Software Visualization

Visualizing Structure, Behavior and Evolution of Software

Lecture Notes: Software Visualization,
Winter Term 2002/2003, Saarland University

Stephan Diehl

January 30, 2003

Contents

1 Introduction **5**
- 1.1 What is it all about? . . . 6
- 1.2 Taxonomies and Surveys . . . 7
- 1.3 Visualization Pipeline . . . 9
- 1.4 Examples of Software Visualization Tools . . . 10
  - 1.4.1 aiCall: Static Program Visualization . . . 10
  - 1.4.2 X-Tango: Algorithm Animation . . . 11
  - 1.4.3 SeeSoft: Software Evolution . . . 11

2 Static Program Visualization **13**
- 2.1 Pretty Printing . . . 13
- 2.2 Program as Publication . . . 13
- 2.3 Jackson Diagrams . . . 15
- 2.4 Control-Flow Graph . . . 17
  - 2.4.1 Automatic Generation of CFGs . . . 18
  - 2.4.2 Syntax of a simple programming language . . . 18
  - 2.4.3 Computation of a CFG . . . 19
  - 2.4.4 Simple Layout of CFGs . . . 20
- 2.5 Nassi-Shneiderman Diagrams . . . 21

3 Visualizing the Results of Program Analyses **23**
- 3.1 Static Analysis . . . 23
- 3.2 Control-Flow Analysis . . . 23
- 3.3 Data-Flow Analysis . . . 24
  - 3.3.1 Available Expressions . . . 24
  - 3.3.2 Live Variables . . . 25
- 3.4 Examples of Visualization of Analysis Results . . . 27

4 Algorithm Animation **29**
- 4.1 What is it about? . . . 29
- 4.2 Why do people animate algorithms? . . . 30
- 4.3 A Short History of Algorithm Animation . . . 31
- 4.4 Some animations produced by X-Tango . . . 32
- 4.5 3D for Algorithm Animation . . . 34
- 4.6 Some Design Issues . . . 35
- 4.7 Architectures of Algorithm Animation Tools . . . 36
  - 4.7.1 Example: Interesting Events in POLKA . . . 36
  - 4.7.2 Example: Declarations in LEONARDO . . . 37
- 4.8 Abstract Algorithm Animation . . . 38

Chapter 1

Introduction

The important role that visualization plays for human reasoning in general and scientific progress in particular has been emphasized by philosophers throughout the centuries.

> "... thought is impossible without an image."
>
> [Aristotle,350 BC]

> "Imagination or visualization, and in particular the use of diagrams, has a crucial part to play in scientific investigation."
>
> [Rene Descartes, 1637]

> "The understanding can intuit nothing, the senses can think nothing. Only through their union can knowledge arise."
>
> [Emanuel Kant,1781]

> "Logicians may reason about abstractions. But the great mass of men must have images."
>
> [Thomas Babington Macaulay,1825]

Today computers have become an important tool to create visualizations and help the user to better understand complex phenomena. As a consequence visualization has become a discipline of computer science.

> [Visualization is] "the use of computers or techniques for comprehending data or to extract knowledge from the results of simulations, computations, or measurements."
>
> [McCormick, DeFanti, Brown,1987]

The above definition does not restrict visualization to rendering information visible, but it is more general and sets visualization equal to *perceptualization*, which includes sonificaton, tactilization and haptilization.
Visualization plays a major role in the use of computers to support human reasoning, a field that was coined "intelligence amplification" or short IA, in contrast to "artificial intelligence" or short AI, where the goal is that the computer itself becomes intelligent.

> "Intelligence Amplification"
>
> [Frederick Brooks, 1969]

Visualization is heavily used in mechanical engineering, chemistry, physics or medicine. Computer scientists have developed sophisticated systems to produce visualizations for these disciplines. Astonishingly enough, computer scientists have only made little use of visualization as a tool for designing, implementing and maintaining software. Even worse, many consider themselves as theoreticians and disregard visualization – an etymologically wrong

dichotomy[1]. Programmers tend to adapt to the level of representation provided by the computer, instead of adapting the computers representations to their perceptive abilities.

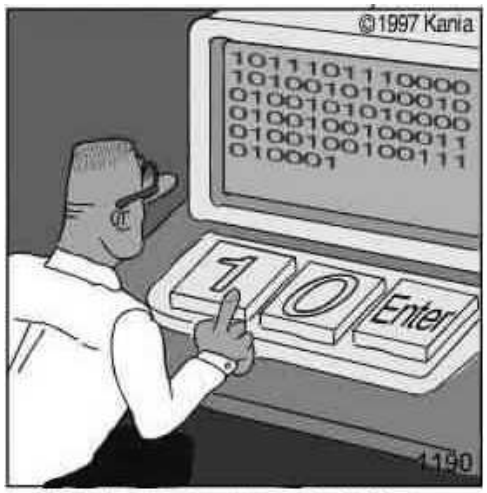

Figure 1.1: No visualization required.

Despite all formal and cryptical notations, the terminology of computer science is rich of metaphors. The goal of such metaphors is to evoke mental images to better memorize concepts and to exploit analogies to better understand structures or functions. Computer scientists use the terms 'automata' and 'machines' for mathematical models of computation. The terms 'tapes', 'trees', 'leaves', 'queues', 'files', 'folders' and 'archives' are used to denote data structures. For example, a Turing machine is a mathematical model comprised of sets, functions, and/or relations. The *machine* analogy lets us transport aspects from the physical world to the mathematical and thus helps to better understand the mathematical model. We might even think of gear wheels and how one drives the others, once we start to turn one of them. The goal of software visualization is not to produce neat computer images, but computer images which evoke mental images for comprehending software better. Finding new metaphors thus will not just produce better visualizations, but it will also improve the way we talk about systems.

1.1 What is it all about?

So far, we talked about visualization, its importance for human reasoning and in particular for science. In the following chapters we will look at the use of visualization in the context of software development. Many authors define software visualization as the

> Visualization of algorithms and programs. (Narrow Definition)

This definition excludes a lot of uses of visualization techniques in computer science and has also hindered synergies in the past. In this text we define software visualization as the

[1] The word 'theory' comes form the Greek word 'theorein' which means 'to view'. The early Pythagoreans supported their theorems not by proofs but by contemplation.

> Visualization of artifacts related to software and its development process. (Wide Definition)

In fact, researchers in software visualization develop and investigate methods and use of computer graphical representations of different aspects of software, e.g. its static structure, its concrete and abstract execution, and its evolution. Short, they are concerned with

> Visualizing Structure, Behavior and Evolution of Software

Today there are two major disciplines of visualization: *Scientific visualization* processes physical data, whereas *information visualization* processes abstract data[2]. As algorithms are a kind of information, we consider software visualization part of information visualization.
Coming from a mathematical background, it is quite illuminating to realize that algorithms or programs are an essential part of nature. For examples, biological programs are encoded in genes and executed by proteins, enzymes, etc. Some behaviors can be seen as programs which animals (and humans) learn through conditioning (Pavlov's dogs).
In 1950 Wiener stated that information is neither matter nor energy. Information is thus an entity of its own, while matter and energy are just media to carry information. One consequence of Einstein's relativity theory (Einstein, 1905) with its equational slogan $E = mc^2$ is that matter can be converted into energy and vice versa. So far, we don't know and we don't expect that there exists a similar relation between information and matter, or information and energy. Nevertheless it has become a popular theme of science fiction.

1.2 Taxonomies and Surveys

Several researchers proposed taxonomies to classify software visualization research and tools.
Myers introduced a taxonomy for program visualization [Mye90] which identifies 6 regions in a 2×3 matrix. He distinguishes data, code and algorithm visualization where algorithm visualizations represent algorithms at a higher level of abstraction than program code.

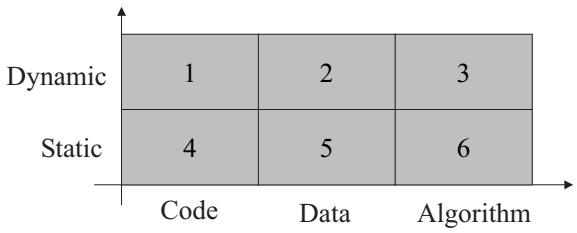

Figure 1.2: Taxonomy by Myers

Three years later Price et. al. suggested a more hierarchical taxonomy of software visualization [PBS93]. They distinguish program visualization, which consists of code and data visualization, and the more abstract algorithm animation. In addition they introduce a number of aspects which should be used to classify software visualization tools:

[2] There have been several attempts in the literature [Chi00] to make the distinction more clear, but there always remains an overlap of both disciplines.

Scope = range of programs used as inputs for the visualization

Content = what information about the software is visualized

Form = characteristics of output of system (e.g. medium)

Method = how is the visualization specified

Interaction = how does the user control the system

Effectiveness = how well does the system communicate information to the user

At about the same time Roman and Cox used a very similar set of aspects to classify some existing software visualization tools [RR93]: scope (code, data state,control state, behavior) abstraction, specification method, interface and presentation.

In 1996 Oudshoorn et. al. proposed a quite different and more technical program visualization taxonomy which was based on what kind of software on what kind of hardware is actually visualized [OWE96].

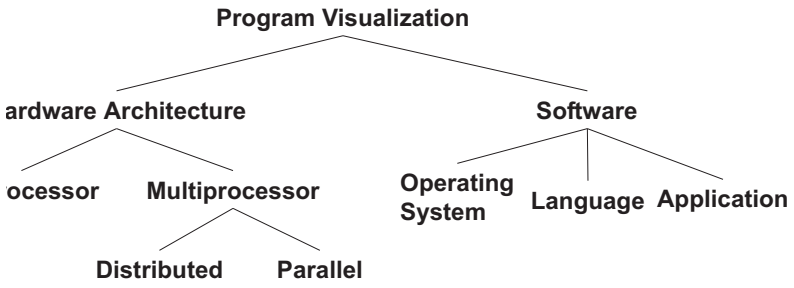

Figure 1.3: Taxonomy by Oudshoorn et. al.

In an attempt to identify open research questions in software visualization the current author did a literature survey and classified research papers in a 4×4 matrix [Die02a]. The two dimensions of the matrix are the classical abstraction layers of software systems (hardware, virtual/abstract machine, program and system) and the static and dynamic phenomena of these layers. The map is incomplete in the sense that one could add additional layers (e.g. operating system) or structures (e.g. project structure). In the matrix shades of gray indicate how much published research exists in certain areas of software visualization. The survey gives a rough orientation on the activity of research in these areas.

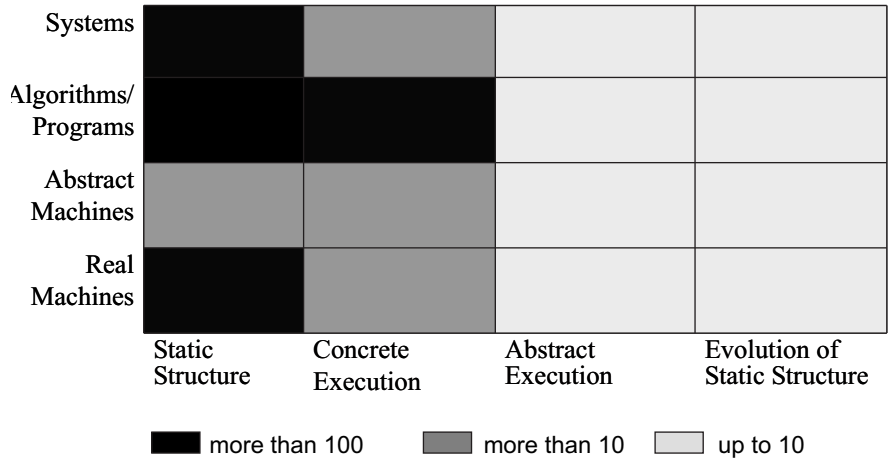

Figure 1.4: Literature Survey

In a recent survey [Kos02] based on questionnaires filled in by 111 researchers from software maintainance, re-engineering and reverse engineering, Rainer Koschke reports that 40 % find software visualization absolutely necessary for their work and that another 42 % find it important but not critical.
In another survey [BK01] with 107 participants mostly from industry Bassil and Keller found the following reasons why practitioners apply software visualization. In the order of decreasing importance the benefits of software visualization tools were:

- Savings in time and money,
- Better comprehension of software,
- Increase in productivity and quality,
- Management of complexity,
- To find errors

They also found that two issues ranked highest on the participants wish list:

- Integration of software visualization tools into other (third-party) tools
- Better import/export of data and visualizations

In the following chapters we will learn what kinds of information about software exist, how they are computed, what visual representations are appropriate for them and how these are computed. Our selection of topics, systems and approaches is by no way complete, but we tried to select seminal work as well as newer approaches which we found most promising.

1.3 Visualization Pipeline

The creation of computer images is just the last step in the visualization pipeline: data gathering, data analysis, and visualization. In interactive visualizations the user can control the previous steps of the pipeline based on the graphical output produced before. This way of interaction is sometimes called visual or computational steering[JPH+99]. It provides the technical basis for

graphical user interfaces designed according to Ben Shneiderman's basic principle, the visual information seeking mantra: *overview first, then zoom and filter, then details on demand* [Shn96].

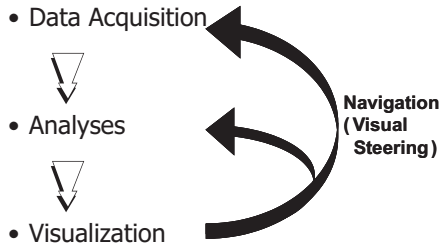

Figure 1.5: Visualization pipeline

1.4 Examples of Software Visualization Tools

To conclude this introduction we will briefly look at three examples of software visualization tools drawn from different areas.

1.4.1 aiCall: Static Program Visualization

AbsInt GmbH's analysis tool called aiCall produces visualizations of control flow graphs of embedded applications [EB02]. In these graphs the results of a static program analysis are shown. For each instruction and each function the analysis computes the stack usage. This information can for example be used to prevent runtime errors due to stack overflow.

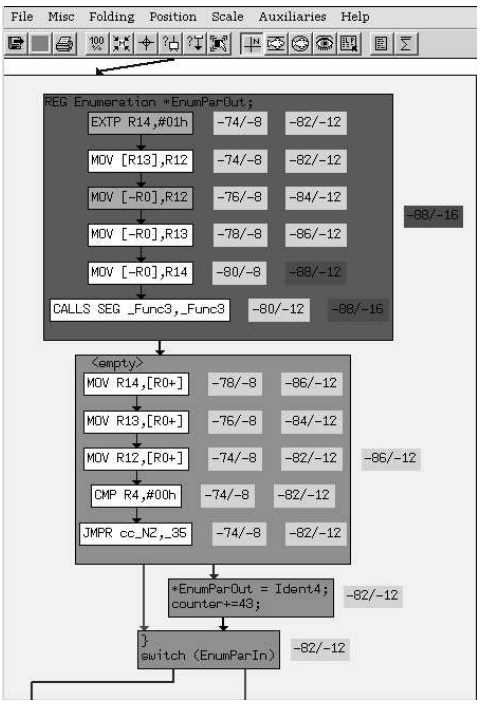

Figure 1.6: Stack Usage

1.4.2 X-Tango: Algorithm Animation

In Figure 1.7 a snapshot of an animation of the Quicksort algorithm is shown. The animation was produced with the X-Tango algorithm animation tool[Sta90b]. The elements to be sorted are shown as vertical bars, nested recursive calls are indicated by the boxes around some of the bars, and the current pivot element, i.e. the element where the list is split, is colored green.

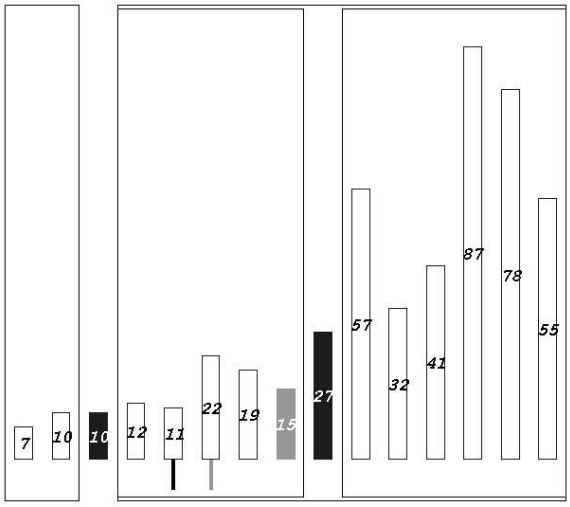

Figure 1.7: X-Tango animation of Quicksort

1.4.3 SeeSoft: Software Evolution

SeeSoft was developed at AT&T Bell Laboratories to visualize changes and metrics related to evolving large (several million lines of code) and complex software systems. Files are represented by rectangles. Within each rectangle colored pixels or lines represent lines of the source code. In the example the color indicates the age of the last modification. Blue (cold) is used for lines which have not been changed for a long time, whereas red (hot) is used for recently changed lines.

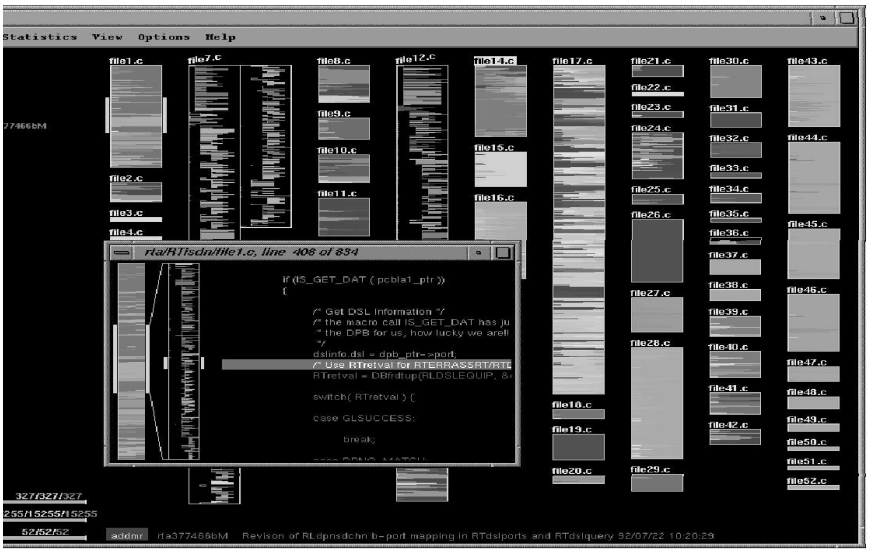

Figure 1.8: Visualizing the age of program code changes

Chapter 2

Static Program Visualization

In this chapter we look at different ways to visualize the structure of a program given the program text as a sequence of characters. We look at both text-based and diagrammatic methods.

2.1 Pretty Printing

Originally pretty printing was restricted to the use of indentation, spaces and line breaks to make the structure of a program more explicit. By tabbing declarations can be vertically aligned. Different width of spaces declarations makes operator precedence more explicit. With advance of technology also fonts, font face and colors are used, e.g. bold face is used for keywords and italics for comments. Different font sizes indicated nesting (lexical scope). As can be seen in the example in Figure 2.1, if done wrong pretty printing can suggest wrong nesting. In the pretty printed text on the right, the statement `i++` seems to be part of the body of the loop.

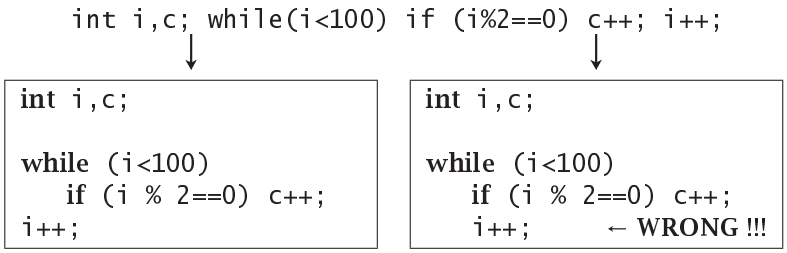

Figure 2.1: Pretty printing

2.2 Program as Publication

In 1984 Donald Knuth, the inventor of the document typesetting system TEX, introduced the term *literate programming* [Knu84, Knu92] evangelizing that programs should be considered as works of literature. To facilitate the production of well-documented and neatly typeset programs, Knuth developed the `WEB` tool.

> "I'm pleased that my work on typography, which began as an application of computers to another field, has come full circle and become an application of typography to the heart of computer science."
>
> [Knuth, 1984]

With less focus on tool support Baecker and Marcus investigated the use of typography to increase the readability of programs[BM98, BM90]. They suggest to produce *program books* with the same care as other textbooks. Their

program book consists of the frontmatter, including a cover page, a title page, an abstract, a program history, information about the authors, and a table of contents. The first chapter contains the user documentation, e.g. as a tutorial on how to use the program. The second chapter gives an overview of the program structure through a program map and the call hierarchy. The program map is a table with thumbnails of each program code page with major function names emphasized. Each subsequent chapter contains the pretty printed program code of a source file (here files with extension `.c` and `.h`) with comments in the margins (see Figure 2.2). The last chapter of the book provides the programmer documentation including the installation and maintenance guides. At the end of the book several indices like cross-references, caller index, and callee index are given. Finally, at the back cover page the highlights of the book content are summarized.

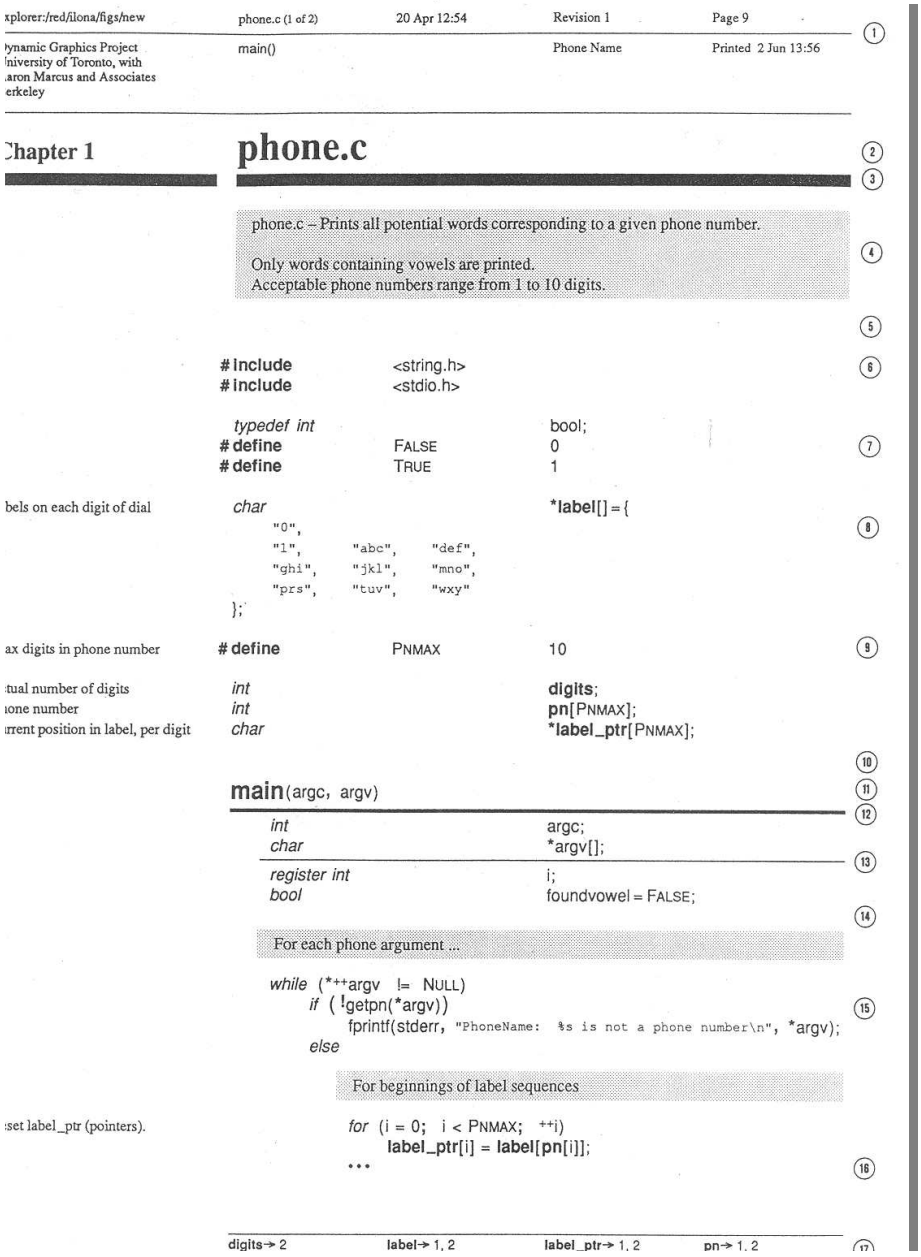

Figure 2.2: A page from the program book

2.3 Jackson Diagrams

We start our discussion of several diagrammatic techniques with Jackson diagrams. They decompose a program hierarchically [Jac75]. According to the Jackson-Structured-Programming methodology data structures involved are first hierarchically decomposed using these diagrams, then the program structure should follow this decomposition.
The basic elements of Jackson diagrams are actions. Actions are decomposed into subactions as shown in Figure 2.3. A sequence A consists of the execution

of the subaction C after the subaction B. An iteration A consists of multiple repetitions of B as long as the iteration condition C holds. Finally, an alternative A is either the subaction B if condition C1 holds, or the subaction C if the condition C2 is true.

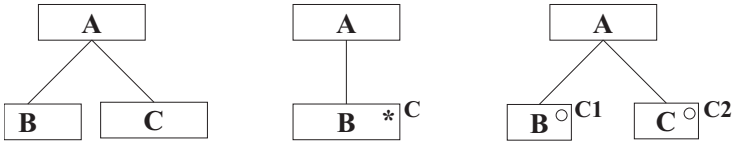

Figure 2.3: Sequences, iterations and alternatives in Jackson diagrams

In the example in Figure 2.4 condition C1 is true if the stack of bills is not empty, C2 is true if the number of the credit card is not valid and C3 is true otherwise.

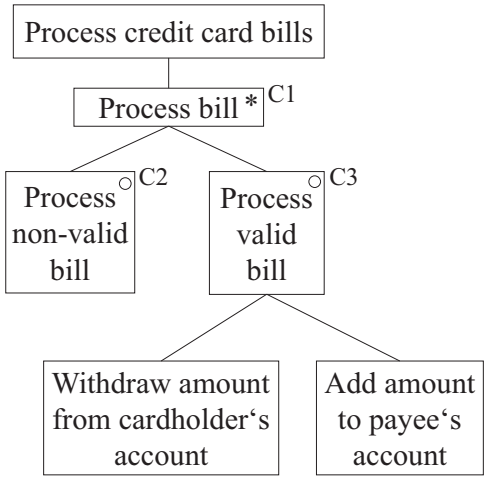

Figure 2.4: Example: Jackson diagram for designing a system

In Figure 2.5 the Jackson diagram for the factorial program below is shown.

```
int fact(n) { if (n>1)
   { nfact=2;
     for(int i=3;i<=n;i++)
       nfact=nfact*i;
   }
  else
   { nfact=1;
   }
  return nfact;
}
```

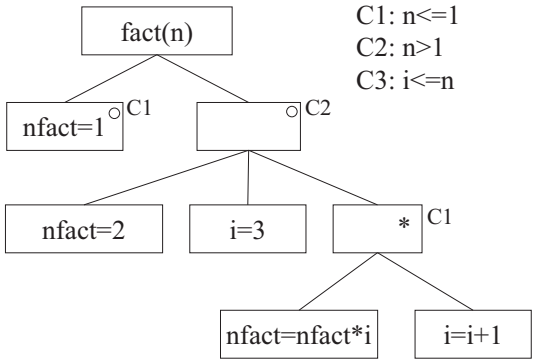

Figure 2.5: Example: Jackson diagram of a factorial program

2.4 Control-Flow Graph

In 1947 Goldstine and von Neumann [GvN47] introduced control-flow graphs (short CFG) to depict the structure of programs. In these graphs rectangular nodes represent evens, activities, processes, functions or statements, whereas nodes in form of a diamond contain branch conditions and can have serveral several exits (outgoing edges). Edges in the graph are drawn as arrows and depict the transition from one statement to another, i.e. the flow of control (see Figure 2.6). Later, many more graphical elements have been added and have been standardized as DIN 66001 (flowcharts).

Figure 2.6: Statements and alternatives in control-flow graphs

In Figure 2.7 a control-flow graph of the previously shown factorial program is shown. To get this graph, the `for` loop has been converted into a `while` loop.

```
int fact(n) { if (n>1)
   { nfact=2;
     int i=3;
     while(i<=n)
      { nfact=nfact*i;
        i=i+1;
      }
   }
  else
   { nfact=1;
   }
  return nfact;
}
```

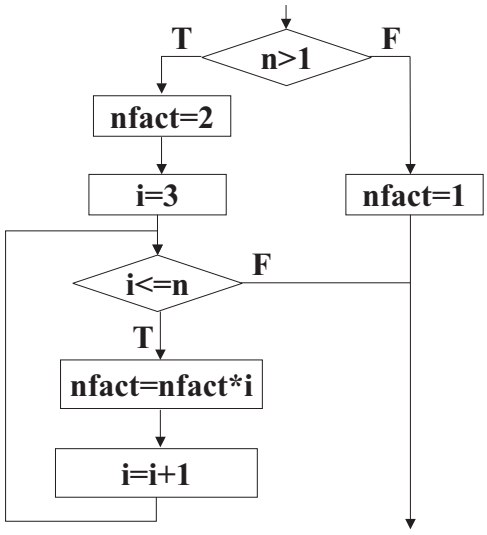

Figure 2.7: Example: Control-flow graph of a factorial program

For many applications of control-flow graphs it is convenient to combine sequences of statements into a single node called basic block as shown in Figure 2.8.

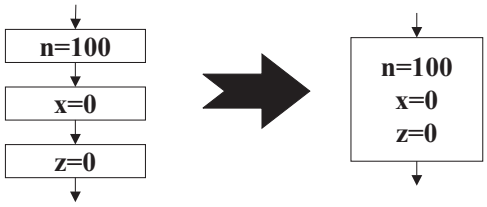

Figure 2.8: Basic block

2.4.1 Automatic Generation of CFGs

The first CFGs were drawn by hand to develop, explain or debug programs. But soon researchers developed programs to automatically compute and layout the control-flow graph of a given program. One of those early systems was developed by A.E. Scott on an IBM 705. The layouted graph was drawn by a text printer with a very limited character set, see Figure 2.9.

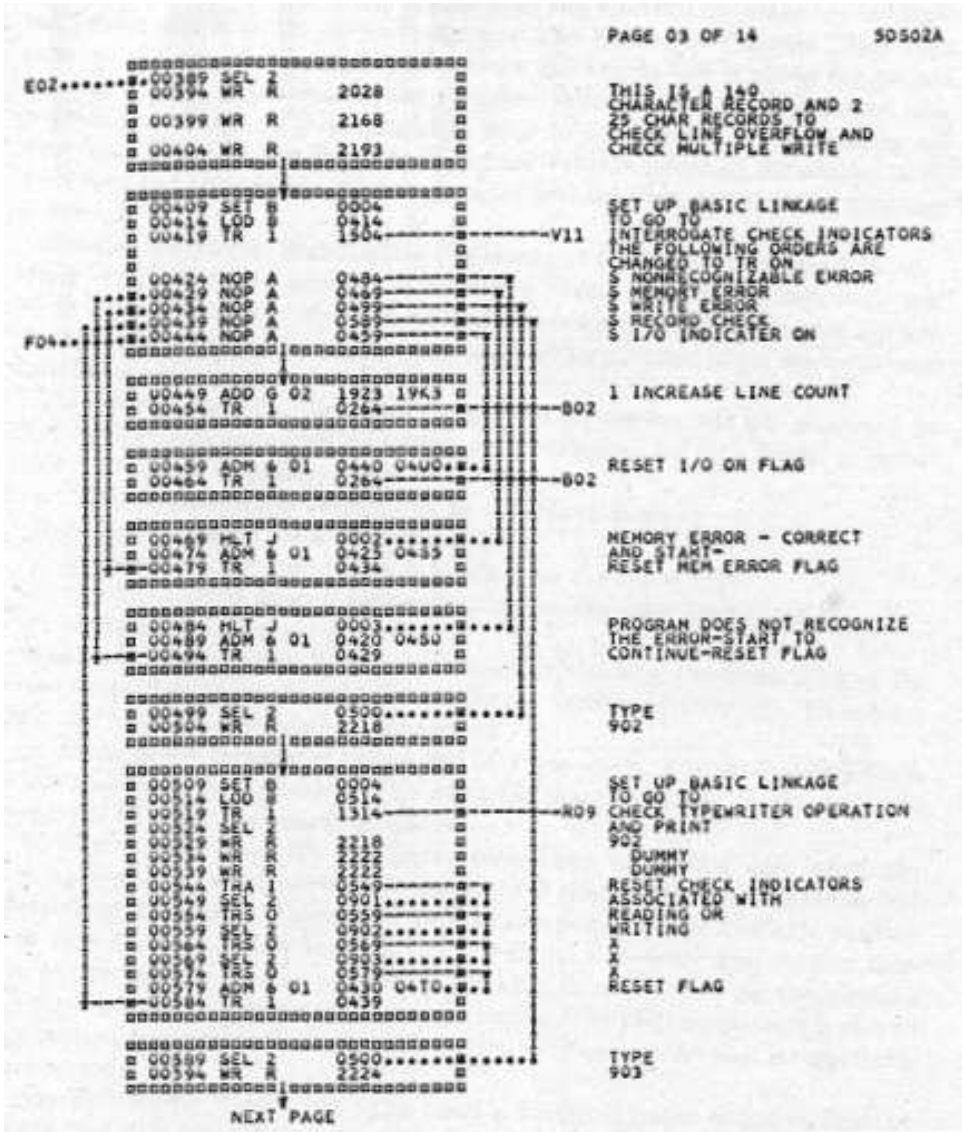

Figure 2.9: Generated control-flow graph printed as text

We will give the synatx of a simple programming language, define how to compute the CFGs of programs in that language and finally give a simple layout algorithm for drawing such graphs.

2.4.2 Syntax of a simple programming language

Programs in the language, that we consider in the rest of this chapter, consists of assignments, alternatives and loops. Expressions can occur on the right-

hand side of an assignment, or in as conditions of alternatives and loops.

$G_{simple} = \{$ S $\longrightarrow$ V=E ^{P}
| S;S
| if (E) {S} else {S} ^{P}
| while (E) {S} ^{P}
V $\longrightarrow$ variable name
E $\longrightarrow$ expression
P $\longrightarrow$ label of program point
$\}$

Figure 2.10: Syntax of a simple programming language

To make the following presentations easier, we also assume that every statement, i.e. program point, has been given a unique label.
In the following we will use the notation $L_G(A)$ to denote the language defined by a grammar G for a non-terminal A, or more formally:

$$L_G(A) = \{w | w \text{ is a terminal word with } A \overset{G}{\rightarrow}{}^{*} w\}$$

In particular, $L_{G_{simple}}(\mathsf{S})$ is the set of all programs which can be written in our simple programming language.

2.4.3 Computation of a CFG

Let $s, s_i \in L_{G_{simple}}(\mathsf{S}), v \in L_{G_{simple}}(\mathsf{V}), p \in L_{G_{simple}}(\mathsf{P})$ and $e \in L_{G_{simple}}(\mathsf{E})$. Figure 2.11 shows how the CFG of a statement is built from the CFGs of the statements that it contains. For an assignment $v{=}e^p$ the CFG simply consists of a node representing the assignment. For a sequence $s_1\,;s_2$ of statements there is an arrow from the end of the CFG built for the first statement to the entry of the CFG built for the second statement. For a loop $\texttt{while(}e\texttt{) \{}s\texttt{\}}^p$ a node with the condition e is connected (true branch) with the CFG built for the body of the loop. From the end of that CFG an arrow is drawn back to the loop condition. Finally for the alternative $\texttt{if(}e\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}}^p$ a node representing the condition is connected with the CFG built for the first statement (true branch) and with the CFG built for the second statement (false branch).

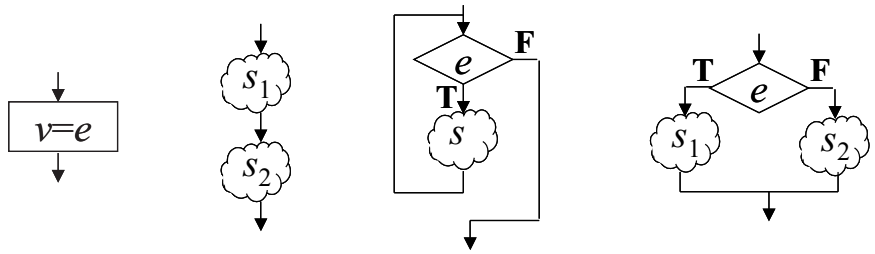

Figure 2.11: CFGs for assignment, sequence, `while` loop, and alternative instructions

To precisely define the computation of a control-flow graph we first give a formal definition of a control-flow graph. A control-flow graph is a tuple (V, E, in, out) where

- V is a set of nodes,

- $E \subseteq V \times L \times V$ a set of edges,
- $L = \{\epsilon, \mathsf{T}, \mathsf{F}\}$ a set of labels,
- and $in, out \in V$ are start and end nodes.

Note, that according to this definition a CFG has exactly on entry and one exit node. Let Γ be the set of all control flow graphs. Below we define a function $\mathsf{cfg} : L_{G_{simple}}(\mathsf{S}) \longrightarrow \Gamma$ which maps programs to control flow graphs. $\mathsf{cfg}(w) = (V, E, in, out)$ where

$$\text{if } w = v{=}e^p \text{ then } \begin{cases} V = \{w, in, out\}, \\ E = \{(in, \epsilon, w), (w, \epsilon, out)\}, \\ \text{and } in, out \text{ are two new nodes.} \end{cases}$$

$$\text{if } w = s_1\,;s_2 \text{ then } \begin{cases} V = (V_1 - \{out_1\}) \cup (V_2 - \{in_2\}), \\ E = \begin{array}{l} (E_1 - \{(v, l, out_1) | v \in V_1\}) \cup (E_2 - \{(in_2, \epsilon, v) | v \in V_2\}) \\ \quad \cup \{(v_1, l_1, v_2) | (v_1, l_1, out_1) \in E_1, (in_2, \epsilon, v_2) \in E_2)\} \end{array} \\ \text{and } in = in_1, out = out_2. \end{cases}$$

where $(V_1, E_1, in_1, out_1) = \mathsf{cfg}(s_1)$ and $(V_2, E_2, in_2, out_2) = \mathsf{cfg}(s_2)$

An invariant in our construction is that an entry node has only one outgoing edge, whereas an exit node can have many incoming edges. To connect two CFGS in a sequence all these incoming edges of the exit node of the first CFG are connected to the target node of the outgoing edge of the entry node of the second CFG.

$$\text{if } w = \texttt{while(}e\texttt{)\{}s\texttt{\}}^p \text{ then } \begin{cases} V = V_0 \cup \{e^p\}, \\ E = \begin{array}{ll} & (E_0 - (\ \{(in_0, \epsilon, v) | v \in V_0\} \\ & \qquad \cup \{(v, l, out_0) | v \in V_0\})) \\ \cup & \{(in_0, \epsilon, e^p), (e^p, F, out_0)\} \\ \cup & \{(e^p, T, v) | (in_0, \epsilon, v) \in E_0\} \\ \cup & \{(v, l, e^p) | (v, l, out_0) \in E_0\} \end{array} \\ \text{and } in = in_0, out = out_0. \end{cases}$$

where $(V_0, E_0, in_0, out_0) = \mathsf{cfg}(s)$

$$\text{if } w = \texttt{if(}e\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}^p \text{ then } \begin{cases} V = (V_1 - \{in_1, out_1\}) \cup (V_2 - \{in_2, out_2\}) \cup \{e^p\}, \\ E = \begin{array}{ll} & (E_1 - \{(in_1, \epsilon, v_1), (v_2, l, out_1) | v_1, v_2 \in V_1\} \\ \cup & (E_2 - \{(in_2, \epsilon, v_1), (v_2, l, out_2) | v_1, v_2 \in V_2\} \\ \cup & \{(in, \epsilon, e^p)\} \\ \cup & \{(e^p, T, v) | (in_1, \epsilon, v) \in E_1\} \\ \cup & \{(v, l, out) | (v, l, out_1) \in E_1\} \\ \cup & \{(e^p, F, v) | (in_2, \epsilon, v) \in E_2\} \\ \cup & \{(v, l, out) | (v, l, out_2) \in E_2\} \end{array} \\ \text{and } in, out \text{ are two new nodes.} \end{cases}$$

where $(V_i, E_i, in_i, out_i) = \mathsf{cfg}(s_i)$

2.4.4 Simple Layout of CFGs

We use rectangular boxes and place in and out nodes in the middle of the upper and lower borders. For expressions the height h depends on the font size and the width also on the length of the expression. For a `while` loop the heights h_1, h_2 and h_3 are fixed. So we can compute $H = h_1 + h_2 + h_3$ beforehand. As a result we have $h = H + h_e + h_s$.

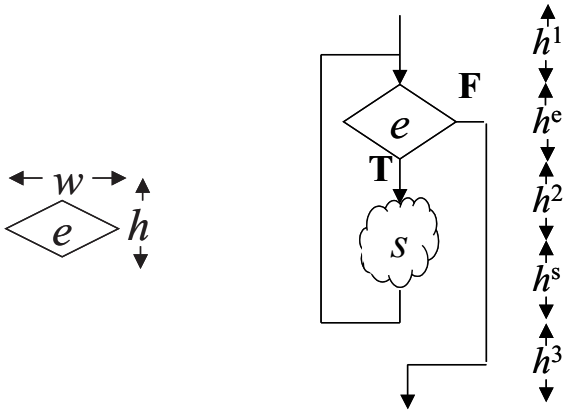

Figure 2.12: Box heights of the CFG for expressions and loops

We define a function box : $L_{G_{simple}}(\mathsf{S}) \longrightarrow \mathcal{R} \times \mathcal{R}$ which maps programs to the sizes of the box to layout its control flow graph:

$\mathsf{box}(e) = (w, h)$ where w, h depend on the font.

$\mathsf{box}(s_1\,;s_2) = (W + \mathsf{max}(w_1, w_2), H + h_1 + h_2)$ where $(w_i, h_i) = \mathsf{box}(s_i)$

$\mathsf{box}(\texttt{while(}e\texttt{)\{}s\texttt{\}}) = (W + max(w_e, w_s), H + h_e + h_s)$ where $(w_e, h_e) = \mathsf{box}(e)$ and $(w_s, h_s) = \mathsf{box}(s)$
Knowing the sizes of each box we can now easily draw the arrows as we only have to connect them to the middle of the upper or lower border of a box.

2.5 Nassi-Shneiderman Diagrams

To enforce more structured programs Nassi and Shneiderman introduced nested rectangular diagrams, also known as *structograms* [IS73].

> "Not only does this notation help the programmer to think in an orderly manner, it forces him or her to do so. ... The absence of any representation of the GOTO or branch statement requires the programmer to work without it: a task which becomes increasingly easy with practice."
>
> [I. Nassi and B. Shneiderman, 1973]

The primitive diagrams are shown in Figure 2.13.

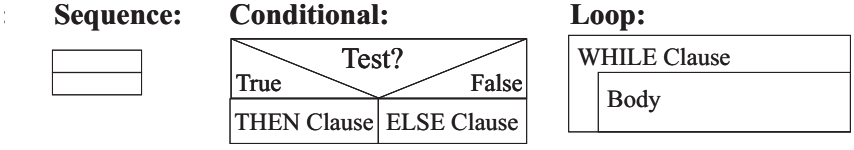

Figure 2.13: Basic Nassi-Shneiderman diagrams

As an example consider the Nassi-Shneiderman diagram of the factorial program in Figure 2.14.

```
int fact(n) { if (n>1)
   { nfact=2;
     for(int i=3;i<=n;i++)
       nfact=nfact*i;
   }
  else
   { nfact=1;
   }
  return nfact;
}
```

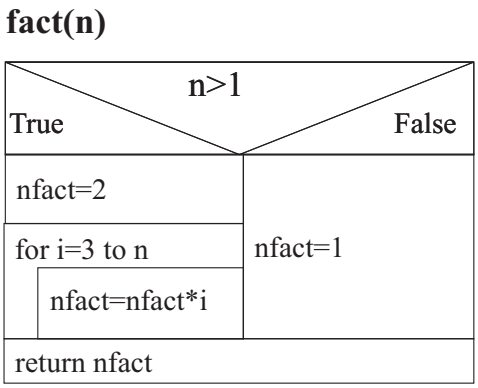

Figure 2.14: Nassi-Shneiderman diagram of a factorial program

The kinds of control flow that can be modelled with these diagrams are restricted by the fact that rectangles are always disjoint from or always fully enclosed by other rectangles. There is no overlap. A similar restriction is true for the control-flow graphs that we constructed above for our simple programming language, but if we would add jumps to our language this would no longer be the case: $G_{spaghetti} = G_{simple} \cup \{ \mathtt{S} \longrightarrow \mathtt{goto\ P}^{\mathtt{P}} \}$. Actually, with these jumps we can even specify programs which have a non-planar control-flow graph, i.e. that can not be drawn without edge crossings. Figure 2.15 shows the smallest such control-flow graph. In essence it is the complete graph with 5 nodes, i.e. all nodes are connected to each other (see Kuratowski Reduction Theorem [Tho81]). In graph theory this graph is often denoted by $\mathcal{K}_5$.

```
if (a1>0)
 { if (a2>0)
    { if (a3>0)
       { if (a5>0)
          { goto L1 }
          else
           { goto L2 } L5
        }
       else
        { if (a4>0)
           { goto L5 }
          else
           { goto L1 } L4
        } L3
    }
   else
    { goto L4 }
 } L2
else
 { goto L3 } L1
```

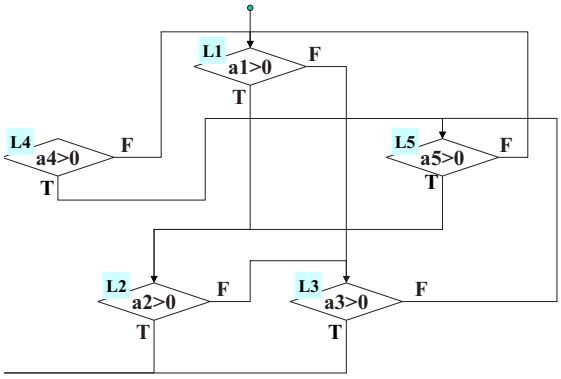

Figure 2.15: Smallest non-planar control-flow graph

Chapter 3

Visualizing the Results of Program Analyses

In this chapter we will look at static program analyses. First we will discuss why control-flow analysis, i.e. the computation of the CFG for a given program, is not always as simple as for the language that we used in the previous chapter. Then we will give an introduction to data-flow analyses and look at two such analyses in more detail. Finally we will give some examples of systems that visualize such and similar analysis results.

3.1 Static Analysis

Static analyses compute properties of a program which hold for all executions of this program. It is important to note, that not every property can be computed because of the Halting Problem

We call those properties that cannot be computed before runtime dynamic.

3.2 Control-Flow Analysis

Control-flow analysis computes the control-flow graph of a program. For the sample language in the previous chapter the algorithm presented was very easy. If we consider real programming languages the problem becomes much harder. Such programming languages typically have procedural abstraction. Each procedure has its own control-flow graph, but due procedure calls within the body of a procedure these graphs are interconnected. Thus we distinguish intra- and inter-procedural control-flow graphs. So, if we have a call of procedure q in the body of procedure p we draw an arrow from the program point of the call to the entry node of procedure q and an arrow from the exit node of q back to the call. Many modern programming languages have function pointers. They are a key feature of higher-order functional languages, but they also exist in C. The problem is that the value of such a function pointer is computed at runtime and thus it can point to all functions of the program or at least all functions of a certain type. As a consequence we have to draw edges from a program point that calls a function via a function pointer to all these functions. A similar problem occurs in object oriented languages like Java. There we do not have function pointers directly, but references to objects which contain functions or in OO lingo methods. What method is called depends on the runtime type of the object referred to. Because of this dynamic dispatch of methods the inter-procedural CFG typically contains edges to all those methods which might be called based on the static type of the reference. By computing better approximations of the runtime type of the reference the number of possible targets of a method call can be considerably reduced [Pro02].

3.3 Data-Flow Analysis

Data-flow analyses compute information for each program point about the data that will reach this program point during execution. In general data-flow analysis works by propagating locally available information over the paths in the control flow graph. We distinguish two kinds of flow problems based on the direction the information is propagated along the edges:

Forward-flow problems: what can happen before control reaches this program point, e.g. reaching definitions or available expressions?

Backward-flow problem: What can happen after control leaves this program point, e.g.: live variables, very busy expressions or reached uses?

Accordingly for each node v of the control-flow graph we compute two functions:

$IN(v)$: information about the state before the program point is executed.

$OUT(v)$: information about the state after the program point is executed.

3.3.1 Available Expressions

As an example of a forward-flow problem we look at the computation of available expressions. A binary expression $e_1\ op\ e_2$ is available at program point p if it has been computed before, i.e. it has to be computed along every path by which p can be reached. Assume that the expression e occurs at program point p and is available on all incoming paths. Let $p_1, \ldots, p_n$ be the program points where the expression was computed before. Then the program can be optimized by inserting an assignment $\mathsf{x}{=}e$ to a new temporary variable x before each program point $p_1, \ldots, p_n$ and replacing all occurrences of e in $p_1, \ldots, p_n$ and in p by x. As a result the expression is only computed once and we avoid avoid recomputation.
Let (V, E, in, out) be a control-flow graph and $\mathcal{E}$ the set of all binary expressions which occur in the program. Furthermore we define two functions GEN and $KILL$: $GEN, KILL : V \to L_G(E)$

$$GEN(v) = \begin{cases} \{e' \mid e' \text{ is a binary subexpression of } e\} & \text{if } v = e \\ \{e' \mid e' \text{ is a binary subexpression of } e \\ \quad \text{and } x \text{ does not occur in it}\} & \text{if } v = x{=}e \\ \emptyset & \text{otherwise} \end{cases}$$

$$KILL(v) = \begin{cases} \{e' \mid e' \in \mathcal{E} \text{ and } x \text{ occurs in } e'\} & \text{if } v = x{=}e \\ \emptyset & \text{otherwise} \end{cases}$$

Now the functions IN and OUT are computed by iterating over all nodes of the control-flow graph.

Algorithm 1 (Available Expressions)
$IN(in) = \emptyset$
$OUT(in) = \emptyset$
for all $v \in (V - \{in\})$ *do*
  $IN(v) = \mathcal{E}$
  $OUT(v) = (IN(v) - KILL(v)) \cup GEN(v)$
while there are changes do
  for all $v \in (V - \{in\})$ *do*
    $IN(v) = \bigcap_{(p,l,v)\in E} OUT(p)$
    $OUT(v) = (IN(v) - KILL(v)) \cup GEN(v)$

Example: Available Expressions

To illustrate the algorithm we show the steps of the computation of available expressions for the control-flow graph in Figure 3.1 below.

```
GEN(in)={}
GEN(p1)={y+2}
GEN(p2)={n/2, 2*y, n/2+2*y}
GEN(p3)={n/2, 2*y, n/2-2*y}
GEN(p4)={}
GEN(out)={}

KILL(in)={}
KILL(p1)={}
KILL(p2)={x-2*y}
KILL(p3)={x-2*y}
KILL(p4)={y+2, 2*y, n/2+2*y,
          n/2-2*y, x-2*y}
KILL(out)={}
```

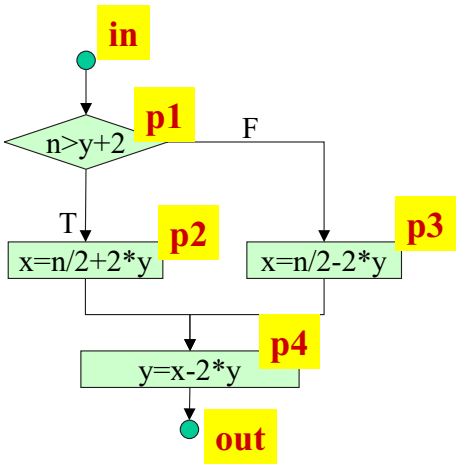

Figure 3.1: Example: Available expressions

Initialization:

```
E={y+2, n/2, 2*y, n/2+2*y, n/2-2*y, x-2*y}
IN(in)={}   OUT(in)={}
IN(p1)=E    OUT(p1)=V
IN(p2)=E    OUT(p2)={y+2, n/2, 2*y, n/2+2*y, n/2-2*y}
IN(p3)=E    OUT(p3)={y+2, n/2, 2*y, n/2+2*y, n/2-2*y}
IN(p4)=E    OUT(p4)={n/2}
IN(out)=E   OUT(out)=E
```

First Iteration:

```
IN(p1)={}                 OUT(p1)={y+2}
IN(p2)={y+2}              OUT(p2)={y+2, n/2, 2*y, n/2+2*y}
IN(p3)={y+2}              OUT(p3)={y+2, n/2, 2*y, n/2-2*y}
IN(p4)={y+2,n/2, 2*y }    OUT(p4)={n/2}
IN(out)={n/2}             OUT(out)={n/2}
```

Second Iteration:
In the second iteration no changes are computed for both functions. Thus the fixpoint is reached.

3.3.2 Live Variables

As an example of a backward-flow problem we discuss the computation of live variables. A variable x is live at program point p

- if there is a path from p to p' and x is used at p', i.e. occurs in an expression,
- and there is no redefinition of x (assignment to x) along that path.

As before, we define the two functions $GEN, KILL : V \rightarrow L_G(V)$

$$KILL(v) = \begin{cases} \{x\} & \text{if } v = x{=}e \\ \emptyset & \text{otherwise} \end{cases}$$

$$GEN(v) = \begin{cases} \{x' | x' \text{ occurs in } e\} & \text{if } v \in \{x{=}e, e\} \\ \emptyset & \text{otherwise} \end{cases}$$

Note, that for assignments the case $x' = x$ is possible.

Algorithm 2 (Live Variables)
for all $v \in V$ *do*
$IN(v) = GEN(v)$
while there are changes do
for all $v \in V$ *do*
$OUT(v) = \bigcup_{(v,l,s) \in E} IN(s)$
$IN(v) = (OUT(v) - KILL(v)) \cup GEN(v)$

Example: Live Variables

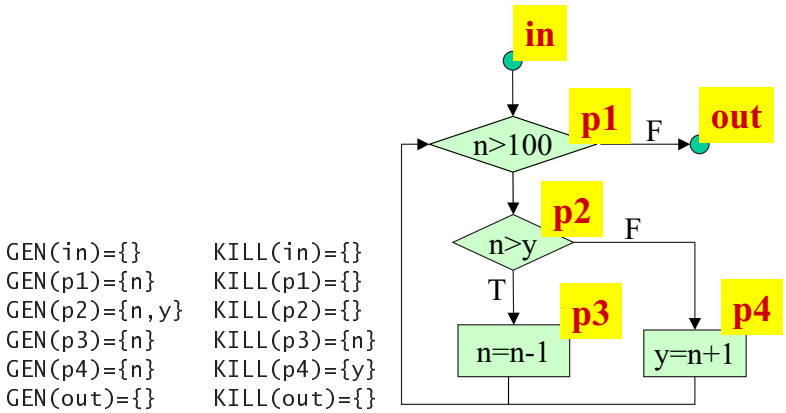

Figure 3.2: Example: Live Variables

Initialization:

`IN=GEN`

First Iteration:

```
OUT(in)={n}              IN(in)={n}
OUT(p1)={n,y}            IN(p1)={n,y}
OUT(p2)={n}              IN(p2)={n,y}
OUT(p3)={n,y}            IN(p3)={y,n}
OUT(p4)={n,y}            IN(p4)={n}
OUT(out)={}              IN(out)={}
```

Second Iteration:

```
OUT(in)={n}              IN(in)={n}
OUT(p1)={n,y}            IN(p1)={n,y}
OUT(p2)={y,n}            IN(p2)={n,y}
OUT(p3)={n,y}            IN(p3)={y,n}
OUT(p4)={n,y}            IN(p4)={n}
OUT(out)={}              IN(out)={}
```

Third Iteration:
In the second iteration no changes are computed for both functions. Thus the fixpoint is reached.

3.4 Examples of Visualization of Analysis Results

Other static analyses include for example the computation of upper bounds of the stack usage for each program point, as shown in Figure 1.6 in the introduction, the inference of types for all expressions in a program, or the computation of upper bounds of the worst case execution times for loops, functions or complete programs.

In Figure 3.3 a control-flow graph was drawn with the aiSee tool. Each program point is annotated with live variables.

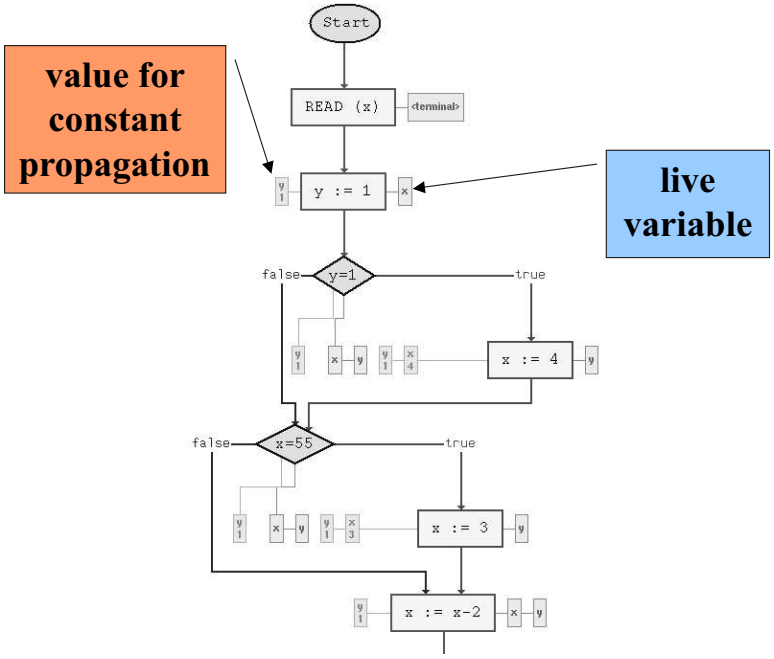

Figure 3.3: Live variables shown with aiSee

Another tool to visualize graphs for program analysis is VISTA [Lab]. It can combine different graphs in a single view. In the example in Figure 3.4 control-flow graphs (CFG), data-dependency graphs (DDG) and control-dependency graphs (CDG) are shown. In control-dependency graphs statements are only dependent on their preceding condition or the entry node. CDG are similar to Jackson Diagrams in that they show the hierarchical dependency, but there is no order on the children of a node. Compared to the layout algorithm for control-flow graph that we sketched in the previous chapter, the algorithms used by VISTA are not application-specific, but general graph drawing algorithms.

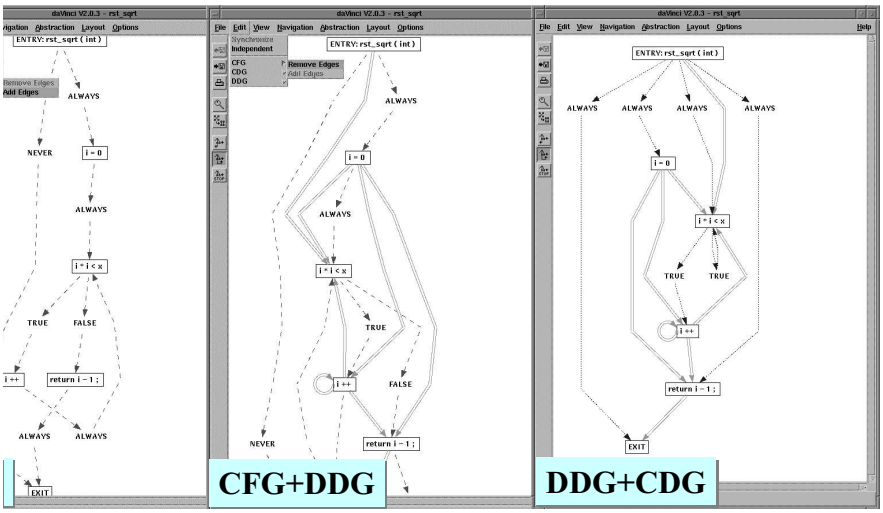

Figure 3.4: Different dependency graphs drawn by VISTA

Chapter 4

Algorithm Animation

In the previous chapters the visualization of the structure of programs and of the results of static program analysis has been discussed. Now we will look at the behavior of programs and algorithms, i.e. what actually happens at runtime. We will give a short history of algorithm animation, look at some examples and then discuss some design issues. Next some principal architectures of algorithm animation systems are characterized. Finally we discuss an approach to the visualization of the abstract execution of algorithms.

4.1 What is it about?

Algorithm animation is the visualization of the behavior of an algorithm. The term "animation" stems from the verb "to animate" which means "to bring to live". We refrain from defining the term "algorithm" [1] here. There are just too many formal and informal definitions and we leave the dispute to others ([Mos01, Gur00]). No doubt, the Church-Turing Thesis, which states that every algorithm can be computed by a Turing Machine [Tur36], constitutes a fundamental insight in computer science, but we certainly don't want to visualize the execution of all kinds of algorithms on Turing Machines. Rather we prefer computational models that are closer to the problem to be solved.

In all cases the execution of an algorithm by a real or mathematical machine leads to a sequence of states. Each step of the algorithm results in a transition from one state to another. Algorithm animations map each state into a visual representation (image) and usually show the transitions as animations between these images.

[1] The term "algorithm" is named after Abu Ja'far Muhammad ibn Musa Al-Khwarizmi, who wrote around the year 840 a treatise on algebra and a treatise on arithmetical calculation with Hindu-Arabic numerals. The Arabic text is lost but a Latin translation, Algoritmi de numero Indorum (Al-Khwarizmi on the Hindu numerals), gave rise to the word algorithm deriving from his name in the title.

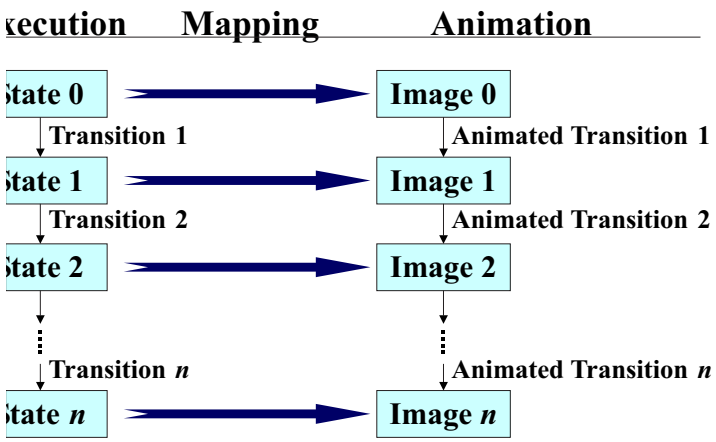

Figure 4.1: Algorithm Animation: Mapping states to images

If we take a closer look at the implementation of algorithm animation systems, we will often find that there is an intermediate layer. The state is mapped onto visual models, i.e. graphical objects or geometric data, which are then rendered to produce the images. Using this intermediate layer animations can be performed on the image level as before, but the real advantage of this approach is that the animations can be performed by continuous transformation of a model to the subsequent model.

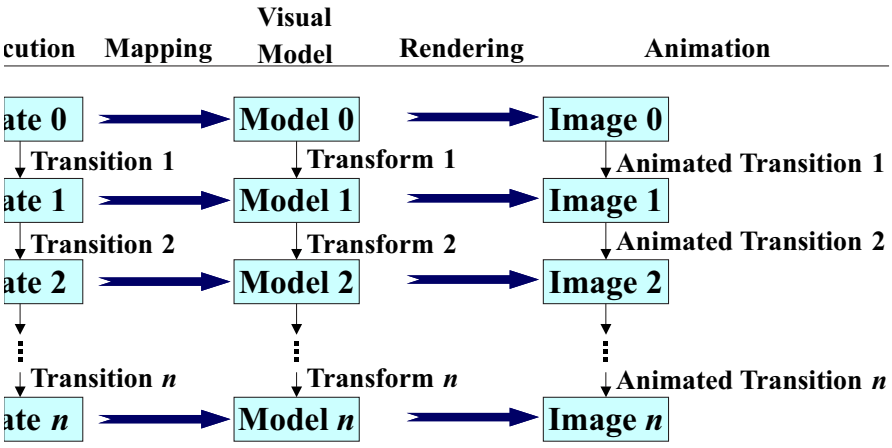

Figure 4.2: Algorithm Animation: Mapping states to models

The challenge of algorithm animation is to find the right models, i.e. appropriate graphical abstractions for states and transitions between states.

4.2 Why do people animate algorithms?

Different people have different motivations for animating algorithm. These motivations also have different requirements of the animations and the way they are produced.

Understanding, Teaching: Teachers visualize algorithms to explain them to their students.

Design: Developers visualize algorithms to better communicate the ideas to other experts.

Optimization: Developers visualize algorithms to better understand and find possibilities to enhance them.

Debugging: Programmers use visualizations to find faults in their programs.

4.3 A Short History of Algorithm Animation

Allegedly, the first algorithm animations ever produced was a movie about list processing with the language L6 [Knowlton:66]. In the sequel the educational promises have been the main motivation for the production of other algorithm animations [Hop74, Bae73]. But a real impetus of the field was the video "Sorting Out Sorting" presented at the ACM SIGGRAPH Conference in 1981 [Bae81] showing a race of 9 different sorting algorithms. Each value of the list was represented by a dot in a matrix as shown in Figure 4.3.

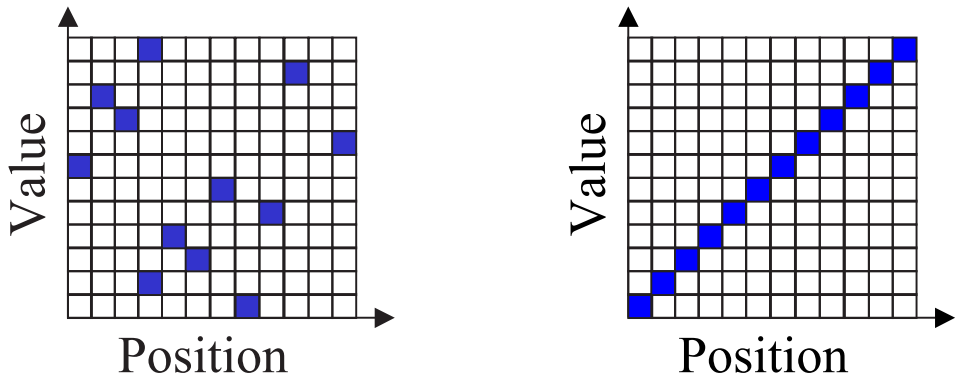

Figure 4.3: Matrix view: unsorted and sorted data

Based on experience with algorithm animations which have been developed from scratch two seminal algorithm animation tools have been developed at Brown University to ease the development of such animations: BALSA[2] [BS84] by Marc Brown (later at MIT and DEC) and TANGO[3] [Sta90a] by John Stasko (later at Georgia Tech). To some extend Brown and Stasko developed in the following years at their new institutions other systems driven by the technological advancement in other areas, in particular 3D computer graphics and networked computers.
BALSA introduced the concept of interesting events (IEs) and related to this the use of several views on the same state. Later in BALSA-II [Bro88] step and stop points were added. Finally in CAT[4] [BN96] and its successor JCAT [BR96] the views were distributed on several computers. TANGO [Sta90a] implemented the path-transition paradigm [Sta90c] to enable smooth, continuous and simultaneous animations of state transitions. TANGO is actually an interpreter for animation commands. The idea was to implement algorithms with an arbitrary programming languages and have them produce commands in the SAMBA language as textual output. To this end the program usually would print these at

[2] Brown University Algorithm Simulator and Animator
[3] Transition-based Animation Generation
[4] Collaborative Active Textbook

interesting program points either to standard output or into a file. This output was then fed post-mortem into the TANGO interpreter to produce the animation. To animate parallel algorithms a library called POLKA[5] was developed [SK93] and using POLKA an animation interpreter called SAMBA [Sta97] provided enabled post-mortem visualizations similar to TANGO. Traces were already used by PVM[6] [Sun90] to visualize parallel programs post-mortem. There traces can be collected at each computer and are merged and visualized later.

```
Marc Brown (and Marc Najork):      John Stasko:
- Balsa (1985)
- Balsa II (1988)                  - XTango/Tango (1989)
- Zeus (1992)                      - Polka + Samba (Frontend)
- Anim3D, Zeus3D (1993)            - Polka3D (1992)
- CAT (WWW, 1996), JCAT (1997)
```

Figure 4.4: Offspring of Balsa and Tango

Some seminal papers about the classical animation tools are contained in an anthology on software visualization [SDBP98] published in 1998. Today there exist many more algorithm animation systems which have been developed by other researchers, e.g. Animal, CATAI, Daphne, Ganimal, Gasp, GeoWin, Jawaa, Jeliot, Leonardo, Mocha, to name a few. An overview of the more recent developments in the field was previously published in a state-of-the-art survey on software visualization [KS02].

4.4 Some animations produced by X-Tango

In Figure 4.5 screen dumps of the animations of algorithms for binary trees, linked lists, bin packing and the n-queens problem are shown.
A binary tree is tree where each node has a maximum of 2 children. A binary tree is natural if the left child is smaller than the right child and both children are small than their parent node. Typical algorithm animations show the insertion and deletion of nodes.
In a linked list each node has a pointer to the next element. The pointer to the first element of the least is the head pointer. The next pointer of the last element of the list is undefined (null). Typical algorithm animations show the insertion or deletion of elements at the head or at the end as well as at any other position of the list.
In the bin packing problem a container with fixed height has to be filled with boxes of different sizes, such that a minimal amount of horizontal space is used. Vertically boxes can be stacked on top of each other as long as the height of each stack does not surpass the height of the container. There is an online and offline version of this problem. For the online problem each box must be placed before the next one is received, i.e. the number of boxes and the size of each of these boxes are not known beforehand. For the offline problem the number and sizes of each box are known before packing. Typical algorithm animations show the next fit, first fit and best fit strategies.
For the n-queens problem n queens must be placed on a $n \times n$ chess board such that they don't capture each other. A typical algorithm places the i-th queen on the first field of the i-th row. If the queen is captured by one of the previously placed queens, the queen is moved to the next field on the row.

[5] Parallel program-focused Object-oriented Low Key Animation
[6] Parallel Virtual Machine

Otherwise the algorithm tries to place the $i + 1$-th queen recursively. If this fails, the queen is moved to the next field and so on.

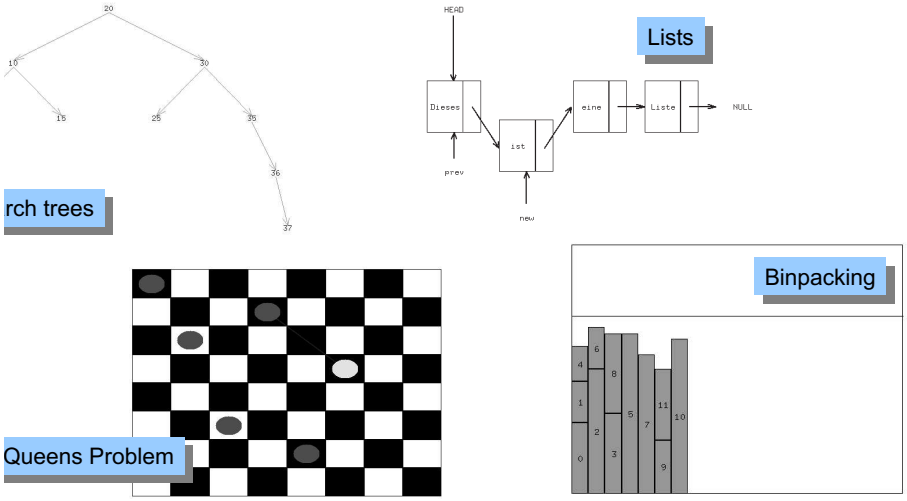

Figure 4.5: Various X-Tango animations

Figure 4.6 shows screen dumps of three animations of sorting algorithms: bubble sort, quick sort and heap sort.

Bubble sort pairwise compares the elements in a sequence from left to right and swaps them if the first element is larger than the second. At the end of the first iteration the largest element is at the end of the sequence. The process is repeated for the rest of the elements until no more elements are swapped.

Quick sort selects an element and splits the list such that the left part contains all elements of smaller or equal value and the right part all larger elements. Then these two parts are recursively sorted in the same way. In the animation the recursively constructed parts are indicated by nested boxes.

A heap is a binary tree where the value of each node is larger than that of each of its children. The heap sort algorithm constructs a binary tree and establishes this heap property such that the largest element is at the root of the tree. To get the second largest element the root is replaced by the rightmost leaf of the tree and the tree is traversed to establish the heap property again. The process is repeated until the tree is empty.

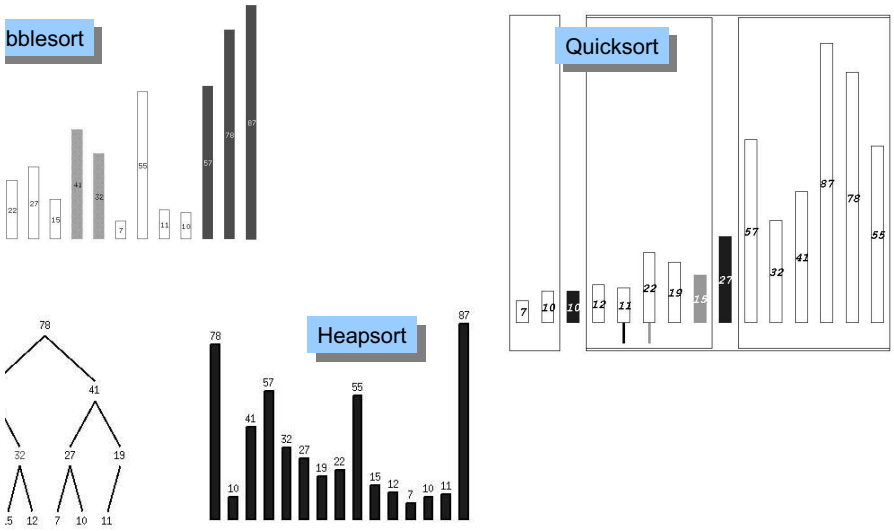

Figure 4.6: X-Tango animations of sorting algorithms

4.5 3D for Algorithm Animation

There are different reasons why people use 3D graphics for algorithm animations.

Aesthetics

Humans are used to three dimensions

Data structures or algorithms are inherently three-dimensional for 3D geometry, e.g. triangulation

3D adds additional information to a 2D representation

Multiple views of an object

History

Figure 4.7 shows a screen dump of an algorithm animation of bubble sort implemented with VRML and JavaScript. The third dimension is used to show the history of the sort, i.e. the partially sorted sequences are shown along the Z-axis.

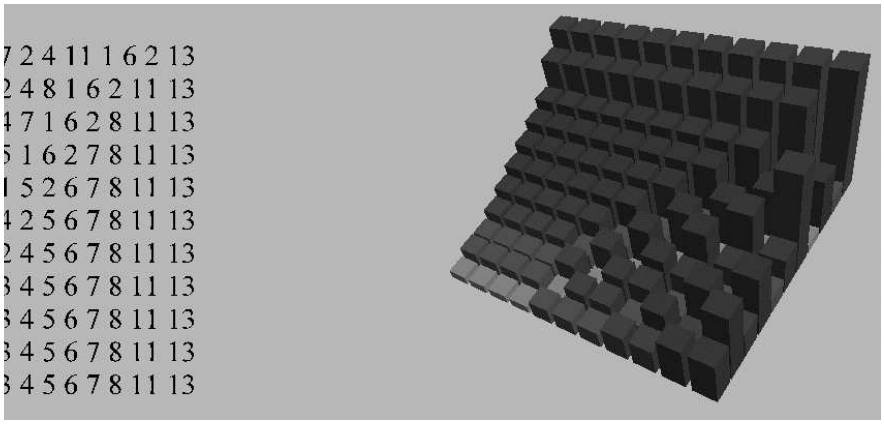

Figure 4.7: Three-Dimensional animation of bubble sort

As another example look at the 3D Animation of the Shortest Path Algorithm (SSSP Single Source Shortest Path) in Figure 4.8 which was produced with Zeus3D [BN93]. In this animation the third dimension is used to display additional information, here costs. The graph is drawn in the XY plane, while the Z axis indicates for each node the costs to get from the source node to this node. Consequently, the source node has costs 0. At the end of the algorithm the shortest-paths tree is shown where for each node the shortest path is the ascending path with lowest height.

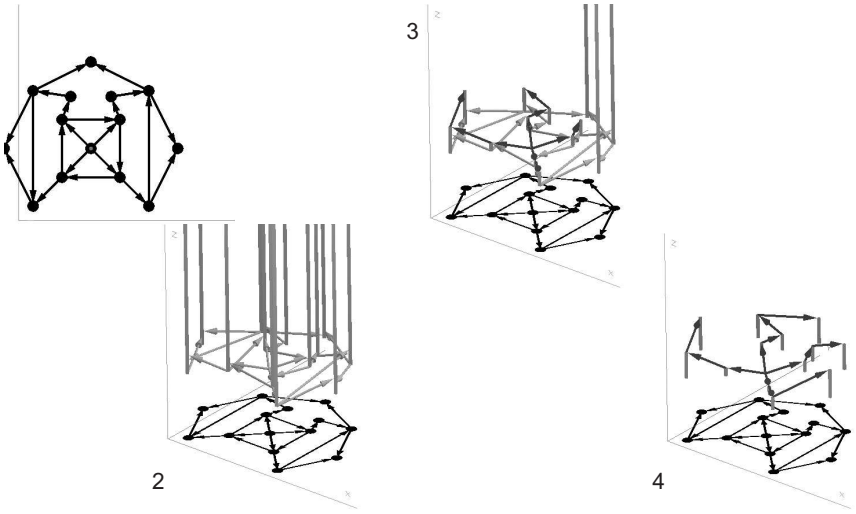

Figure 4.8: 3D Animation of SSSP

4.6 Some Design Issues

How are invariants visualized? In the 3D-Heapsort the heap-property is shown as follows: along each path the columns have increasing height.

How does focussing work? The active parts of the data structure can be drawn in a certain color, a pointer can be placed next them, they can move to a certain location on the screen, or their size can be increased (zooming).

How is recursion displayed? The depth of a recursion and the different functions invoked can be displayed by frames, colors, and even sound.

What is the goal of the animation system? Should it be easy-to-use, comprehensible or powerful?

How are algorithm and animation coupled or separated? see next section.

4.7 Architectures of Algorithm Animation Tools

Adhoc Don't use a tool at all, implement everything from scratch.

Libraries Use at least libraries with graphical abstractions, control-elements, etc.

Special datatypes Program the algorithm with datatypes which have built-in visualizations

Post-Mortem Algorithm and visualization tool are two separate applications. When the algorithm executed a trace or animation plan (typically not a full-fledged programming language) is produced and later visualized by a separate component.

Interesting Events Interesting events and multiple views Annotate essential program points with interesting events. During execution these events are sent to one or multiple views. The approach usually applies the MVC[7] design pattern.

Declarative Annotations and algorithm are separated. There are two approaches. The first is state mapping, where a demon watches state changes and updates the visualization of the state accordingly. The second approach is are constraints-based systems. They work in a similar way, but the program to be visualized itself is written in a constraints-based language.

Semantics-Directed The algorithm is executed by a visual interpreter or debugger which produces the visualizations automatically, but usually on a low level of abstraction.

Both the declarative and the semantics-directed approach are usually non-intrusive, i.e. the program code must not be changed to get a visualization of the program.
The following two examples are taken from a recent paper that compares the declarative (state mapping) and the interesting events approach [DFS02].

4.7.1 Example: Interesting Events in POLKA

Using POLKA program points are annotated at essential program points with calls to methods. These annotations are called interesting events (IEs). Whenever the program point is reached during the execution of the program point, the method sends information about the current program state to all views.

[7] Modell-View-Controller

```
void main() {
 bsort.SendAlgoEvt("Input",n,v);
 for(j=n; j>0; j--)
    for(i=1; i<j; i++)
        if (v[i]>v[i+1])
           { int temp= v[i];
              v[i]=v[i+1];
              v[i+1]=temp;
              bsort.SendAlgoEvt("Exchange",i,i+1);
           }
}
```

4.7.2 Example: Declarations in LEONARDO

LEONARDO[DF] integrates both a C and a Pascal compiler together with a source code editor. It uses a virtual machine with invertible instructions to visualize a program and provide undo/redo functions of every execution step. The visualization is separated from the program by writing the visualization declarations are as comments in the program code. The declarations are written in a kind of logic programming language called ALPHA. It describes a set of visual objects. An alpha program consists of a sequence of predicates that define these objects and their relationships. Each predicate is defined by a head-body-pair where the head specifies the name and formal parameters while the body specifies the computation.

```
void main() {
 for(j=n; j>0; j--)
    for(i=1; i<j; i++)
        if (v[i]>v[i+1])
           { int temp= v[i];
              v[i]=v[i+1];
              v[i+1]=temp;
           }
}

/** View(Out 1); Rectangle(Out ID, Out X, Out Y, Out L Out H, 1);
    For N: InRange(N,0,n-1)
        Assign X=20+20*N   Y=20  L=15  H=15*v[N]  ID=N;
**/
```

4.8 Abstract Algorithm Animation

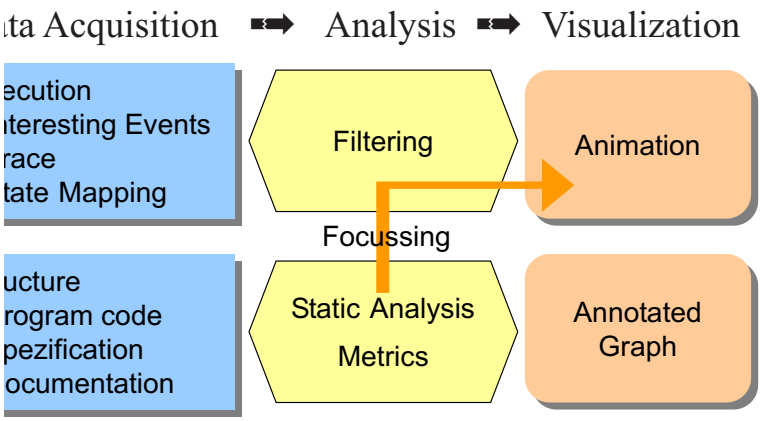

Figure 4.9: Using program analysis to focus algorithm animations

Focussing is a big problem in algorithm animation. What part of the data structure should be displayed on the screen? Not all data structures should be displayed at all times and often the amount of data makes it impossible to show all data simultaneously. So we have to manage the screen estate carefully.

Usually at each program point only a small fraction of data can be accessed. One solution to determine this fraction before run-time by using static program analyses which compute information about the accessible data structures for each program point. Such an analysis called has been developed by Sagiv, Reps and Wilhelm [SRW96, SRW99] and is called shape analysis. It computes an abstract representation of linked data structures, which focusses on the active parts of this structures. For each program point it yields a finite set of shape graphs.

Braune and Wilhelm suggest to animate the abstract execution based on shape graphs [BW00]. They call the resulting animations algorithm explanations to emphasize that they show the invariants of the data structures at each program point. Recently the approach has been extended to also show non-structural invariants [WMS02].

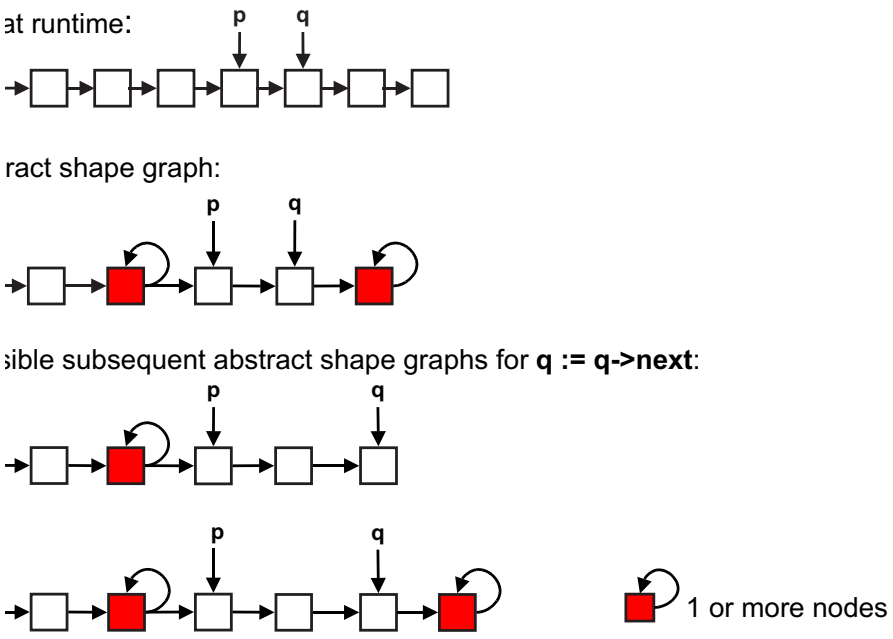

Figure 4.10: Abstract execution of `q := q->next`

In Figure 4.10 the abstract execution of a program point is shown. The abstract shape graph before the execution shows that the pointer `q` points at an elements immediately after the element pointed to by `p`. Furthermore there is at least one element after the element pointed to by `q`. If we now execute the assignment `q:=q->next`, then there are two possible subsequent abstract shape graphs. The first shows the case that there was exactly one element more in the list, while the second shape graph represents the case where there where at least two more elements.
Note, that a transition from an abstract state as_1 to an abstract state as_2 is only legal, if a transition from a concrete state cs_1 to a concrete state cs_2 exists where cs_1 is represented by as_1 and cs_2 by as_2. Visual abstract execution must only show legal transitions.

Bibliography

[Bae73] R. Baecker. Towards Animating Computer Programs: A First Progress Report. In *Proceedings of the Third NRC Man-Computer Communications Conference*, 1973.

[Bae81] R. Baecker. Sorting out Sorting. 30 minute color film (developed with assistance of Dave Sherman, distributed by Morgan Kaufmann Pub.), 1981.

[BK01] Sarita Bassil and Rudolf K. Keller. Software Visualization Tools: Survey and Analysis. In *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC2001)*, Toronto, Ontario, Canada, 2001.

[BM90] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. Addison-Wesley, Reading, MA, 1990.

[BM98] Ronald M. Baecker and Aaron Marcus. Printing and publishing c programs. In *Software Visualization – Programming as a Multimedia Experience [SDBP98]*. 1998.

[BN93] Marc H. Brown and Marc Najork. Algorithm animation using 3d interactive graphics. In *Proceedings of ACM Symposium on User Interface Software and Technology*, 1993.

[BN96] M. Brown and M. Najork. Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom. In *Proceedings of the 1996 IEEE International Symposium on Visual Languages*, Boulder, CO, 1996.

[BR96] M. Brown and R. Raisamo. JCAT: Collaborative Active Textbooks Using Java. In *Proceedings of CompuGraphics'96*, Paris, France, 1996.

[Bro88] M. Brown. Exploring Algorithms with Balsa-II. *Computer*, 21(5), 1988.

[BS84] M. Brown and R. Sedgewick. A system for Algorithm Animation. In *Proceedings of ACM SIGGRAPH'84*, Minneapolis, MN, 1984.

[BW00] B. Braune and R. Wilhelm. Focussing in algorithm explanation. *Transactions on Visualization and Computer Graphics*, 6(1), 2000.

[Chi00] Ed H. Chi. A taxonomy of visualization techniques using the data state reference model. In *Proceedings of the Symposium on Information Visualization (InfoVis '00)*, Salt Lake City, Utah, 2000. IEEE Press.

[DF] Camil Demetrescu and Irene Finocchi. `http://www.dis.uniroma1.it/~demetres/Leonardo"`.

[DFS02] Camil Demetrescu, Irene Finocchi, and John Stasko. Specifying Algorithm Visualizations: Interesting Events or State Mapping? In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*. 2002.

[Die02a] Stephan Diehl. Future Perspectives. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*. 2002.

[Die02b] Stephan Diehl, editor. *Software Visualization*, volume 2269 of *LNCS State-of-the-Art Survey*. Springer Verlag, 2002.

[EB02] Alexander A. Evstiougov-Babaev. Call Graph and Control Flow Graph Visualization for Developers of Embedded Applications. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*. 2002.

[Gur00] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1), 2000.

[GvN47] H. H. Goldstine and J. von Neumann. Planning and Coding of Problems for an Electronic Computing Instrument, 1947. Part II, vol I of a report prepared for the U.S. Army Ord. Dept., reprinted in [Tau65].

[HDS02] Christopher Hundhausen, Sarah Douglas, and John Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(2), 2002.

[Hop74] F. Hopgood. Computer Animation Used as a Tool in Teaching Computer Science. In *Proceedings IFIP Congress*, 1974.

[IS73] I.Nassi and B. Shneiderman. Flowchart Techniques for Structured Programming. *SIGPLAN Notices*, 12, August 1973.

[Jac75] Michael Jackson. *Principles of Program Design*. Academic Press, 1975.

[JPH+99] C. Johnson, S.G. Parker, C. Hansen, G.L. Kindlmann, and Y. Livnat. Interactive simulation and visualization. *Computer*, 12, 1999.

[Knu84] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2), 1984.

[Knu92] D. E. Knuth. *Literate Programming*. Center of the Study of Language and Information - Lecture Notes, No. 27. CSLI Publications, Stanford, California, 1992.

[Kos02] Rainer Koschke. Software Visualization for Reverse Engineering. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*. 2002.

[KS02] Andreas Kerren and John T. Stasko. Algorithm Animation. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*. 2002.

[Lab] Cigital Labs. `http://www.cigitallabs.com/research/demos/vista/`.

[Mos01] Yiannis N. Moschovakis. What is an Algorithm? In Bjorn Engquist and Wilfried Schmid, editors, *Mathematics Unlimited – 2001 and Beyond*. Springer Verlag, 2001.

[Mye90] B. Myers. Taxonomies of visual programming and program visualisation. *Journal of Visual Languages and Computing*, 1, 1990.

[OWE96] M. J. Oudshoorn, H. Widjaja, and S. K. Ellershaw. Aspects and taxonomy of program visualisation. In Peter D. Eades and Kang Zhang, editors, *Software Visualisation*, volume 7, pages 3–26. World Scientific, Singapore, 1996.

[PBS93] B. A. Price, R. Baecker, and I. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[Pro02] Christian Probst. *A Demand-Driven Solver for Constraint-Based Control Flow Analysis*. Phd thesis, University of Saarland, Saarbrücken (Germany), 2002.

[RR93] G.C. Roman and K.C. Roman. A taxonomy of program visualization systems. *Computer*, December 1993.

[SDBP98] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price. *Software Visualization*. MIT Press, 1998.

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of 1996 IEEE Conference on Visual Languages*, Boulder, CO, 1996. IEEE Press.

[SK93] J. Stasko and E. Kraemer. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18(2), 1993.

[SRW96] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM SIGACT-SIGPLAN Symposium on Priciples of Programming Languages*, St. Petersburg Beach, Florida, 1996.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape-analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Antonio, Texas, 1999.

[Sta90a] J. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23(9), 1990.

[Sta90b] J. T. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23(9):27–39, 1990.

[Sta90c] J. T. Stasko. The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.

[Sta97] J. Stasko. Using Student-Built Algorithm Animations as Learning Aids. In *Proceedings of the 1998 ACM SIGCSE Conference*, San Jose, CA, 1997.

[Sun90] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, 2(4), 1990.

[Tau65] A. H. Taub, editor. *John von Neumann: Collected Works*, volume V: Design of Computers, Theory of Automata and Numerical Analysis. Pergamon Press, New York, 1965.

[Tho81] C. Thomassen. Kuratowski's theorem. *Journal of Graph Theory*, 5, 1981.

[Tur36] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society, Series 2*, volume 42, 1936.

[WMS02] Reinhard Wilhelm, Tomasz Müldner, and Raimund Seidel. Algorithm Explanation: Visualizing Abstract States and Invariants. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*. 2002.