information not only can be passed from the annotator to the viewer in the form of graphical representation of data structures, but also can be passed from the viewer to the annotator in the form of modification requests. Although this cannot be regarded as a complete two-way communication between the annotator and the viewer because of the unequal status of the two participants (i.e., the annotator has complete control of the views to be constructed while the viewer can only modify the program’s data structures under the restrictions imposed by the annotator), it provides a powerful mechanism to encourage the viewer to be more active in exploring the program and gaining new insights. This capability makes Swan different from most algorithm animation systems in which the viewer can only watch the animation passively. We believe it not only makes Swan more suitable to its educational purpose, but also shows Swan’s potential as a graphical debugging tool at the abstract level.
CHAPTER 1. INTRODUCTION

1.3 Related Work

Two areas of work have motivated and influenced the present research on Swan. We discuss each of them in turn.

1.3.1 Data Structure Display

Several existing systems automatically generate a static graphical display of the data structures in a program. The advantage of these systems is that data structures in an arbitrary program can be viewed without modifying the original program, but the disadvantage is that their generic representation does not necessarily convey how the data are really used [9]. Brief descriptions of several systems follow.

Incense

Myers [32, 33] developed Incense on the Alto personal computer at Xerox PARC in early 80s. It is a system written in and for the Pascal-like language Mesa. Incense can automatically generate pictures for any data type during the execution of an actual Mesa program. This is done in real time without modifications to the source program. The user needs only specify at debug time the name of a variable to get a full pictorial display generated in a rectangular region. Unfortunately, user-defined graphical representation of data types, which would be a very useful functionality as claimed by Myers, was not implemented at that time. Also, the system was not integrated with any standard debugger, which limited its usefulness.

PROVIDE

Moher’s source-level Process Visualization and Debugging Environment (PROVIDE), was developed at the University of Illinois at Chicago [28]. It uses multiple Macintoshes as graphical front ends to a VAX 11/789. The program being animated is compiled on the VAX and run on the Macs, and extensive execution traces are stored on disk. The user then
CHAPTER 1. INTRODUCTION

observes execution through the Macs and controls what part of the program’s execution to view by making queries of the database.

One important contribution of PROVIDE is the ability to modify the view of program execution without re-entering the edit-compile-execute cycle. In PROVIDE, pictorial objects may be created or removed at any time during a program’s execution. Modifying the illustration of execution does not require re-translation or re-execution overhead.

Another interesting feature in PROVIDE is graphical editing, which allows the user to manipulate pictures directly to affect the values of the objects which drive the picture’s representation. For example, if an array is displayed as a histogram, in which the height of each rectangle is proportional to the value of the corresponding element in the array, an editing handle can be activated on top of each rectangle. The user can pull these handles up and down to modify the values of the array’s elements directly. These new values are effective in subsequent executions.

1.3.2 Algorithm Animation Systems

*Sorting Out Sorting*, a 30 minute 16mm color film, is a well known early effort to illustrate algorithms graphically [2]. Because of hardware limitations, early algorithm animation systems were not interactive [1, 2]. The introduction of BALSA in 1984 marked the emergence of the current generation of program visualization techniques and the first serious attempt to organize the task of constructing complex, customized visualizations [41].

BALSA

The common approach today for animating algorithms specified in high-level procedural languages was pioneered in BALSA — Brown Algorithm Simulator and Animator [8]. Programs are annotated with user-defined *interesting events*, which cause changes to the displayed image, called a view. Each view controls a window on the screen that is notified when one of the specified events happened in the algorithm. A view is responsible for updating its graphical display appropriately based on the incoming event information. Views
CHAPTER 1. INTRODUCTION

can also return information from the user to the algorithm. For example, if a binary tree algorithm has an input module for the user to specify which node to delete by pointing at it with a mouse, the view must be able to map a point on the screen into a coordinate system meaningful to the view and the input module. In addition, BALSA is the first program visualization system used as a teaching aid in introductory computer science courses.

BALSA-II

BALSA-II is a direct descendent of BALSA [10]. The concept of interesting events introduced in BALSA is separated into algorithm output events and algorithm input events. BALSA-II provides scripting facilities: instead of recording every keystroke or mouse activity, higher level information is stored, enabling the user to switch between passive viewing and active exploration. One of BALSA-II's drawbacks is that it requires Macintosh coding to create new animation views which makes it tied to the operating system.

Zeus

Zeus, another descendent of BALSA, includes many of its basic capabilities as well as a number of innovative features [11]. Multiple algorithm views can be constructed. Zeus is noteworthy for its use of objects, strong-typing, and parallelism. Zeus can automatically generate utility views based on the set of interesting events of the investigated algorithm. Two of Zeus' utilities are very useful for debugging algorithms [48]:

1. The transcript view contains typescript that displays each generated event as a symbolic expression.

2. Zeus provides a control panel containing a button for each event, with appropriate widgets for specifying additional parameters. Clicking on any one of these buttons causes an event to be generated with the specified parameter values.

With the support of these utility views, one can develop (and debug) a view before and even without ever implementing an algorithm [11].
CHAPTER 1. INTRODUCTION

Tango

Tango (for “transition-based animation generation”) is an algorithm animation system developed at Georgia Tech [43, 44]. Based on the path-transition paradigm proposed by Stasko, Tango focuses on generating smooth movement of objects. An animator producing an animation in this paradigm creates, manipulates, and edits instances of four abstract data types: images, locations, paths, and transitions. An image is a graphical object that undergoes changes in characteristics such as location, size, and color, during an animation. A location is a position identified by an \((x, y)\) coordinate pair in the animation coordinate system. The path and transition data types help the animator easily interpolate between animation states.

Tango also implemented a finite-state transducer, which can map program events to display events. Thus, the same program event can be mapped to different display events at different times, depending on the state of the transducer.

Lens

Lens is a program visualization system seeking to bridge the two domains of data structure display and algorithm animation [31]. It integrates an algorithm animation system with a standard debugger, DBX [30]. Lens is aimed at providing application-specific animation views for debugging purposes. Users can design their own animations without graphics programming. Once the animations are built, the user can execute the program and see the animation by choosing the “Run” command. Lens then pops up an animation window and displays the animation. Lens uses the routines from the Tango system to generate the animations it presents.

In addition to the systems mentioned above, there are many others which made significant contributions to the development of program visualization [7, 15, 16, 29, 36, 37, 40]. The hereditary relations of representative program visualization systems is shown in Figure 1.4.
Figure 1.4: Genealogy of program visualization systems
CHAPTER 1. INTRODUCTION

1.3.3 Summary

In the NSF Educational Infrastructure project, we want to provide a visualization tool to be used by both instructors and students in data structure or algorithm classes. The tool should be easy to use so that instructors and students do not spend much time on the annotation process. Usually, people do not want to spend more time working on creating the visualization of a program than on the program itself. Also, most students in these classes are not experienced programmers. Therefore, the tool should be simple so that students can have hands-on experience in the annotation process, that we believe is more effective for them to understand the fundamental data structures used in the program than simply watching the available algorithm animations. We also want the tool to be used to create application-specific views for different programs. Once the students understand how to use the tool, they can use it to work on different programs and do not need to learn many other tools.

We studied most of the available program visualization systems as discussed earlier in this section. Most data structure display systems do not modify the source code and do not require coding. They are useful in testing and debugging software. They are general enough to be applied on different programs. However, they cannot provide application-specific views. Algorithm animation systems can be used to design application specific views which are useful for program understanding and teaching [31]. But their complicated animation processes make it very difficult for novice programmers to get involved. Integrating data structure display systems and algorithm animation systems would make application-specific visual debugging possible. Lens is a pioneer system in this direction emphasizing generation of good animations while debugging. But it hides the annotation process behind its interactive tool. The user of the system cannot get involved in the underlying process. Finally the necessity of building a new system for our specific purposes is confirmed.

Swan focuses on annotation simplicity, ease-of-use, and flexibility. Several approaches
CHAPTER 1. INTRODUCTION

used in Swan distinguish it from most other data structure display systems:

1. **Swan** provides a compact annotation interface library. There are less than 20 frequently used library functions.

2. **Swan**’s user interface is simple and straight-forward.

3. The annotator decides the semantics of the views, i.e., with which variables the graphical elements in the views are associated, and controls the progress of the annotated program, which makes annotation more flexible.

4. **Swan** provides automatic layout of a graph so the annotator need only concentrate on the graph’s logical structure. When necessary, the annotator is able to manually layout the graphs constructed in **Swan** and control their display attributes.

The design of **Swan**’s architecture is discussed in Chapter 2. The **Swan** Viewer Interface and **Swan** Annotation Interface Library are described in Chapters 3 and 4, respectively. Implementation details can be found in Chapter 5 followed by a description of **Swan**’s graph layout algorithms in Chapter 6.
Chapter 2
System Design

2.1 Overview

Swan is a data structure visualization system. Swan has three main components according to its visualization model (Figure 2.1): Swan Annotation Interface Library (SAIL), Swan Kernel, and Swan Viewer Interface (SVI). SAIL is a small set of easy to use library functions that allow the annotator to design different views of a program. SVI allows a viewer to explore Swan annotated programs. Swan Kernel is the main module in Swan. It is responsible for constructing, maintaining, and rendering all the views generated through SAIL library functions. It accepts viewer’s requests through SVI and takes appropriate actions. It is also the medium through which the annotator communicates with the viewer. The framework of Swan is shown in Figure 2.1.

![Diagram of Swan components]

Figure 2.1: Three main components of Swan

The functionalities of SVI and SAIL will be elaborated in Chapters 3 and 4, respectively. Design details of the Swan Kernel are discussed in the following sections of this chapter.
CHAPTER 2. SYSTEM DESIGN

2.1.1 Concepts

In general, data structure visualization is a mapping of the data structures used in a program to a graphical representation (Figure 1.1). In Swan, visualization particularly refers to a graphical view of a data structure used in a program as well as the abstraction represented by the data structure. For example, two views of a graph can be built in the annotated minimum spanning tree algorithm (Figure 1.2): The view on the right is an adjacency list representation of the graph, a visualization of the physical implementation used by the annotated program. The view on the left shows the logical topology of the graph, an abstraction represented by the adjacency list. These two kinds of views of data structures coexist in Swan in a consistent form, called a graph.

Because Swan is a data structure visualization system specially designed to support various graph algorithms in which graphs are the most frequently referenced objects, it is natural that graphs are chosen as the basic elements in Swan. The concept graph in Swan has specific implications. Graphs are frequently mentioned in this thesis. The reader should be careful to distinguish between Swan's graph representation, and the graph data structure appearing in annotated programs.

All views in Swan are composed of Swan graphs. To keep Swan simple, a graph is the only element that the annotator uses to construct different views. A Swan graph has a set of nodes and edges. A Swan graph is defined by the annotator via its nodes and edges. A Swan graph has default display attributes for its nodes and edges, which are used by Swan to render the corresponding graphical objects. Nodes and edges can have their own individual display attributes which can override the graph's default values.

2.1.2 Architecture

The complete architecture of Swan is shown in Figure 2.2. The annotator uses SAIL to annotate the program; the viewer investigates the annotated program through SVI. Other modules in Figure 2.2 belong to the Swan Kernel, except the Geosim Interface Library
CHAPTER 2. SYSTEM DESIGN

(GIL) [5] which is the graphics library used by Swan. GIL has versions available for the X-Window System, DOS, and Mac OS. Thus, Swan may be ported to these platforms without rewriting the graphical display module interface. The graph layout module will be described in Chapter 6. Other modules in the Swan kernel are discussed in the following section.

![Diagram of Swan architecture]

Figure 2.2: The architecture of Swan

2.2 The Swan Kernel

The Swan Kernel is the core of the system. It is not only responsible for all processing related to graphs constructed in the annotated program, but also the hub of communication between the system, the annotated program and the viewer. The Swan kernel has four modules: Swan Logical Layer, Swan Physical Layer, Swan Event Handler, and the Graph Layout Algorithms module.
CHAPTER 2. SYSTEM DESIGN

The logical structures of graphs built by the annotated program are stored in the **Swan** Logical Layer. After appropriate layout algorithms are applied, a physical representation of the layout is kept in the **Swan** Physical Layer. The separation of a graph's representation into a logical layer and a physical layer makes **Swan** adaptable to changes in its graphics display toolkit (**GIL**) and also portable to other graphics platforms. Graph layout algorithms used in **Swan** only need to determine the graphs' topology without worrying about details of visual attributes of the graphs.

### 2.2.1 Swan Logical Layer

The **Swan** Logical Layer contains all the internal representations of graphs created by the annotated program. For each graph, a standard adjacency list representation is used to store all the graph's nodes and edges. One drawback of the adjacency list representation is the redundancy when an undirected graph is represented. Also, the dynamic allocation of memory costs extra time. It is expected to support different kinds of graphs, from a simple array or linked list to a complicated general directed graph. The adjacency list is a simple way to represent all these graphs in a consistent form. In practice, the time required by dynamic memory allocation is not significant when the program is running on a workstation and the graph is not very large.

### 2.2.2 Swan Physical Layer

Every **Swan** graph has physical attributes, that affect its graphical display. The most important attribute is the position of the graph and positions of all of the nodes and edges in this graph, that is, the layout of the graph. To decide these positions is to layout the graph. Several graph layout algorithms have been implemented in **Swan** to deal with different graphs types so that the annotator does not need to spend much time on layout himself. This process is covered in Chapter 6.

Other physical attributes of a graph include: default values for node attributes (such as color, shape, line thickness, and bounding box), and default values for edge attributes (such
CHAPTER 2. SYSTEM DESIGN

as color, and line thickness).

Before a graph is displayed, its layout must be determined. This is executed by the Swan Physical Layer automatically. Then Swan display functions are called to render graphical objects in the Swan display window. These display functions belong to the Swan Physical Layer and are based on GIL functions.

2.2.3 Swan Event Handler

The Swan Event Handler is the center of process control in Swan. Events generated by SVI due to the interactions between the viewer and Swan are sent to the Swan Event Handler. When the handler has control of the process (Section 2.3.1), these events are analyzed and actions are taken by the handler accordingly.

The events processed by the Swan Event Handler can be divided into five groups:

- window operations
- requests to modify the physical attributes of a graph
- requests to re-layout graphs
- requests to modify the logical structure of a graph
- requests to change the state of the running process

Window Operations

Because SVI is a simple graphical user interface, only a few fundamental operations are supported.

Panning: If several graphs are displayed at the same time, the viewer may not be able to see all of them due to the limited display area. The panning operation allows the viewer to be able to explore all the graphs by moving the viewed area.
CHAPTER 2. SYSTEM DESIGN

Zooming: Zooming allows the viewer to investigate the graph by viewing details of graphs at different levels. If the view is zoomed in, details of a graph can be seen more clearly. Conversely, if the view zoomed out, the whole structure of the graph is exhibited.

Picking: More information about a node or an edge can be retrieved by a pick operation. Using the mouse to click on a node or an edge, a small window will pop up to show the identity and label of the node or edge.

Modify the Physical Attributes of a Graph

The physical attributes of a graph can be modified if requested by the viewer. After the event handler accepts these requests, it will notify the Swan physical layer to make corresponding changes, then the graph is displayed again with the new graphical attributes. Details of these modifiable attributes can be found in Section 3.2.4.

New Graph Layout

The new graph layout request can be originated either directly from the viewer, or due to certain modifications on the physical attributes of a graph which may force the layout to change accordingly. For example, if the size of the bounding box of a node is changed, the graph may need to be laid out again to avoid edges crossing the node.

Modify the Logical Structure of a Graph

Swan provides a mechanism so that the logical structure of a graph can be modified during execution of the annotated program which makes it possible for the viewer to modify the data loaded in the annotated program interactively, without re-starting the program with another set of input data. This is an important characteristic of Swan because it establishes a two-way communication between the annotator and the viewer. The annotator allows the viewer to modify the logical structure of a graph as follows:

- The Swan graph editing menu must be enabled by the annotated program.
CHAPTER 2. SYSTEM DESIGN

- The annotated program provides subroutines to modify the graph according to the information received from the Swan Event Handler.

- The viewer modifies the logical structure of a graph through the graph editing menu in the Swan control panel.

- The Swan Event Handler is responsible for notifying the annotated program what the viewer’s requests are.

Change the State of the Process

There are two states in an annotated program’s running process: Run and Step. The viewer can change the state of the process from one to the other. See Section 2.3.3 for more details.

2.3 Process Control

To illustrate the dynamic properties of the Swan system, we describe the execution of a Swan annotated program in this section.

2.3.1 Control Switching

A Swan annotated program runs as a single thread process. The events generated from the Swan Viewer Interface are stored in an event queue. Initially the annotated program has control of the process. Whenever a SAIL function is invoked, Swan will process all events in the event queue. At this point, Swan’s Event Handler takes control. After the SAIL function completes, control is returned to the annotated program.

2.3.2 State Diagram

The state transition diagram of a Swan process is shown in Figure 2.3. Circles in the diagram represent states. Arc labels are the events which cause state transitions. RUN
and **STEP** refer to the events generated by the viewer clicking the **RUN** or **STEP** buttons, respectively.

![State Diagram of Swan](image)

Figure 2.3: The state diagram of **Swan** annotated program

After a program is started, it enters state **S₀**. To initiate the **Swan** environment, **sw_init** must be called. **sw_init** must be the first **Swan** function call in the annotated program.

There are three basic states in **Swan** when it is active: **Run**, **Step** and **Pause**. Transitions among these states are the main activities in **Swan**. Details of this part will be discussed in Section 2.3.3.

If function **sw_quit** is invoked sometime in the process, the **Swan** system will clear all objects created by the annotated program and quit. The process will continue running as a normal process without **Swan** annotation.

### 2.3.3 State Transitions

State transitions among **Run**, **Step** and **Pause** are the main activities in the running process of an annotated program.

Essentially, the process may run continuously (i.e., in **Run** state) or step by step (i.e.,
in **Step** state). "Step" here refers to the execution of a code segment ending at the next breakpoint set by the **SAIL** function **sw.break**. The size of each step varies, depending on the length of the code segment between two breakpoints. **Swan** lets the annotator decide the size of the step because it is difficult, if not impossible, for **Swan** to identify the interesting events in the annotated program. We also want the annotator to have more flexibility to define the behavior of the annotated program.

The process can change from **Run** state to **Step** state if the **SAIL** function **sw.step** is invoked from within the annotated program, or the button **STEP** on the control panel is clicked. Conversely, the process can change from **Step** state to **Run** state if the **SAIL** function **sw.run** is invoked from within the annotated program, or the button **RUN** on the control panel is clicked (Figure 2.4).

If the process is in **Step** state and the **SAIL** function **sw.break** is invoked, the process will enter **Pause** state. In **Pause** state, the process will wait for the viewer to click either the **STEP** button or the **RUN** button to continue running in either **Step** state or **Run** state. Because either of these two buttons can be disabled in the annotated program, the annotator should be careful not to disable both of them at the same time in case the process
CHAPTER 2. SYSTEM DESIGN

runs into Pause state and can never be activated again. The reason Swan does not force at least one of the two buttons to be active is that the annotator may want the annotated program to keep running during some portion of the program without interference from the viewer.
Chapter 3
The Swan Viewer Interface (SVI)

3.1 Overview

The viewer interacts with an annotated program through the Swan Viewer Interface (SVI) as shown in Figure 1.2. The SVI main window contains a control panel and three child windows: the display window, I/O window and location window. The display window contains the graphs output by Swan. The I/O window is used by the annotator and Swan system to display one-line messages and get input from the viewer. The coordinates of the current position of the cursor in the display window are shown in the location window.

3.2 SVI Functionalities and Implementations

3.2.1 Picking

The viewer can pick a node or an edge in the Swan display window to get more information about it.

When the viewer clicks in the Swan display window, the coordinates of the selected point are recorded. All nodes and edges currently displayed are searched to determine whether the selected point is within a node or close enough to an edge. If so, the node or the edge is picked. A popup window appears near the picked node or edge to show information associated with it.

It is not efficient to search every node and edge in the displayed graph. Swan uses the concept of graph layout components (LC) (Chapter 5) to limit the search. It is easy to determine which node is picked by going downward through the LC tree, because every LC has a bounding box. If the clicked point is not within an LC’s bounding box, we don’t need
CHAPTER 3. THE SWAN VIEWER INTERFACE (SVI)

to check the nodes within that LC.

The situation is different for edges. Because edges may connect nodes in different LC’s, we must visit every edge by traversing the edgelist in the graph (Chapter 5).

3.2.2 Panning and Zooming

On the upper right hand of the main window are eight buttons. Six of them are used to pan and zoom graphs in the display window. The corresponding functions of each button should be self-evident through its label. The top four are for panning in the indicated directions. The next two buttons allow the viewer to control the viewing scale by zooming in or out.

3.2.3 Process Control

The three buttons on the lower right hand corner of the main window are used to control the running process of the annotated program. Clicking RUN will make the program run in continuous mode. STEP will make the program run in step mode. When the program is running in step mode, it stops at any break point set by the annotator, waiting for the viewer to push STEP to run in step mode or RUN to change to continuous mode in which the break points will be ignored. A green frame will appear around the button when the process is running in its corresponding mode. The annotator can disable one or both of these buttons. A disabled button has no effect on the program if it is clicked. QUIT can be used to quit the running process of the annotated program. The annotator cannot disable QUIT. Implementation details of process control can be found in Section 5.7.

3.2.4 Modifying Graphical Attributes

Every graph, node and edge created in Swan has a set of graphical attributes. For a graph, attributes include default graphical attributes for its nodes and edges and its layout method. For a node, attributes include type, color, size and line thickness. For an edge, attributes include color and line thickness.
CHAPTER 3. THE SWAN VIEWER INTERFACE (SVI)

To modify graphical attributes, the viewer can click the **Attributes** button on the top left corner of the **Swan** main window. A popup menu will appear. It has four items: **Graph Attributes**, **Node Attributes**, **Edge Attributes** and **Global Attributes**. The viewer can select any one of these items to modify its graphical attributes accordingly. For example, if the viewer selects **Graph Attributes**, **Swan** will display a message in the **I/O window** that asks the viewer to pick a graph from the graphs in the **Swan** display window. The viewer can pick a graph by clicking in the area it covers in the **Swan** display window. A popup window containing all the modifiable graphical attributes of the picked graph appears (Figure 3.1). The viewer can change the attributes to the ones he prefers. Finally, the button **OK** in the popup window is clicked to confirm all modifications, and the graph is redrawn using the new set of attributes. Otherwise, if the button **Cancel** is clicked, all the modifications are canceled. Graphical attributes of nodes and edges can be modified in a similar way.

3.2.5 Graph Editing

The viewer can interactively modify the logical structure of a **Swan** graph. **Swan** allows insertion or deletion of nodes and edges. The annotator can enable or disable any of these functions. An editing function is effective only when it is enabled.

To edit a graph, click the button **Edit** on the upper left corner of the main window. A popup menu appears. It has four items: **Insert Node**, **Delete Node**, **Insert Edge**, and **Delete Edge**. They correspond to the four graph editing functions mentioned above. Select one of these items to initiate the corresponding function.

Because the annotator has sole control of a graph's logical structure, any modification to a graph's logical structure must be implemented by the annotator. Therefore, the annotator can determine the semantics of the above four editing functions. In most cases, the annotator provides functions corresponding to the labels of those menu items, but he has the choice to provide functions that may have nothing to do with insertion or deletion of nodes and edges. The viewer needs to be careful while editing to make sure to follow the
CHAPTER 3. THE SWAN VIEWER INTERFACE (SVI)

Figure 3.1: Modifying graphical attributes in Swan

instructions in the Swan I/O window given by the annotator. See Section 5.6 for more details.

3.2.6 Graph Layout Saving and Restoring

While executing the annotated program, the viewer may be interested in a particular graph layout and want to keep it for future reference. This is supported in Swan by copying the graph structure to a disk file. Saving a graph layout means saving the physical layout of the graph and all the graphical attributes associated with it so that when the layout is restored, the graph will look exactly as when it was saved.
Chapter 4

Swan Annotation Interface Library (SAIL)

The Swan Annotation Interface Library (SAIL) is a set of easy to use functions for annotating a program so that its significant data structures and the manner in which the data structures change during the execution of the program can be visualized. Given an appropriate description of the data structures used in a program, Swan is able to display them using different graphical elements as specified by the annotator.

The annotator should have a good understanding of the data structures used in the program before designing the views of these data structures. A view is a graph in Swan which consists of a set of nodes and a set of edges. The semantics of nodes and edges in the graph are decided by the annotator, i.e., the annotator decides what structures these nodes and edges represent.

Graphs can be used to represent either the actual data structures in the program or the logical concepts implied by those data structures. For example, consider a C program to find a minimum spanning tree in a graph $G$. $G$ is stored in the program using adjacency lists. To annotate this program, two views can be constructed. A view of the adjacency list is a direct representation of the actual data structure used in the program. The logical view of $G$ is an undirected graph (Figure 1.2).

Every graph constructed in Swan has a logical topology and a physical representation (i.e., layout). The graph's logical topology is determined by the nodes and edges in the graph. The annotator decides what nodes and edges should be in this graph and their semantics.

The layout of a graph is a drawing of the graph on a 2-dimensional surface. Specifically, it is an assignment of Euclidean coordinates to the nodes and edges on the layout plane in
CHAPTER 4. SWAN ANNOTATION INTERFACE LIBRARY (SAIL)

Swan 4.1.1. A graph may have infinitely many drawings. A drawing with good readability can help the viewer's understanding of the graph. There are numerous graph drawing algorithms. Several algorithms are implemented in Swan to draw arrays, linked lists, binary trees, general rooted trees, general undirected and directed graphs (Chapter 6).

The annotator also has control over several buttons in the Swan main window. He can decide whether the viewer is allowed to modify the topology of the graph by enabling or disabling items in the Edit menu. He can also enable/disable the RUN and STEP buttons to allow/disallow the viewer's control of the running process.

4.1 Basic Elements

SAIL provides a small set of elements for the annotator to construct different views of a data structure. These elements not only have logical meanings, but also have graphical attributes since they can be displayed in the Swan display window. These elements include:

**graph** - A graph has not only a set of nodes and edges but also a set of graphical display attributes. Every graph has a unique ID in Swan.

**node** - A node in a graph. Every node has an ID which must be unique within a graph.

The same ID can be used for nodes in different graphs.

**edge** - Any edge in a graph. An edge connects two nodes (e.g., node s and e) in a graph.

If the graph is directed, the edge has a direction (for example, *from* node s to node e). If the graph is undirected, the order of the two nodes makes no difference.

**LC** - The annotator provides graphical layout hints to Swan using the mechanism of layout components (Section 5.2.5).

The following are valid LC types:

- **ARRAYACROSS** - a horizontal array.
- **ARRAYDOWN** - a vertical array.
CHAPTER 4. SWAN ANNOTATION INTERFACE LIBRARY (SAIL)

LISTACROSS - a horizontal linked list.

LISTDOWN - a vertical linked list.

CIRCLenet - Nodes will be evenly distributed on a circle and edges are straight lines forming chords of the circle.

BINTREE - The graph will be laid out as a binary tree.

TREE - The graph will be laid out as a rooted general tree.

KKNET - The general undirected graph will be laid out using the Kamada and Kawai’s algorithm [26].

HIERARCHY - The general directed graph will be laid out hierarchically.

4.1.1 Geometry of the Graph Layout Space

Graphs generated in Swan are drawn on a two-dimensional layout plane with a Cartesian coordinate system. The origin of the coordinate system is at the upper-left corner of the Swan display window when Swan is initialized. The x coordinate increases toward the right and the y coordinate increases toward the bottom. Coordinate units correspond to pixels. The Swan display window has a similar screen coordinate system except its origin is always on the upper-left corner of the display window.

The coordinates of a graph’s nodes and edges in the layout plane are determined when its layout is completed and it is ready to be displayed. When a graph is displayed, its layout coordinates are first mapped to Swan screen coordinates, then it is drawn by Swan display functions.

If the cursor is in the Swan display window, the layout coordinates (not screen coordinates) of the pixel pointed to by the cursor are shown in the Swan location window.

4.2 SAIL Functions

Functions in SAIL can be classified into four categories:
CHAPTER 4. SWAN ANNOTATION INTERFACE LIBRARY (SAIL)

1. construction and manipulation of graphs

2. specification of graphic attributes

3. program status control

4. input and output

A brief discussion of the usage of these functions is given in this section. Refer to the Swan User’s Manual [50] for a complete description.

4.2.1 Graph Construction

Graph

A graph is created by \texttt{sw.newgraph}. A unique ID should be provided by the annotator so that the graph can be referred to later in the annotated program. If the annotator does not want to specify the ID, \texttt{NULLGRAPHID} can be used as the corresponding argument. Then the function will return an ID generated automatically by the system. This ID must be used later to refer to this graph. The graph can be created as a \textit{directed} graph or an \textit{undirected} graph. The default display type of nodes in this graph has also to be declared (e.g., \texttt{BOX} or \texttt{CIRCLE}).

A graph is deleted by \texttt{sw.deletegraph}. If the graph was displayed, it will be removed from the Swan display window. The graph’s ID will become invalid.

A graph can be displayed by \texttt{sw.displaygraph}. \texttt{sw.displayallgraphs} is used to display all the valid graphs in Swan.

A graph contains a set of nodes and edges. The default graphical attributes of nodes and edges in a graph can be set by the function \texttt{sw.setgraphattr}.

Node

A node is inserted in a graph by calling function \texttt{sw.insertnode}. The ID of the node is given by the annotator which can be anything castable to \texttt{NODEID}, a SAIL data type.
CHAPTER 4. SWAN ANNOTATION INTERFACE LIBRARY (SAIL)

A node can be deleted from a graph by the function `sw.deletenode`. It will be deleted both logically and physically, i.e., if the node is already displayed, it will be removed from the Swan display window. Further, all the edges incident on the node will be deleted.

The graphical attributes of a specific node in a graph can be modified by the function `sw.setnodeattr`. This function does not affect settings of other nodes' graphical attributes.

**Edge**

An edge is inserted in a graph by functions `sw.insertedge` or `sw.insertnodeedge`. The first function will insert an edge between `node1` and `node2`. If either `node1` or `node2` is not in the graph, the edge cannot be inserted successfully. Obviously it's cumbersome to insert the two end nodes of an edge every time before the edge can be inserted. The second function is more powerful from this point of view. Its main purpose is to insert an edge while if either or both of the nodes on which the edge is incident are not in the graph, they will also be inserted. The order of a node's children in a general rooted tree is the order that the children were inserted into the graph.

`sw.insertbinedge` is used to insert edges into a binary tree. Identifying whether a node is its parent's left or right child makes edge insertion in a binary tree special.

In a directed graph, the edge will have a direction which is from `node1` to `node2`. In an undirected graph, the order of the edge's two nodes makes no difference.

An edge can be deleted from a graph by the function `sw.deleteedge`. The graphical attributes of an edge can be modified by the function `sw.setedgeattr`. This function does not affect settings of other edges' graphical attributes.

**Layout Component (LC)**

A graph in Swan consists of a set of LC's. Each LC has a set of nodes and edges. When the graph is displayed, the layout of nodes and edges will be determined by the type of the LC they belong to. Details regarding design and implementation of the LC concept can be found in Section 5.2.5.
CHAPTER 4. SWAN ANNOTATION INTERFACE LIBRARY (SAIL)

The graph has a current LC. All the insertions of nodes and edges in the graph will happen in this current LC. Therefore, at least one LC has to be created in each graph so that nodes and edges can be inserted.

An LC is created by \texttt{sw.newlc}. It is deleted by \texttt{sw.deletelc}. The current LC can be changed by \texttt{sw.setcurlc}. The default graphical attributes of nodes and edges in an LC can be set by the function \texttt{sw.setlcaattr}. These attributes override the default attributes of the graph.

4.2.2 Process Control

Swan provides several process control mechanisms, including two control buttons (Section 3.2.3), one graph edit menu and a set of process control functions. Swan lets the annotated program be the main controller of its running process. The viewer of the annotated program has limited controls which have no major effects on the main direction of the process.

Graph Edit Menu

On the upper left corner in the Swan main window, there is a popup menu which will appear when the Edit button is clicked. This menu contains the following four edit actions: insert a node, delete a node, insert an edge, and delete an edge. After the viewer selects one of these items, the annotated program will be notified if the program is waiting for any of these actions to be taken. The annotator can decide what to do according to the menu item selected. Thus, the semantics of these menu items are decided by the annotator, which is not necessarily the same as what the label of the item implies. The annotator can enable or disable any of these menu items by calling function \texttt{sw.enablemenuitem} or \texttt{sw.disablemenuitem}.
CHAPTER 4. SWAN ANNOTATION INTERFACE LIBRARY (SAIL)

Process Control Functions

There are a few functions in SAIL which are solely used to control the process. There are several others which have side effects on the running process.

Function \texttt{sw\_init} initializes the Swan system. It must be the first SAIL function call in the annotated program. After Swan is initialized, its main window is displayed and it is ready to receive SAIL function calls from the annotated program.

Function \texttt{sw\_quit} should be called to quit Swan. This function informs Swan to delete all the elements it created and close all Swan windows.

In the annotated program, \texttt{sw\_run} and \texttt{sw\_step} can be used to set the current mode of the running process. \texttt{sw\_run} makes the program run in continuous mode, in which the break points set in the annotated program are ignored. \texttt{sw\_step} makes the program run in step mode so that the program will stop at any break point set by the annotator.

Break points in the annotated program are set by SAIL function \texttt{sw\_break}. If this function is called in the program, the process will stop and wait for the viewer to click either RUN or STEP to resume running in the corresponding mode. The annotator can enable or disable these two buttons by using \texttt{sw\_enable\_buttons} or \texttt{sw\_disable\_buttons}.

Function \texttt{sw\_wait} will cause the program to stop execution and wait for certain interesting events to happen. These events include selection of a graph editing function or clicking on process control buttons. Once one of these events has happened, \texttt{sw\_wait} returns with the ID of that event.

There is a group of SAIL functions for getting viewer’s input as either a string of characters or a sequence of mouse operations. These functions include \texttt{sw\_get\_str}, \texttt{sw\_pick\_graph}, \texttt{sw\_pick\_node}, \texttt{sw\_pick\_edge} and \texttt{sw\_pick\_pos}.

Errors

Errors could occur when SAIL functions are called with arguments of incorrect values (e.g., out of range), when functions are called in an inappropriate order, and under other
CHAPTER 4. SWAN ANNOTATION INTERFACE LIBRARY (SAIL)

circumstances. SAIL has an internal variable to keep an ID number of the most recent
error. When an error occurs, SAIL function sw_errno can be used to get the ID of the
error. Function sw_errmsg can be used to get a brief description of an error. In addition,
all the errors occurring during the last Swan session are recorded in the file error.log for
future reference.

4.3 Annotation Techniques

Following is a summary of some useful annotation techniques gained from our experience
in using SAIL to annotate typical graph algorithms.

4.3.1 An annotation template

The following is a template which shows the basic procedures to annotate a program
using SAIL:

```
#include "sail.h"  /* The header file for SAIL which must be
                  included in every annotated program. */
GRAPHID g_id;    /* ID of the graph to be created in Swan. */
LCID lc_id;      /* ID of the layout component to be created in Swan.
                  Every graph created in Swan must have at least one LC.
                  Here lc_id is the LC in the graph g_id. */

main()
{
    ...
    sw_init();  /* Initialize Swan. This function must be the first
                 SAIL function call in the annotated program. */
    ...
    create_a_graph(); /* Create a graph in Swan. Details are shown below. */
    ...
    sw_displayallgraphs(); /* Draw all the graphs created in the Swan
                             display window. */
    ...
    sw_quit(); /* Quit Swan. All the graphs created are deleted.
                 The Swan display window is cleared. */
    ...
}
```
CHAPTER 4. **SWAN ANNOTATION INTERFACE LIBRARY (SAIL)**

```c
create_a_graph()
{
    ...
    /* Create an undirected graph in Swan. The shapes of nodes in the graph
     * are circles. The graph ID returned from sw_newgraph will be used in
     * following SAIL function calls to refer to this graph. */
    g_id = sw_newgraph(NULLGRAPHID, CIRCLE, UNDIRECTED);

    /* sw_setgraphattr is used to set graphical attributes of the graph.
     * Here, for example, the node color is light yellow, and both the width
     * and height of the node are 20 pixels. */
    sw_setgraphattr(g_id, NODECOLOR, YELLOW);
    sw_setgraphattr(g_id, NODEWIDTH, 20);
    sw_setgraphattr(g_id, NODEHEIGHT, 20);
    ...
    /* The LC is created for graph g_id. It informs Swan to draw the
     * graph as a circle. */
    lc_id = sw_newlc(g_id, NULLLCID, CIRCLENET);

    /* Set current LC as lc_id so that nodes or edges will be inserted
     * into it if any insertion operations are executed. */
    sw_setcurrLC(g_id, lc_id);
    ...
    /* Insert a node into graph g_id. It will be inserted in the LC lc_id. */
    sw_insertnode(g_id, (NODEID)node_id, node_label);
    ...
    /* Insert an edge into graph g_id. It will be inserted in the LC lc_id. */
    sw_insertedge(g_id, (NODEID)node_id1, (NODEID)node_id2, edge_label);
    ...
}
```

4.3.2 Run the program repeatedly

The viewer may want to run the annotated program several times without quitting the **Swan** system. The annotator can use the following mechanism to make it possible. That is, the annotator can rename the main function in the original C program (e.g., to algo_1), and then make the main function in the annotated program to contain an infinite loop which
CHAPTER 4. SWAN ANNOTATION INTERFACE LIBRARY (SAIL)

repeats the following steps:

- Initialize Swan once, then wait for the viewer to click RUN or STEP to start;
- Execute the function algo_1; and
- Wait for the viewer clicking RUN or STEP to run again.

Following is a code segment to describe this strategy:

```c
main()
{
    ...
    sw_init();
    while (TRUE) {
        algo_1(...);
        sw_print("Press RUN or STEP to run again");
        sw_wait(RUN | STEP);
    }
    ...
}
```

The viewer can click the button QUIT to stop running the program.

4.3.3 Re-build a graph

SAIL provides several functions to create a graph. If an existing graph topology is modified during the running of the annotated program, Swan has to be notified about this modification in order to make future operations on this graph correct.

Usually, there are two methods. One is to use available SAIL functions to do the modification directly. For example, if a node needs to be inserted, call sw_insertnode. If an edge needs to be deleted, call sw_deleteredge to delete it from the graph. However, the annotator also has to modify the data structure used by the annotated program accordingly to make the program really run on a modified graph. If the data structure is complicated, the modification will be a non-trivial task for the annotator.
CHAPTER 4. SWAN ANNOTATION INTERFACE LIBRARY (SAIL)

The other method is to prepare a graph building function for each graph. Whenever this graph’s topology is changed, this function is called. The main operations in this function are to delete the existing graph in Swan and build a new graph to replace the old one according to the graph’s current structure. This method makes graph modification much easier. The main disadvantages are the possible inefficiency when the graph being modified is large and the change of graph layout and identity which may cause inconvenience or inconsistency.

Different methods can be chosen for different graphs to achieve better performance and make annotation easier.

4.3.4 Input and output

The Swan message window is divided into two parts: the top line is used by the annotator to display a one-line message, while the bottom line is used by the viewer to enter any data required by the annotator.

The annotator can use SAIL function sw.print to display a character string in the message window. And he can use sw.getstr to get an input from the viewer.

4.3.5 Communicate with the viewer

The viewer of Swan applications has certain capabilities to control the running of the annotated program and modify logical structures of graphs under the annotator’s permission.

The annotator can allow the viewer to control the running process by enabling the RUN or STEP buttons. He can insert function sw.break anywhere in the annotated program so that the process can stop at interesting points when the viewer steps through the program.

The annotator has to consider carefully whether he allows the viewer to modify the graphs generated by the annotated program. If modifications are allowed, facilities need to be built in the annotated program to support these modifications.
CHAPTER 4. SWAN ANNOTATION INTERFACE LIBRARY (SAIL)

The annotator can create a function `modify_graph()` to be inserted at certain places in the annotated program when he/she allows the viewer to modify one or more graphs.

The following code segment is an illustration of the function `modify_graph()`:

```c
modify_graph()
{
...
    sw_print("Please modify the graph or press STEP to continue") ;

    /* sflag is TRUE after RUN or STEP is clicked. Otherwise, FALSE. */
    sflag = FALSE ;

    /* Enable the control buttons and EDITMENU to allow graph modification. */
    sw_enablebuttons(RUN|STEP) ;
    sw_enablemenuitem(EDITMENU, ITEMINSNODE) ;
    sw_enablemenuitem(EDITMENU, ITEMDELNODE) ;
    sw_enablemenuitem(EDITMENU, ITEMINSEDGE) ;
    sw_enablemenuitem(EDITMENU, ITEMDELEDGE) ;

    /* Start a loop. If any edit menu item is selected, take actions correspondingly.
    If RUN or STEP is clicked, exit the loop. */
    while (TRUE) {
        switch (sw_wait(RUN|STEP|INSNODE|DELNODE|INSEDGE|DELEDGE)) {
            case STEP:
            case RUN:
                sflag = TRUE ;
                break ;
            case INSNODE:
                .insertnode() ;
                break ;
            case DELNODE:
                .deletenode() ;
                break ;
            case INSEDGE:
                .insertedge() ;
                break ;
            case DELEDGE:
                .deleteedge() ;
                break ;
        }
    if (sflag) break ;
```
CHAPTER 4. **SWAN ANNOTATION INTERFACE LIBRARY (SAIL)**

```c
}

/* Disable EDITMENU to disallow graph modifications. */
sw disablemenuitem(EDITMENU, ITEMINSNODE);
sw disablemenuitem(EDITMENU, ITEMDELMODEL);
sw disablemenuitem(EDITMENU, ITEMINSEDGE);
sw disablemenuitem(EDITMENU, ITEMDELEDGE);
...
```

_theinsertnode, _deletenode, _insertedge, and _deletededge_ are functions provided by the annotator to actually carry out the modifications of the graphs.

Basically, _modify graph_ sets up a communication channel between the annotator and the viewer. When the annotator allows the viewer to modify the graph, he enables the graph editing menu items. Otherwise, he disables those menu items. If these menu items are enabled, **Swan** will inform the annotated program when the viewer chooses any one of them.
Chapter 5

System Implementation

5.1 Overview

Swan is a system prototype. The goals for Swan's implementation are to make it simple, easy to use, fast, efficient and portable. Swan is aimed at data structure visualization of graph algorithms.

Swan is implemented in C++. The main reason to choose C++ is that it supports object-oriented design. There are several classes implemented in Swan, including graphs, nodes, and edges. Each of these has associated attributes and operations. Data abstraction and information hiding as provided by object-oriented languages restricts low level details within each object so that Swan's high level system structure is clean. In addition, C++'s popularity, portability and compatibility with C make it preferable to other object-oriented languages, since SAIL can easily support programs written in either C or C++.

Portability is achieved by implementing all graphical operations using GIL, a portable graphical user interface library [5].

The basic data visualization process in Swan is as follows.

- Various graphs are constructed in the annotated program by invoking SAIL functions.
- These graphs are displayed for the viewer to investigate.
- If a graph's logical or physical properties are modified by the viewer, the graph is re-displayed.

Therefore, a graph is the most important data object in Swan. Many activities in Swan are related to graphs. How to represent a graph and what operations can be applied to a graph
CHAPTER 5. SYSTEM IMPLEMENTATION

are the most important issues to be considered. Other issues considered while designing Swan include:

- physical layout of graphs,
- controlling graphical attributes of graphs, and
- supporting interactive modification of graphs by the viewer.

Swan’s solutions to these problems are discussed in the following sections. The implementations of SVI and SAIL are discussed in Chapters 3 and 4.

5.2 Fundamental Classes

The primary data object in Swan is a Graph. Other data objects are either elements or accessories for constructing and maintaining graph objects. These auxiliary data objects include Node, Edge and Layout Component (LC). Each of these data objects are defined as a class. Thus, there are four basic classes in Swan: swGraph, swNode, swEdge, and swLC.

All the graphs created in Swan (i.e., the instances of class swGraph) are kept in a global graph list (gg1). In Swan’s implementation, gg1 is a pointer to a doubly linked list whose nodes contain pointers to swGraph objects (Figure 5.1).

![Figure 5.1: The global graph list gg1](image)

Details of each class will be discussed in the following subsections.

42
CHAPTER 5. SYSTEM IMPLEMENTATION

5.2.1 Class swGraph

Class swGraph is used to model graphs created in Swan. nodelist, edgelist, and 1clist are three members of swGraph. A graph in Swan is represented by an adjacency list. The nodelist is constructed for this purpose. nodelist is a doubly linked list of pointers to instances of class swNode. Each instance of swNode represents a node in the graph. The information about its neighbors is kept within the node itself, as will be illustrated in Section 5.2.2.

Although the nodelist is able to contain all necessary information about the topology of a graph, we use member edgelist in swGraph to link all the swEdge instances in the graph. Any operations targeting all the edges in a graph can be carried out with less overhead, especially when the graph is undirected (because of redundancy in the adjacency list representation for an undirected graph). edgelist is a doubly linked list of pointers to instances of class swEdge.

1clist is a doubly linked list of pointers to instances of class swLC. A discussion of layout components can be found in Section 5.2.5.

Other members in class swGraph include: graph info, node info, edge info and several flags indicating the status of the graph. graph info keeps general information about the graph such as whether it is directed or undirected. node info and edge info store default graphical attributes of the nodes and edges in the graph when they are drawn. The structure of class swGraph is shown in Figure 5.2.

Frequently used member functions in class swGraph include: insertNode, insertEdge, deleteNode, deleteEdge, layoutGraph, and displayGraph. They are used to construct and maintain the graph. When a graph is to be displayed, layoutGraph is invoked to decide positions for all nodes and edges, and then displayGraph is called to draw nodes and edges.
CHAPTER 5. SYSTEM IMPLEMENTATION

![Diagram of graph structure]

**Figure 5.2: The structure of class swGraph**

### 5.2.2 Class swNode

Nodes are the basic components of a graph. Class *swNode* is used in *Swan*’s implementation to represent a node. Each node has an ID specified by the annotator. Node ID’s within a graph are unique. Although a system-wide unique node ID could make the *Swan* implementation easier (e.g., when checking ID’s validity), it may cause inconvenience for the annotator when he wants to use the same ID in different graphs. Each node belongs to exactly one LC in a graph.

Each node keeps a neighbor node list *nblist* which is a doubly linked list of pointers to a structure *nb_node* (Figure 5.3) that, in turn, has a pointer to a neighbor node and a pointer to an edge which is incident on these two nodes.

Each node has a relative position which is an X-Y coordinate within its LC and an
CHAPTER 5. SYSTEM IMPLEMENTATION

```c
struct nb_node
```

```plaintext
Figure 5.3: Structure nb_node
```

absolute position which is an X-Y coordinate in the Swan layout plane (Section 4.1.1). The relative position of a node is given after a layout algorithm is applied on the graph. The absolute position is determined at the proper time before the graph is displayed in the Swan display window.

Each node also keeps a set of graphical attributes including color, shape, and line thickness. These attributes are used when the node is drawn. They override default node attributes in the graph.

The structure of class swNode is shown in Figure 5.4.

The member function displayNode is used to draw the node. The absolute position of this node must be decided before it can be displayed. There are many other member functions supporting basic operations on a node, such as panning, zooming and picking operations.

5.2.3 Class swEdge

Class swEdge has two members, s_node and e_node, which are pointers to instances of swNode representing the nodes on which the edge is incident. Through these two pointers, the end nodes of the edge can be found. On the other hand, if the node is given, any edge incident on it can be found by searching its neighbor node list (Section 5.2.2).
CHAPTER 5. SYSTEM IMPLEMENTATION

Figure 5.4: The structure of class **swNode**

**rpath** is another member of **swEdge**. It is a doubly linked list of **POINT**s which is a structure in **Swan** consisting of the X-Y coordinate of a point on a plane. Edges of a graph in **Swan** are drawn using polylines. **rpath** is used to represent the polyline for an edge. Similar to **rpos** in the **swNode** class, the coordinates of points on the polyline represented by **rpath** are relative to the edge's layout component. **swEdge** also keeps the absolute coordinates of the polyline in its member **apath**. Considering the possible frequent manipulation of graphs by the viewer, more memory space is traded for faster interactive performance.

The set of graphical attributes of an edge is stored in member **info** in **swEdge**. It overrides default edge attributes in the graph.

The structure of class **swEdge** is shown in Figure 5.5.

The member function **displayEdge** is used to display the edge. Other member functions support basic operations on an edge.

### 5.2.4 Relations of swGraph, swNode and swEdge

The relations of class **swGraph**, **swNode** and **swEdge** is illustrated by the following example.

The topology of a graph $G$ is shown at the top of Figure 5.6. It has four nodes $A$, $B$, $C$, and $D$, and four edges $e_1$, $e_2$, $e_3$ and $e_4$. 

46
CHAPTER 5. SYSTEM IMPLEMENTATION

![Diagram of swEdge class structure]

Figure 5.5: The structure of class swEdge

The representation of $G$ in Swan is shown at the bottom of Figure 5.6, rectangles containing lines with arrows at the end represent pointers. Solid arrow lines are pointers to instances of class swNode represented by round corner rectangles labeled $A$, $B$, $C$, and $D$ corresponding to the four nodes in graph $G$, respectively. Dashed arrow lines are pointers to instances of class swEdge represented by round corner rectangles labeled $e_1$, $e_2$, $e_3$ and $e_4$ corresponding to the four edges in graph $G$, respectively. Dotted arrow lines are pointers connecting nodes within a doubly linked list.

The nodelist contains a doubly linked list of pointers to each node. Each node contains a doubly linked list of neighbor nodes represented by pairs of boxes (i.e., pointers). The first pointer of each pair is a pointer to the neighbor node. The second pointer of each pair is a pointer to the edge between the node and the neighbor. For example, the first pair of pointers in node $A$ contains a pointer to node $B$ and a pointer to edge $e_1$. Since adjacency list representation of graphs is used in Swan and graph $G$ is an undirected graph, the first pair of pointers in node $B$ contains a pointer to node $A$ and a pointer to edge $e_1$.

The edgelist contains a doubly linked list of pointers to each edge. Each edge contains a pair of pointers to the nodes on which it is incident. For example, edge $e_1$ contains pointers to node $A$ and node $B$ as shown in the figure.
CHAPTER 5. SYSTEM IMPLEMENTATION

Figure 5.6: A graph $G$ and its representation in Swan
CHAPTER 5. SYSTEM IMPLEMENTATION

The representation of a graph in Swan makes it efficient to find all the neighbor nodes of a specific node and all the edges incident on it. Also, the nodes on which an edge is incident can be found easily.

5.2.5 Class swLC

The layout component (LC) is a special term used in Swan. Swan allows the annotator concentrate on designing the logical structure of graphs. However, without any information about the layout requirements of a graph, it is difficult for Swan to generate an aesthetically pleasing layout of the graph. Even worse, Swan may draw the graph in a way which is totally unexpected to the annotator. For example, if the annotator created a graph which is intrinsically a linked list, Swan has no idea about the structure of the graph and may draw it as a circle if the annotator did not inform Swan about this. Thus, it is beneficial to both the annotator and Swan to require that the annotator provide necessary information about the layout of a graph.

If a drawing of a graph can be characterized in a single layout category such as a linked list or a circle, Swan only needs a layout type identifier to apply appropriate layout algorithms to the graph. However, the annotator may create a graph consisting of several subgraphs. Each of these subgraphs may have different layout requirements. A single type identifier is not enough to describe all these requirements. Therefore, Swan contains a mechanism to deal with complicated graph layout requirements.

Our initial attempt was to use node sharing to establish connections among different graphs. We can make each node ID unique within Swan. The annotator can create separate graphs with simple layout types. Then a node shared by two graphs is the connection point of the two graphs. A group of graphs sharing some common nodes is called a cluster. If the layout of one graph in a cluster is determined, layout of other graphs sharing nodes with this graph can be determined according to positions of the shared nodes and their own layout types. Because nodes can be shared by several graphs, the idea to layout graphs in a cluster sequentially could be easily overwhelmed by complicated node sharing situations.
CHAPTER 5. SYSTEM IMPLEMENTATION

If two graphs share more than one node or share edges, it would be very difficult to satisfy the individual layout requirements of the two graphs. The system wide unique node ID also forces the annotator to use several ID's if he wants to create views of the same data object as nodes in different graphs.

Under such a circumstance, we chose another method. First we allow a graph to have subgraphs. Subgraphs are organized hierarchically. Edges connecting nodes in different subgraphs establish the relations among the subgraphs. Then, we devised a concept, called layout component (LC). The layout component allows the annotator to specify what type of layout should be applied to a portion of a graph. Further, it can be used to describe the layout of each subgraph if the annotator chooses to divide the graph into subgraphs so that appropriate layout algorithms can be applied to each subgraph according to its specific requirement. Using LC, the layout of each subgraph can be determined independently. Then the relative positions of subgraphs can be determined by the hierarchy of subgraphs established through edges between different subgraphs. However, due to the complexity of the relations among subgraphs, the current implementation of LC in Swan only supports a subgraph hierarchy in which only one edge connects two subgraphs.

The adjacency list of an swGraph object is the representation of a graph's logical structure, while the LC is the representation of a graph’s layout structure. Layout components in a graph are organized as a tree structure. Every graph always has a root LC. It is generated automatically by Swan when the graph is created. All other LC’s in the graph are descendents of the root LC. They correspond to the layout information of each subgraph in the graph. The hierarchy of LC’s in a graph is established by the edges connecting two different LC’s. For example, if an edge between node u and node v is inserted in a graph G (node u and node v have been inserted before), node u belongs to layout component L_1 and node v belongs to layout component L_2, then layout component L_1 will be the parent LC of L_2, while L_2 will be a child LC of L_1. swLC has a member children which is a doubly linked list keeping all the children LC’s in this layout component. swLC keeps two pointers for the nodes connecting it with its parent LC: p_node and f_node. For the above
example, they are pointing to nodes $u$ and $v$, respectively, in $L_2$. SwLC also keeps a pointer to its parent LC and a pointer to the graph it belongs to.

The above structure is illustrated in Figure 5.7. Each dashed line rectangle is an LC. $L_1$ is a child of the RootLC. $L_2$, $L_3$ and $L_4$ are children of $L_1$. The parent-child relation between $L_1$ and $L_2$ is established by the edge from node $u$ to node $v$.

![Figure 5.7: The layout component structure in Swan](image)

Theoretically, the tree structure of layout components in a graph allows very complicated layout hierarchies to be constructed if the graph is partitioned into several subgraphs. However, allowing an arbitrary combination of different types of layout components in one graph may not help when producing a meaningful and visually pleasing layout of the whole graph. In many cases, designing a comprehensive layout algorithm accommodating too many variations becomes too difficult to be feasible.

In Swan's implementation, only combinations of four types (i.e., LISTACROSS, LISTDOWN, ARRAYACROSS, ARRAYDOWN) of layout components are allowed for one graph. And two LC's can be connected by at most one edge because two or more edges connecting two LC's could cause contradictory layout requirements. Although these requirements can be compromised, more algorithms need to be developed to retain the effectiveness of LC as a layout hint at a certain level.
CHAPTER 5. SYSTEM IMPLEMENTATION

The member type in swLC indicates what type of physical layout this LC intends to be. Valid types include LISTACROSS, LISTDOWN, TREE, and HIERARCHY. A complete list can be found in the Swan User's Manual [50].

swLC has a member chnodelist which is a doubly linked list of pointers to the nodes which belong to this LC, so that any operations performed on this LC can be applied on its nodes directly without traversing the nodelist of the graph.

There are several members in swLC mainly used for graphical display. bbox is a structure to store the bounding box of an LC. lc_info contains information about the specific LC. node_info and edge_info keep the default graphical attributes for the nodes and edges in this LC. These attributes override default graphical attributes of the graph.

The structure of swLC is shown in Figure 5.8.

Figure 5.8: The structure of class swLC
CHAPTER 5. SYSTEM IMPLEMENTATION

5.3 Graph Layouts

The layout of a graph must be determined before the graph is displayed in the Swan display window. The member function layoutGraph in swGraph is invoked whenever a graph’s layout needs to be calculated. This function uses different layout algorithms according to the type of the graph. There are a number of layout algorithms implemented in Swan (Chapter 6). Each algorithm is implemented as an independent module connected to the function layoutGraph so that more layout algorithms can be added to Swan.

There are several situations in which a graph’s layout needs to be changed. First, when the topology of a graph is modified, a new layout is needed to show the modified structure, such as when nodes or edges are inserted to or deleted from the graph. Second, when the viewer chooses to switch from one kind of layout of the graph to another, for example, Swan provides two kinds of layouts for undirected graphs: CIRCLENET and KKNET. If the graph was first drawn by using the CIRCLENET layout algorithm, when the layout is changed to KKNET, positions of all the nodes and edges will be recalculated. Third, if changes on some graphical attributes of the graph are significant, it may be better to run the layout algorithm again, such as when the sizes of nodes in the graph are changed.

In most cases, Swan’s layout mechanism is automatically activated to renew a graph’s layout. The annotator does have the power to turn automatic layout on or off, allowing manual control. Two flags in class swGraph, fNewLayout and fAutoRelayout, are used to indicate whether a graph needs a new layout and whether this is to be done by Swan automatically. Both of them can be set through the SAIL function sw_setgraphattr.

5.4 Graph Display

Once the layout of a graph is determined, the absolute positions of all the nodes and edges in the graph must be calculated before the graph can be drawn in the Swan display window. Because LC’s in a graph is in a tree structure, the absolute position of the root LC is determined first. Then absolute positions of all the nodes and children LC’s in
CHAPTER 5. SYSTEM IMPLEMENTATION

the root LC are calculated. This process is recursively applied to all the LC’s until all the absolute positions of nodes in the graph are determined. The absolute positions of edges can be calculated after the absolute positions of all the LC’s are decided.

After the absolute positions of nodes and edges are determined, they are drawn in the Swan display window by graphical display functions in Swan Physical Layer according to their graphical attributes.

5.5 Control of Graphical Attributes

Graphical attributes of the layout help the viewer to identify objects and their properties in the graph. In Swan, nodes are drawn either as circles or as boxes. Edges are drawn as polylines. Different colors can be used to show special properties.

Each graph created in Swan has a set of default graphical attributes for its nodes and edges. Each node or edge has its own attributes, which override the graph’s default attributes.

The viewer can modify the graphical attributes of a graph through SVI. The system switch AutoRedraw controls whether the graph will be redrawn automatically after any of its graphical attributes is changed.

Some changes will cause the layout algorithms to be invoked. For example, if the shape of nodes in a graph is changed from circle to box, the layout algorithm has to be called to recalculate the exact intersection positions between the node and edges incident on it.

5.6 Communication Medium

To make Swan applicable in more situations, it is implemented as a communication medium between the annotator and the viewer. It does not assume any knowledge of the data structures used in the annotated program. Swan only recognizes its own data objects — graph, node and edge. Because graphs are widely used to represent various structures and concepts, Swan is able to deal with more general problems through this abstract structure.
CHAPTER 5. SYSTEM IMPLEMENTATION

As mentioned in Chapter 1, the annotator creates graphs by using SAIL library functions and Swan displays these graphs to the viewer in one direction of the communication. The other direction for communication distinguishes Swan from other program visualization systems. This is the viewer's capability to modify the topology of a graph. The following mechanism is provided in Swan to support this two-way communication.

If the annotator allows the graphs created in Swan to be modified by the viewer, Swan must be notified about what “signals” the annotator is expecting to receive. The SAIL library function sw.wait is used in a loop to send queries to Swan. If any expected signal is received from Swan, the corresponding action will be taken and the annotator can decide whether to continue querying or not.

As the medium between the annotator and the viewer, once any of those signals in the system is expected by the annotator, Swan will check repeatedly whether any signals are generated as the result of the viewer’s actions. If any of the generated signals are expected by the annotator, the annotator will be alerted, i.e., sw.wait will return with the signal’s ID.

The viewer may modify graphs created in Swan only if such modifications are supported by the annotator. If so, the menu items in the Edit menu will be activated. The viewer can choose any of those active menu items to make the modifications by following instructions in the Swan message window.

The above process shows that Swan only plays the role of a messenger during communication. The semantics of any of those available modifications are determined by the annotator. This not only gives the annotator more flexibility, but also relieves Swan from being tied to an application-specific design.

5.7 Process Control

The running process of the annotated program is mostly controlled by the annotator, though the viewer also has choices under some circumstances.
CHAPTER 5. SYSTEM IMPLEMENTATION

The SAIL functions used by the annotator to control the process are sw.run, sw.step, sw.break. The first two are used to swap the states of the running process between Run and Step. sw.break is used to set the breakpoints in the running process. Their functionalities are described in Chapter 2.

The implementation of process control looks as follows. A system state switch is set by either sw.run or sw.step. Whenever sw.break is invoked in the annotated program, Swan will check the current system state. If in Run state, sw.break will return immediately. Otherwise, Swan will start a loop waiting for the viewer to click the Step button on the Swan control panel. After Step is clicked, sw.break will return.

Our implementation of process control reflects the design principle of giving the annotator more flexibility. Unlike a common source code debugger, it is a very difficult task to automatically decide what is the appropriate stop position when Swan is running in Step mode because of the various application programs it is dealing with. It is much better to let the annotator decide so that the highlights of annotated programs are deliberately revealed.
Chapter 6

Graph Layout Algorithms

6.1 Introduction

Graphs are widely used in computer science and many other areas to represent logical structures of theoretical problems, or to model complex systems. The capability of a graph to help human beings understand a structure depends largely on its readability, that is, the degree of ease to which the meaning of a graph is captured by the viewer.

Manual graph drawing with the aid of a graph editor gives graph designers high flexibility to improve the readability of a graph. However, manual drawing is always tedious if the graph is large and it is difficult for the designer to show the structure of a graph clearly if it is complicated.

Automatic graph drawing has several advantages over manual drawing. It can draw graphs subject to certain aesthetic criteria and geometric constraints with relatively high speed. The quality of the final drawing is determined by the automatic drawing algorithms. The graph designer has little control over the drawing of the graph except through the constraints he may specify before starting the algorithm.

There is no generic algorithm for drawing all kinds of graphs in an aesthetically pleasing manner. Most automatic graph drawing algorithms apply to specific types of graphs, such as trees [6, 38, 49], planar graphs [12, 20, 25], general undirected graphs [21, 26], or directed graphs [19]. The annotated bibliography [4] is a good resource to study the state of the art of automatic graph drawing algorithms.

A drawing of a graph is also called a layout of the graph. In Swan, the layout of a graph refers to the drawing of a graph on a 2-dimensional Euclidean plane. The positions of all

57
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

the vertices and edges in the graph are determined by the algorithm. Then the vertices are
drawn as rectangles or circles and the edges are drawn as polylines.

The ability to draw graphs automatically makes Swan different from many other pro-
gram visualization systems. It makes annotation much easier because the annotator does
not need to work on the layout of the graph himself. The annotator only needs to specify
the type of the graph.

There are several algorithms implemented in Swan to layout different kinds of graphs
automatically. Because Swan is a data structure visualization system prototype, exploring
innovative graph layout algorithms is not our focus at this stage. We are more concerned
with the efficiency of a graph layout algorithm, because the annotated program will be
run interactively. An aesthetically pleasing layout is important, however. New graph layout
algorithms can be integrated into Swan easily (Chapter 5).

6.2 Swan’s Graph Layout Algorithms

Swan’s graph layout algorithms are discussed in detail in this section. Some of them
are exclusively designed according to Swan’s internal structure, some of them are modified
versions of available algorithms, and others are simply implementations of published
algorithms.

As explained in Chapter 5, the layout algorithm only needs to determine the relative
position of a node within its layout component.

The drawing of labels of nodes is relatively easy because every node has a certain
bounded area. If the label fits into the area, it is drawn directly. Otherwise the first
few letters are shown. Since Swan supports a picking operation, the viewer can view the
complete label by clicking on the node he is interested in.

Displaying labels of edges is rather difficult. Because the edge is drawn as a polyline,
there is no place reserved for its label. A simple approach is to place the label at the mid-
point of the edge. The result is not attractive in most cases due to the overlapping of labels
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

on nodes and edges. Most graph layout algorithms take advantage of the special properties of the graphs they are dealing with so that the edge labels can be drawn at the proper place to reduce overlappings. In other words, each graph layout algorithm has its own edge label drawing module which will be discussed later in this section.

6.2.1 Linked Lists and Arrays

Linked lists and arrays are fundamental data structures. They are graphically represented in Swan by two special graphs.

There are two types of linked lists in Swan: LISTACROSS and LISTDOWN. A linked list with type LISTACROSS will be drawn horizontally, while a linked list with type LISTDOWN will be drawn vertically. Similarly, there are two types of arrays in Swan: ARRAYACROSS and ARRAYDOWN.

Nodes in a linked list are drawn as either boxes or circles connected by the edges with arrows at the end to indicate the relation between two neighbors. Nodes in an array are drawn consecutively without edges between.

If the nodes in the linked list can only have one successor, the layout algorithm is trivial. We can traverse the linked list to decide the position of each node according to the size of the node and minimum length of edges between two neighbor nodes. Edges are drawn to connect two neighbors.

With the support of the layout component mechanism (Chapter 5), an annotator is allowed to build a more complicated structure than the simple linked list and array.

As explained in Chapter 5, every LC has a flag node and a parent node which are used to establish the connections between different LC's. In Swan, a node in an LC of type LISTACROSS or LISTDOWN may be a parent node of another LC of type LISTACROSS or LISTDOWN. Therefore, a simple linked list can be recursively expanded in the X-Y plane to represent relatively complex structure if necessary. Figure 6.1 shows the structure conceptually. Nodes are drawn as rectangles or ellipses with solid lines. Edges are solid lines with arrows. Each dashed line rectangle represents a layout component. Node A has a neighbor
(node B) in \( L1 \) and a neighbor (node F) in \( L2 \). The edge between node A and node F connects not only the two nodes but also the two layout components.

![Diagram of linked lists]

Figure 6.1: A hierarchy of linked lists created in Swan

To decide the position of a node in a linked list, the bounding box of its predecessor's neighbor LC must be known, while calculating the bounding box of an LC can only be done after all its nodes have been assigned a position. This leads to a recursive process. The core part of the algorithm is summarized as following:

**Algorithm:** layoutlinkedlist

**input:** \( lc_1 \) - the LC to be layed out

**output:** all the nodes’ positions in \( lc_1 \) are calculated

1. all the positions of children LC's in \( lc_1 \) are calculated
2. \( lc_1 \)'s bounding box is calculated

**begin**

for all children LC's \( lc_2 \) of \( lc_1 \) do

layoutlinkedlist(\( lc_2 \));

get first node \( n_1 \) of \( lc_1 \);

\( cur.x = cur.y = 0; \)

while \( n_1 \neq NULL \) do

set \( n_1 \)'s position to \( (cur.x, cur.y) \);

if \( n_1 \) has a neighbor LC \( lc_2 \) then

set \( lc_2 \)'s position according to the type of \( lc_1 \);

if \( n_1 \) has a neighbor node \( n_2 \) in \( lc_1 \) then
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

set \( n_2 \)'s position according to the type of \( lc_1 \)
and the bounding box of \( n_1 \)'s neighbor LC if any;
set \( cur_x \) and \( cur_y \) to the next available position according to
\( n_2 \)'s position, bounding box and the type of \( lc_1 \);
\( n1 = n2 \);
width of \( lc_1 \)'s bounding box = \( cur_x \);
height of \( lc_1 \)'s bounding box = \( cur_y \);
end.

The complexity of the above algorithm is \( O(V + E) \) where \( V \) and \( E \) are the total number
of nodes and edges in all the linked lists, respectively.

Array layout can use the same algorithm. Swan allows array LC's and linked list LC's
to be used together. For example, the node in a LISTDOWN LC may have a neighbor LC of
type ARRAYACROSS.

The view of adjacency lists in Figure 1.2 is an example of using linked list layout com-
ponents in Swan.

6.2.2 Trees and Binary Trees

Swan also contains an algorithm to draw rooted trees. To draw a tidy tree, most of the
aesthetic criteria suggested by Bloesch [6] are satisfied. These criteria are:

- Sibling nodes should have their top edges aligned horizontally.
- Sibling nodes should be drawn in the same left-to-right order as their logical order.
- Parent nodes should be centered over the center of their leftmost and rightmost chil-
dren.
- An edge joining the center of the bottom of a node with the center of the top of a
  child should not cross any other such edge or node.
- All nodes that share a raster should be separated horizontally by at least a distance
  \( p > 0 \).
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

- Each node should be separated vertically from its parent by exactly a distance $q$.

The layout algorithm carries out a postorder traversal of the tree. For each node, during the processing, a leftmost position is given. If the node has any children, the algorithm is applied on each child with the leftmost position modified after the bounding box of all previous children has been calculated. After all children are processed, the parent node is placed on the center between its leftmost and rightmost children, and its bounding box is returned. The algorithm is shown as follows:

**Algorithm: layouttree**

**Input:** root — the root node the tree
  start.x — the leftmost position of the tree
  start.y — the top boundary of the tree
  rightside — the right boundary of the tree layout
  bottomside — the bottom of the tree layout

**Output:** All the nodes’ positions in the tree are calculated

**Begin**

if root == NULL then
  rightside ← start.x;
  bottomside ← start.y;
  return;

if there are any children $n_1$ of root not laid out then
  layouttree($n_1$) with modified start.x and start.y according to the bounding box of its previous sibling node;
  Place root at the center of its leftmost and rightmost children;
  Set rightside and bottomside according to the whole bounding box

**End.**

A binary search tree generated in *Swan* is shown in Figure 6.2.

6.2.3 Undirected Graphs

Recently published layout algorithms for general undirected graphs tend to construct physical models in which the graph layout problem can find its counterpart. Then a solution to the problem in the physical model usually implies a good layout for the graph. This idea was pioneered by Eades [17] who took it from a VLSI technique called *force-directed*
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

![Graph Layout Algorithm Diagram](image)

**Figure 6.2:** A binary search tree

placement. Eades modeled a graph as a system of rings and springs. The springs attract the rings if they are too far apart, and repel them if they are too close. The minimal energy state of the system is used to generate a layout.

If Eades's method can be considered as macroscopic, Fruchterman and Reingold's algorithm models a graph in a micro world [21]. In their algorithm, vertices behave as atomic particles, exerting attractive and repulsive forces on one another; the forces induce movement. The goal is to find the equilibrium of the system. Simulated annealing is used in this algorithm, inspired by Davidson and Harel's work [14].

Two algorithms are implemented in **Swan** to draw general undirected graphs. The first one is to distribute nodes along the circumference of a circle evenly (Figure 1.2).
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

Edges are drawn as straight lines between its two end nodes. The second algorithm is an implementation of Kamada and Kawai’s algorithm [26] which is a variation of the force-directed placement method mentioned above. In this algorithm, the total balance of a layout is considered more important than simply reducing the number of edge crossings. Figure 6.3 shows the layout of the same graph in Figure 1.2 by using Kamada and Kawai’s algorithm.

Figure 6.3: A layout of a graph by using Kamada and Kawai’s algorithm

In Kamada and Kawai’s algorithm, a spring model is constructed. Each node in the graph corresponds to a particle in a plane. Each edge in the graph corresponds to a spring which connects two adjacent particles. Then the graph corresponds to a dynamic system with a certain amount of elastic energy. The problem of finding a balanced layout of the
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

graph is converted to another problem: to distribute the particles in the plane so that the total elastic energy is minimized.

The energy of the dynamic system can be computed using Hooke’s theorem. Kamada and Kawai proposed the concept of an ideal distance between vertices that are not neighbors: the ideal distance between two vertices is proportional to the length of the shortest path between them. To find the global minimum energy involves solving a set of $2n$ simultaneous non-linear equations which is time consuming. Instead, a heuristic method is used to find the local minimum of the global energy. Details can be found in [26].

6.2.4 Hierarchical Drawing of Directed Graphs

Directed graphs are useful tools to represent entities and their relationships. For example, directed graphs can represent the module dependency relationship in a large software system, the call-graph in a program, a finite state automaton, or a database design.

Directed graphs are also called digraphs. Edges in directed graphs are also called arcs. There are several ways to draw digraphs. Hierarchical drawing is one important subclass because hierarchy is an intrinsic property of many systems represented by digraphs. To draw these digraphs hierarchically could reveal the internal structure naturally and clearly.

Hierarchical layout of directed graphs is one of the most active subareas in layout algorithms. Many techniques have been devised and a few related theorems have been proved. Several working systems have been constructed, including the GITTO system from the University of Rome [47], the “Graph Browser” at Berkeley [39], the DAG system at AT&T [23], and the EDGE system [35].

The survey paper by Eades and Sugiyama [19] summarizes characteristics of most available working systems and presents a systematic way for hierarchical drawing of directed graphs. The following aesthetic criteria are generally agreed as necessary conditions to draw directed graphs nicely:

1. There should be as few arcs pointing upward as possible.
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

2. Nodes should be distributed evenly over the page.

3. There should be as few arc crossings as possible.

4. Arcs should be as straight as possible.

One algorithm is implemented in Swan to draw a digraph hierarchically. The aesthetics are based on the aforementioned criteria, thus basic procedures in this algorithm follow Eades and Sugiyama’s [19] suggestions though the detailed algorithms are carefully trimmed for Swan.

Given a digraph, the following steps are performed in sequence to draw a hierarchy:

1. Make the digraph acyclic.

2. Layer the acyclic digraph.

3. Order the nodes in each layer.

4. Position the nodes.

Make a Digraph Acyclic

If a graph is known to be acyclic, this step can be omitted.

If a graph contains cycles, we need to reverse those upward arcs. A set $R$ of arcs of a digraph $G = (V, A)$ is a feedback arc set if the reversal of every arc in $R$ makes $G$ acyclic [19].

Suppose that we assign a unique label $\lambda(v) \in \{1, 2, \ldots, |V|\}$ for each node $v$ in a node labeling and we consider an arc $u \rightarrow v$ with $\lambda(u) < \lambda(v)$ as a downward arc, and an arc $u \rightarrow v$ with $\lambda(u) > \lambda(v)$ as an upward arc.

Depth-first search is a simple and efficient way to find the feedback arc set of a digraph. The algorithm used in Swan is as follows:

**Algorithm:** Swan_FAS  
**input:** a digraph $G$
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

output: a uniquely labeled graph $G$ with upward arcs reversed

begin
  label $\leftarrow 0$
  mark each node as unvisited
  for each unvisited node $v$ do $dfs(v)$
end.

Procedure: $dfs(v : node)$
begin
  mark $v$ as visited
  $\lambda(v) \leftarrow label + 1$
  $label \leftarrow label + 1$
  for each neighbor node $u$ of $v$ do
    if $u$ is unvisited do $dfs(u)$
    else if $\lambda(v) > \lambda(u)$ then
      reverse the arc $v \rightarrow u$ to $u \rightarrow v$
end.

Although depth-first search is simple and fast, it is not very effective because algorithms with other heuristics can find a feedback arc set with fewer number of arcs [19].

Layer an Acyclic Graph

This step operates on the acyclic digraph produced by step 1. Nodes are placed on different layers so that all the arcs point downward. The most commonly used layer method is the longest path layering [19] which is also used in Swan.

Algorithm: LongestPathLayer
input: an acyclic digraph $G$
Output: the nodes in $G$ are placed in different layers
begin
  Topological sort the nodes of $G$;
  for each node $v$ do $\lambda(v) \leftarrow 0$;
  for each node $v$ in reverse topological order do
    for each neighbor node $u$ of $v$ do
      $\lambda(u) \leftarrow \max\{\lambda(u), \lambda(v) + 1\}$
    for each node $v$ do
      place $v$ into layer $L_{\lambda(v)}$
end.
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

This algorithm runs in linear time. Another property is that each node is in its lowest possible layer. However, the longest-path layering may be very wide, especially near the bottom of the drawing. Other heuristics can be used to generate better layerings with the cost of more computation time, such as the Coffman-Graham algorithm [13] and the network simplex algorithm [22].

The span of an arc \( u \to v \) with \( u \in L_i \) and \( v \in L_j \) is \( i - j \). A long arc in the layered graph is an arc with a span of more than one node. The next step, ordering nodes in each layer, assumes that there are no long arcs because it is difficult to handle crossings between long arcs. Thus each long arc \( u \to v \) is replaced with a path \( u = v_0 \to v_1 \to \cdots \to v_{j-i} = v \) by adding dummy nodes \( v_1, v_2, \cdots, v_{j-i-1} \). For example, in the graph shown in Figure 6.4, the span of the arc from node 1 to node 7 is 3. Then this arc is separated into three segments with two dummy nodes added. The two dummy nodes are not drawn as ellipses as real nodes in the final layout but their existence is identified by the two bends of the arc from node 1 to node 7.

Dummy nodes should be avoided for a couple of reasons. First, the time used in the next step depends on the total number of nodes. Dummy nodes will cost extra time. Second, bends in arcs in the final drawing occur only at dummy nodes. Gansner et al. gave an algorithm to minimize the number of dummy nodes [23]. The algorithm is not implemented in Swan because of its time complexity.

Order the Nodes in Each Layer

The purpose of this step is to order the nodes in each layer such that the number of arc crossings is reduced. Note that the number of arc crossings in a drawing of a layered graph without long arcs only depends on the ordering of the nodes within each layer. To minimize the arc crossings here is NP-hard [24].

There are several heuristic algorithms available. The barycentric heuristic [46] is adopted in Swan's implementation because it can be shown that as the density of the graph increases, the barycentric heuristic is arbitrarily close to optimal [18].
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

The basic idea is to do a “layer-by-layer sweep” by applying the heuristic. First, an initial ordering of layer \( L_1 \) is chosen. Then for \( i = 2, 3, \ldots, n \), the ordering of layer \( L_i \) is held fixed while layer \( L_{i+1} \) is reordered to reduce crossings between layer \( L_{i+1} \) and \( L_i \).

The algorithm is as follows:

**Algorithm:** \( barycenter(L_{i+1}) \)

begin

for each node \( v \in L_{i+1} \) do

\[ \text{avg}(v) \leftarrow \frac{1}{d^+(v)} \sum_{u \in N^+(v)} x(u); \]

Order all nodes in \( L_{i+1} \) on \( \text{avg} \);

end.

\( N^+(v) \) is the set of nodes on which edges from node \( v \) are incident. \( d^+(v) \) is the number of nodes in \( N^+(v) \), also known as the out-degree of node \( v \).

Positioning the Nodes

It is easy to decide \( Y \) coordinates of nodes. Nodes on the same layer are assigned the same \( Y \) coordinates. The distance between two neighbor layers must be larger than a minimum separation distance.

We should be careful when deciding \( X \) coordinates because it is critical for the final layout to satisfy our aesthetic principles. Because bends in arcs occur at the dummy nodes, better node positions could reduce the angle of such bends. This is an optimization problem which can be stated as a quadratic programming problem [19]. Although standard methods exist, solving the problem is time-consuming.

Gansner [22] proposed a group of heuristics for this problem. However, simply to apply all these heuristics does not guarantee a best possible solution. As the layout algorithm of a digraph in Swan is still in its experimental stage, only the \textit{medianpos} heuristic [22], which is considered the most effective one, is implemented. Details of other heuristics can be found in [22]. The heuristics are applied downward and upward iteratively. In one iteration, if the result is better than the previous best position assignment, it is saved as the best assignment.
CHAPTER 6. GRAPH LAYOUT ALGORITHMS

The medianpos heuristic assigns each node both an upward and downward priority given by the weighted sum of its in and out edges, respectively. On each downward iteration, nodes are processed in the downward priority order and placed at the median position of their downward neighbors subject to the placement of higher priority nodes and space requirements of nodes not yet placed. Upward placement is similar.

There are three types of edges according to the types of their two nodes:

1. Both of the two nodes are real nodes.

2. One node is a real node and the other is a dummy node.

3. Both of the two nodes are dummy nodes.

Since edges between real nodes in adjacent layers can be drawn as straight lines, it is more important to reduce the horizontal distance between dummy nodes. Therefore, the priority ranges from lowest for edges of type 1 to highest for edges of type 3. The actual values for these weights are decided by experiments, such as 1 for type 1, 2 for type 2, and 8 for type 3.

An example of hierarchical layout of a directed graph is shown in Figure 6.4.

6.2.5 Edge Label Layout

In Swan, we consider edge label layout as a part of a graph layout algorithm. The labeling of edges provides more additional information about the graph, which is usually very important in applications.

It is not difficult to draw edge labels for linked lists, rooted trees, binary trees and hierarchical drawings of directed graphs. In these graphs, the positions of edge labels can be decided easily.

In Swan, there are two layout algorithms for undirected graphs: CIRCLENET (Figure 1.2) and KKNET (Figure 6.3). Many edge crossings exist in both of these layouts. The situation is worse in CIRCLENET. To find minimum edge crossings while nodes are placed on a circle
Figure 6.4: A hierarchical layout of a directed graph

is an NP-complete problem [27]. To avoid placing edge labels on the area with high density of edge crossings, a simple and effective heuristic is used in Swan for both of the above two layout algorithms.

For each edge in the graph, its crossings with other edges are found. These crossings separate the edge into line segments. Then the longest line segment is found beside which the edge label is drawn.
Chapter 7
Usability Testing

Ease-of-use for both annotator and viewer is one of the most important characteristics of Swan. Simple, fast and effective data structure visualizations are also goals we are trying to achieve. To see how well these goals have been achieved, we conducted empirical tests with graduate students as annotators. The tests described here are an attempt to make an estimate of the usability of Swan and get some feedback from users to assist future development of the system. They were not intended to be a comprehensive test for all of Swan's functionality.

Tasks

There are three parts to this test.

In the first part, the Swan User's Manual [50] and necessary software libraries are provided to subjects. Examples were introduced to the subjects. Both unannotated and annotated versions of the sample programs were provided for comparison. Subjects were expected to be familiar with the system after reading the manual and experimenting with a couple of examples.

In the second part, subjects were given an implementation of heapsort in C and requested to annotate it. to show the data structure used in the program and highlight significant operations of the algorithm.

In the last part, questionnaires were given to subjects for them to provide suggestions and comments in addition to answering specific questions about the learning process.
CHAPTER 7. USABILITY TESTING

Results

The results from this test are encouraging. The questionnaire and collected data are shown in Appendix B. The three subjects spent from 2 to 8 hours to annotate the heapsort.c program. Although not all the annotated programs are as attractive as we expected, all the subjects used Swan's annotation mechanism correctly. The differences largely originated from subjects' understanding of the heapsort algorithm and how to use SAIL functions properly. The subject who finished the annotation within two hours is an expert C programmer and has good knowledge of algorithms. But he didn't pay much attention to improve the quality of his annotation. Other subjects spent more time since they gave more consideration on how to highlight the important steps in the algorithm. Lack of detailed specification is another cause for diversity of the results. Because annotation can be considered as a process of creative thinking, we believe it is better to leave more scope for the subjects to take advantage of their creativity. Therefore, few restrictions were given for how to annotate the algorithm.

In spite of various minor difficulties encountered, all the subjects spent less than 3 hours studying the manual and found the basic concepts and structures used in Swan understandable. They found that the small set of SAIL library functions were enough for them to annotate the heapsort algorithm. Of course, they may require more functionality if they annotate a large complicated program.

All the subjects reported that Swan is a useful and easy-to-use system, especially for its educational purpose. Swan's potential to help program debugging was not sufficiently realized, mainly because the sample program did not have (obvious) bugs. The simple Swan viewer interface was also reported to be easy to use. In general, most subjects did not have much trouble in using the system.

From the above results, Swan can be considered initially to have achieved most of its goals. Even though some subjects took 8 hours to finish the annotation, it is reasonable to expect they could achieve better performance if they have a chance to annotate more
CHAPTER 7. USABILITY TESTING

programs using Swan in the future. Suggestions for improvement have also been collected. One typical request is to give the annotator more control of the graph layout. Another point is that the viewer should be able to modify the graph’s topology with more freedom. This is a request to improve our two-way communication model between the annotator and the viewer, which definitely needs to be considered in our future work. On the other hand, this also indicates that the significance of one of the most important characteristics of Swan’s visualization model is recognized by the user.
Chapter 8

Conclusion

Visualization systems are usually considered as tools to provide information from different perspectives to help viewers to gain insight into the visualized objects or processes. However, in most cases, it is not very effective if the user of the visualization system is only watching images move around on the screen without being actively engaged. As a data structure visualization system prototype, Swan presents the viewer with an interactive environment to explore the programs annotated with the Swan Annotation Interface Library. With the support of Swan’s communication mechanism, the viewer is no longer only an observer, but also an active player.

Swan is a simple, easy-to-use system. Its flexibility not only makes it generally applicable but also improves its expressiveness. All these characteristics serve Swan’s educational purpose well. The feedback from usability testing consistently confirms that this objective has been successively achieved to a large degree.

The Swan system prototype possesses properties from data structure display systems, algorithm animation systems, and visual debugging systems. There is much work yet to do to improve Swan, such as:

- Support other basic abstract data types in Swan, such as list, stack, queue and set.
- Continue study on the Layout Component to support complicated layout requirements in one graph.
- Refine the fundamental classes used in Swan and improve the efficiency of the system.
- Improve the communication channel between the annotator and viewer.
CHAPTER 8. CONCLUSION

- Implement and/or integrate better automatic graph layout algorithms.
- Polish the Swan Viewer Interface and provide a source code view.
- Port Swan to MS Windows and Mac OS.

In addition to make Swan a better data structure visualization system, it is worth to investigate the potential of integrating more debugging functionalities into Swan so that it can be used as a graphical debugging tool. Swan is also a good platform for researchers in the area of automatic graph layout algorithms to experiment with their new ideas.
Appendix A

Test Questions

Instructions

You are being asked to solve some problems using the Swan system. You are not being tested; you are helping test the software. There are three parts to this test. Please make sure you understand all the questions before you start. After you finish, please send the annotated program in Part 2 and the questionnaire in Part 3 to the experimenter. Thank you for your participation.

Part 1

Please read the manual and explore the demos to make sure you can answer the following questions:

- What can Swan do?
- What is annotation and how can one annotate a program?
- What functions can you use to create graphs in Swan?
- What functions can you use to control the running of the annotated program?
- How can one modify visual attributes of the graph?
- How can one compile, link and run the annotated program?

Part 2

Given a program heapsort.c (both C and C++ versions are available), design and implement an annotation by using Swan to show the data structure and highlight significant operations in the algorithm. (Hint: A heap can be built by using certain graphs supported in SAIL. Show both the array (physical implementation) and the tree (logical representation).)

Part 3

Please answer the questions in the attached questionnaire.
Appendix B

Questionnaire and Collected Data

Note: S1, S2 and S3 refer to the subjects from whom the testing data were collected.

*Please answer the following questions according to your experience in this test:*

1. How long did it take to study the manual?
   - S1: 4 hours
   - S2: 2 hours
   - S3: 3 hours
   - Average: 3 hours

2. What parts of the manual were confusing?
   - S1: Cannot find how to control the position of a graph.
   - S2: No big problem.
   - S3: Comments are marked in the manual.

3. How long did it take to annotate the program (i.e., heapsort.c)?
   - S1: 8 hours
   - S2: 8 hours
   - S3: 2 hours
   - Average: 6 hours

4. Are concepts in Swan understandable? Are there areas of confusion?
   - S1: 1. Yes. 2. Some overlaps in graph attributes and LC attributes.
   - S2: 1. Yes. 2. No confusion.
   - S3: 1. Yes. 2. The concept of LC is little bit hard to absorb.

5. Should any new library functions be added to SAIL?
   - S1: Enough for current use (the test).
   - S2: No.
   - S3: Maybe it’s better to give the annotator more control of the size and location of the graph.
APPENDIX B. QUESTIONNAIRE AND COLLECTED DATA

Please answer the following questions. For each question, circle the number that most closely matches your response. Please write any comments about a question next to the question.

Annotating a program by using SAIL is:

Difficult 0 1 2 3 4 5 6 Easy
S1: 5  S2: 6  S3: 5  Average: 5.3

Running the annotated program makes you understand the algorithm in the program more easily and more completely:

Disagree 0 1 2 3 4 5 6 Agree
S1: 5  S2: 6  S3: 6  Average: 5.7

Swan can help you find logical errors in the original program:

Disagree 0 1 2 3 4 5 6 Agree
S1: 4  S2: 6  S3: 6  Average: 5.3

Swan user interface is:

Difficult to use 0 1 2 3 4 5 6 Easy to use
S1: 5  S2: 6  S3: 6  Average: 5.7

Overall Reactions to the system:

The system is:

Useless 0 1 2 3 4 5 6 Useful
S1: 5  S2: 6  S3: 6  Average: 5.7

The design of SVI is:

Terrible 0 1 2 3 4 5 6 Wonderful
S1: 5  S2: 5  S3: 4  Average: 4.7

Using the system is:

Frustrating 0 1 2 3 4 5 6 Satisfying
S1: 5  S2: 5  S3: 4  Average: 4.7
APPENDIX B. QUESTIONNAIRE AND COLLECTED DATA

Comments:

Please write down any suggestions for improvement.

S1: My opinion is that Swan is a very useful tool, particularly for an educational purpose.

S2: Please make the names of some programs more understandable, such as mst.c.
Swan is a useful system.

S3: It would be better if the annotator has more control over the graph layout.
REFERENCES


REFERENCES


REFERENCES


REFERENCES


VITA

Jun Yang was born on October 15, 1968 in Beijing China. He received a B.S. degree in Computer Science from Tsinghua University, Beijing, China in 1991. From June 1992 to July 1993, he was a graduate student at the University of Regina, Regina, Canada. He began graduate studies at Virginia Tech in August 1993. The work reported in this thesis was completed between January 1993 and December 1993. His research interests include programming environments, algorithms, computer graphics, GUI and image processing. He was a system programmer at CAD Center, Tsinghua University from August 1991 to May 1992 and a research assistant at Virginia Tech from January 1993 to December 1993.

[Signature]