Principles for Design of Software Engineering
Visualization Tools

by

Jeffrey Howard

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2000

©Jeffrey Howard, 2000. All rights reserved.

Certified by...

Nancy G. Leveson
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by...

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Principles for Design of Software Engineering Visualization Tools

by

Jeffrey Howard

Submitted to the Department of Electrical Engineering and Computer Science on August 3, 2000, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science

Abstract

Research in software engineering produces analysis techniques well ahead of the state of industry. Sophisticated static analysis of source code and dynamic observation of running code lend considerable insight into a system’s workings. When automated, these analysis algorithms provide, in minutes, information that may take hours for programmers to construct by hand. The failure to incorporate these techniques into industry traces, in part, to a lack of tool automation. Researchers rarely develop their proofs into software implementations. Those that do ignore human factors to the extent that considerable knowledge of their algorithms is required to interpret the tool output. This thesis contains a survey of current research in visualization for software engineering, develops guidelines for good visualization systems, and tests these guidelines by applying them to a sample problem: the simulation of a state-machine based specification.

Thesis Supervisor: Nancy G. Leveson
Title: Professor of Aeronautics and Astronautics
Acknowledgments

I would like to thank the following people for their assistance, support, and encouragement.

My supervisor, Professor Nancy Leveson, took me on as a student. She had little advance warning, stepping into circumstances that cast her as a supporter or advocate as much as an advisor. Thank you.

When I decided to major in computer science, a housemate took me aside and said, “You’re going course 6? Anne Hunter is your best friend. Do everything you can to make her life easy, because she might just save yours some day.” I am graduating, in part, because of her herculean labors on my behalf. Before I leave MIT, I plan to find a student just joining the EECS department, so I can repeat the advice given to me.

Patrick Anderson has been an invaluable thorn in my side. I’ve never met someone who could make me so happy to hear: “You could say that, but, of course, you’d just be completely wrong.”

Heather Grove gave me all the encouragement and support of a thousand witty sayings about goals and the paths to reach them. She did so without a single cliche or quotation from Tennyson. But for her, I would have no priorities.
## Contents

1 Introduction \hspace{1cm} 6
   1.1 Formal Methods \hspace{1cm} 7
   1.2 Computational Support \hspace{1cm} 9
      1.2.1 Support for Formal Methods \hspace{1cm} 11
      1.2.2 Support for Other Methods \hspace{1cm} 12
   1.3 Visualization and Human-Computer Interaction \hspace{1cm} 14
      1.3.1 Graphical Representation without Visualization \hspace{1cm} 15
   1.4 Simulation \hspace{1cm} 16
      1.4.1 Design as a Motivation \hspace{1cm} 17

2 HCI and Visualization \hspace{1cm} 19
   2.1 Printed Graphics \hspace{1cm} 19
   2.2 Editors \hspace{1cm} 22
   2.3 Visual Languages \hspace{1cm} 26
   2.4 Debuggers and Profilers \hspace{1cm} 27
      2.4.1 Event Graphs \hspace{1cm} 27
   2.5 Design Aids \hspace{1cm} 30
   2.6 Algorithm Animation \hspace{1cm} 32
   2.7 Graphical Elements and Representation \hspace{1cm} 33
   2.8 Visual Feedback in Direct Manipulation \hspace{1cm} 34
   2.9 Data Dictionaries \hspace{1cm} 36
   2.10 Extensions for Collaborative Systems \hspace{1cm} 37
   2.11 Guidelines and Methods \hspace{1cm} 38
2.11.1 Process .................................................. 38
2.11.2 Architecture ............................................. 39
2.11.3 Usability .................................................. 39
2.11.4 Visualization ............................................. 40

3 State Machine Simulation .................................... 43

3.1 Simulation Representation .................................. 45
  3.1.1 Model ................................................... 45
  3.1.2 Signal .................................................. 46
  3.1.3 Specifier ............................................... 48
  3.1.4 Condition ............................................... 48
  3.1.5 Miscellany ............................................. 48

3.2 Simulation Operation ....................................... 49

3.3 Methods of Visualization .................................. 51
  3.3.1 Analogy ................................................. 51
  3.3.2 State Machine Display ................................ 52
  3.3.3 Details on Demand .................................... 53
  3.3.4 Texture, Color, and Shape ............................ 54
  3.3.5 Timing and Navigation ................................. 54
  3.3.6 Annotation Mechanism ................................. 57
  3.3.7 Design Support ........................................ 57
  3.3.8 Configuration .......................................... 58
  3.3.9 Direct Feedback ....................................... 58

3.4 Contributions ............................................... 58
Chapter 1

Introduction

The tools of software engineering develop in fits and starts. A thriving industry develops and markets Computer Aided Software Engineering (CASE) applications. Despite the successes described by advertising copy, many of these tools become shelfware [22]. Those that see use do not guarantee productivity improvements for their users. Many programmers continue to work with tools no more sophisticated than a text editor. Visualization, a subset of CASE in general, fares no differently.

Programmers work hours and days to come to realizations that an algorithm could illustrate in minutes. Research in formal methods, static analysis, theorem checking, and the like provides the ability to determine many properties of a program. Unfortunately, these methods rarely penetrate production environments. The academic setting that produces analysis methods prizes further theoretical research. Production quality code and ease of use lag far behind other priorities.

Slowly, the ideas recorded in journals and conference proceedings work their way into software engineering tools. Software engineers ignore many of these tools. One theory explains that the tools are often tied up in the mathematics of the methodology. Tools with indecipherable output are quickly abandoned by practicing programmers. To successfully transfer new analysis methods from academia to industry, the tools implementing those methods must offer powerful visualization capabilities. They must be straightforward to use and understand.

Human-Computer Interaction broadly covers all aspects of user interface design.
Software engineering visualization occupies a much smaller application niche. The HCI community successfully distills guidelines, heuristics, and rules of thumb for graphical user interface design [68]. Visualization in software engineering, specifically, seems to have no analogous catalog of successful methods and mechanisms.

Chapter 1 of this thesis introduces an overview of the ways in which software visualization tools support software engineering. Basic principles of human computer interaction, as the thesis relies on them, are also introduced. Chapter 2 delves into a survey of research that proposes visualizations of software structure and behavior. Specifics, such as dialog-box button layout and context-sensitive help entries are ignored in favor of distinguishing features and general principles. The second chapter concludes with a list summarizing the discovered guidelines. Chapter 3 tests the guidelines uncovered in chapter 2 by applying them to the design of a finite-state machine specification simulator.

1.1 Formal Methods

Formal methods are attempts to incorporate mathematical rigor into computer science. Facets of a program's structure or behavior map to mathematical constructs. Rigorous mathematical reasoning leads to some conclusion about the software under scrutiny. Researchers of formal methods extol the virtues of this approach to software development. The firm, mathematical foundation of their theories provides firm guarantees about the behavior of software (assuming the mathematical reasoning is not flawed).

These methods remain largely ignored by practicing software engineers [26]. The mathematics is too time consuming, and in some cases too intimidating, to incorporate into the general practices of programming [19]. Formal methods have a redeeming feature in this respect – the strict rules they obey are often convenient for coding into an automated process. In fact, many theoreticians do a small amount of coding to demonstrate the validity of their ideas. When properly constructed, software tools insulate the software engineer from the mathematical details of a formal analysis. The
tools' user interfaces and data visualization methods create this sense of insulation. They must communicate results clearly enough that the user can understand and employ the output of an analysis without necessarily understanding the theory that makes the analysis work.

Theoreticians counter this argument by pointing out that one should understand the underlying algorithm anyway. Most algorithms break down in corner cases. Understanding the theoretical underpinnings of an analysis lets the user recognize when it does not apply successfully. A cornerstone of many science curricula, computer science included, is developing the intuition to know when a result looks wrong, requiring further investigation.

However, the “theory is good for you,” argument has its own flaws. Even for those with an understanding of the underlying algorithm, good user interface design prescribes that such an understanding should not need to be at the forefront of short term memory to use the tool. If anomalous cases must be more thoroughly investigated, the guidelines to recognize those anomalies should be as short and easy to remember as possible. Additionally, many software engineers already use tools they do not fully understand. As an example, many systems engineers who design, write, and debug code daily would be hard pressed to describe the LALR parsing algorithm their compiler uses. Lastly, software engineering, like any engineering discipline, applies to a bumpy world of approximations and non-idealities. Ninety percent solutions abound, and an automated analysis tool that works most of the time is often good enough.

In essence, formal methods seem to work best as blackbox abstractions. The failure to make these methods easy to use explains their lack of adoption into software engineering practice. Good evidence for this argument comes from examination of an environment where new analysis methods thrive. In compilers, a software module is the consumer of analysis output. New analysis tools require no user interface at all. They merely need to provide the API expected by the compiler. This is convenient enough that many new methods are couched in terms of how they can be applied within optimizing compilers [58].
1.2 Computational Support

Whether or not formal methods are applied, software support is invaluable to software engineering process [39]. Software engineering research proposes a variety of processes for developing software [55]. The least evolved, and least successful, process has no formal structure at all. Uncontrolled hacking produces wild spaghetti code. Such a program may or may not meet its requirements, since none were recorded. Lack of written requirements is fine, in a sense, as no testing is done anyway.

Logically organizing development leads to the waterfall model of development [25]. In the waterfall model, development teams first perform a detailed requirements analysis. When the requirements are frozen, the design phase begins. When finished, design documentation is handed off to the implementation team; generally, they also perform unit tests on the software. Finally, the whole system is integrated for system test. After the system is deployed, it is supported and maintained. An overwhelming amount of detail must be managed, including requirements specifications, design documents, and multiple versions of code. Tests must be created, documented, performed, and outcomes recorded.

Some methodologists suggest that the best use of the waterfall model is in a series of “mini-waterfalls.” Each cycle controls the project in successive waves of requirements gathering, design, and implementation [38]. These small cycles allow workers to learn from the mistakes of past cycles and correct mistakes in future work on the project. The smaller the individual cycles, the more overhead is required for planning and tracking activities [42]. Because the plan is more complicated, so is comparing projected progress with actual; this complexity and overhead is the direct cost of the additional flexibility to adjust and correct mid-course.

Taking the mini-waterfall idea to an extreme yields something that looks similar to the starting point: controlled hacking. Because testing, peer review, and small development cycles are advantageous, one method suggests that they should be pushed to the point of being near constant activities. Beck’s “Extreme Programming” method features programming in pairs, for immediate peer review, and daily or twice daily
builds with full regression tests to assure correctness [6]. These ideas are not without precedent; nearly three decades earlier, Mills suggested that embedded systems be built as a continuously functioning system [50]. Systems were kept functional at all times by starting with a high-level main loop supported by stubs. Constructing the system meant fleshing out the stubs until all system functionality was completed.

Regardless of method, tool support is of great value [56]. Many details require tracking during software engineering. Requirements must be gathered and categorized. Design documents and diagrams must be produced. Source code must be written and stored in a version control system. Test cases must be generated, stored, and results tracked. These applications focus only on the capacity of computers to act as a database of details [66]. The real power of software engineering support lies in having tools do tiresome or error-prone analyses automatically [20]. Better than just tracking test result, a tool can automatically perform the tests and compare the results to expected outcomes. With more advanced analysis, some tools determine what inputs are necessary, automatically generating test data patterns [2].

During the 1980’s, many software vendors observed the potential market for software engineering tools. A craze of computer-aided software engineering (CASE) tools was born. The wave of tools, each promising a silver bullet for software engineering, met with disappointment. Unfortunately, as vendors promoted their tools based on sales figures, little data substantiated the usefulness of the tools, once bought [32]. The problems of software engineering were not magically solved by any particular CASE tool set [12]. The CASE tool trend left many developers with harsh feelings towards tool support in general.

Studies of visual programming languages [80] suggest that software engineers may not be a good source of information about what tools best aid them. Programmers generally prefer tools with which they are already familiar. A recent MIT doctoral graduate has said that, “a good programmer just using emacs is better than a mediocre programmer with all the fancy CASE tools in the world,” [28]. In essence, the question he addressed is irrelevant. The question should be whether tools can be made such that the same programmer (whether expert or incompetent) will perform
better with the tool than without. Ignoring marketing hype promising a panacea, automated tool support can offer software engineers insightful and labor saving views on their projects.

1.2.1 Support for Formal Methods

Many different mathematical models and methods belong under the banner of formal methods. The quality of software tool support varies greatly across the breadth of formal methods. Some cannot be successfully implemented in a software tool; others have been common since the early days of electronic computation.

Static analysis is one major category under formal methods. Static analysis is an integral part of any optimizing compiler. The source, scanned and parsed, is presented as an abstract syntax tree for a variety of analysis modules. These modules improve the runtime performance of the source code while preserving its semantics. Much of the work done by the analyzers involves tracking branching points, procedure entry and exit, loop invariants, and block structure. The properties discovered by static analysis often overlap with those found by programmers by manually reading source code. Given good visualization capabilities, the output of static analysis may be used more broadly in software engineering.

Syntax-directed editors are becoming more common. Integrated Development Environments (IDE’s) dominate the CASE market in number and variety. Most IDE’s feature a text editor with menu options for compiling, debugging, and integrating with repository systems. To enhance their appeal, many IDE vendors enhance the editor component by making it aware of the syntax rules of at least one programming language. In a simple form, this means highlighting identifiers, reserved words, and symbols in different colors and fonts.

More sophisticated analysis increases the power of these editors. A dependence graph [36] simplifies isolating differences between versions of a program. The current least common denominator, implemented in programs like diff, uses line by line comparison of source code text. Additionally, editors could illuminate paths of dependency between modules, illustrating possible call chains.
Automated analysis can also prove simple correctness properties. Dependencies between variables can be established. Programs can be proven to uphold invariants on data structures. Testing also benefits from better tool support. Code coverage tools analyze paths through code, surveying which statements will be exercised by current test data and which will not. In some cases, source code may be simple enough that automated boundary analysis can determine good test values.

Program slicing is a particularly powerful example of analysis [37]. A software engineer selects a line of source code and a variable. The slicer determines the full set of program statements that could affect the value of the variable at that line in the program. If a variable has an unexpected and erroneous value, slicing provides a clear picture of all the statements that may have affected that value. Reverse slicing is possible also. In this case, the analysis determines all the uses of the value in the program that may be affected by the selected statement. One could check that a statement cannot have harmful side effects on key variables. There are additional variations. Some slicing algorithms require that the variable be used in the statement selected; others do not. Some slicing programs opt for a much simpler approach, merely culling out all statements in a program that directly reference a variable.

1.2.2 Support for Other Methods

The simplest computer support for software engineering uses the computer as a bookkeeping tool [66]. Management of project documents, staffing assignments, and document review are three of the most time consuming tasks [13]. In simple cases, off the shelf software may solve these problems. Databases or spreadsheets track staff assignments and documents produced. Email and web sites communicate the schedule for development and review. Special purpose workflow management software improves on this by providing one application that tracks all artifacts associated with a project. Workflow management applications also accommodate staff associated with a project. The software routes documents as needed between staff members. These bookkeeping services free staff to spend their time more productively, rather than keeping track of what it is they should do next.
Software engineers also rely on automated aids to debugging. The simplest debugger is a print statement added by a programmer. While this is surprisingly effective in a wide range of circumstances, a debugger can present the same information while providing more control over the program as it runs. The programmer can peek at values in memory, change them while the program runs, and step through execution a line at a time.

Visual GUI prototyping is another tool driven method for simplifying software engineering. GUIs are more costly to develop than text interfaces. GUI components rest in a two dimensional plane on screen, but a linear textual language specifies their positions and relationships. To accomplish this, programmers maintain a correspondence between abstract numerical coordinates and screen positioning [51]. Often, this breaks down to trial and error. A programmer compiles code for a GUI screen, inspects it, adjusts numerical constants, and repeats the cycle. GUI prototyping tools help programmers develop user interfaces more quickly. The prototyping tool provides ways to draw GUI screens, and a back-end produces a code skeleton to render the screen.

Visual GUI prototyping tools evolved to incorporate a code skeleton for event handling. These tools add basic logic for application event handling, confining programmers to adding code snippets that respond to button clicks and menu selections. Vendors call these tools visual languages. Detractors call it “programming by Madlibs” [3]. Microsoft’s Visual Basic [18] is one widely recognized example. These languages incorporate a great deal of work through a GUI, but calling them visual languages classifies them poorly. The programmer still constructs the functionality of the system with text source code. The visual portion of the language applies only to laying out the graphical user interface widgets.

True visual languages use graphical representations for almost all aspects of language syntax, including basic flow control constructs and operations on primitive types. Formalisms for visible language syntax bear resemblance to the formalisms for textual grammars. Relation Grammars construct sentences using symbols and the spatial relationships between those symbols [21]. Symbols have productions and
relations have evaluation rules.

1.3 Visualization and Human-Computer Interaction

Given the potential advantage of sophisticated analysis in software engineering, it seems incongruous that so little has been incorporated into the average programmer's daily working environment. Most analysis sees little use because of poor human-computer interaction: "the interaction with the user is usually an afterthought rather than a primary design criterion" [49]. Researchers produce analysis algorithms in an environment that rewards soundness of theoretical presentation. Correctness and rigor of the proof are paramount; implementation is a secondary concern. If any program implements the new algorithm, it is likely to be a small demonstration version. These demonstration implementations are often the minimum necessary to test the theory. Necessary features for a production version of the tool are never considered. Good user interface design simply isn't a concern; the only user to encounter the program is likely to be the researcher developing it. Even outside the academic community, integration of good human factors into software engineering is an open topic [29] [77] [45].

For use by the software engineering industry at large, analysis tools must clearly and understandably communicate results to a programmer. The conventional model of an analyzer inside a compiler is not sufficient. Tools must produce human-readable results, not just data structures for other modules. An ML compiler provides an excellent example of the problem [49]. ML does not require data types in variable declarations. Instead, ML works up from primitive operations that the variables participate in. By running a constraint propagation algorithm, ML automatically deduces the most appropriate data type possible for a given variable name. Type reconstruction reduces the chance of type-related errors on the programmer's part while retaining the safety of a strongly typed language. However, the reference ML
compiler produces abstruse error messages when warning programmers of inconsistent type constraints. A strong and beneficial analysis can be crippled by an obfuscated interface.

Software engineering education actively experiments with visualization techniques [81] [40]. Educational institutions have produced a number of sophisticated systems for algorithm animation. These systems aim to help students understand basic data structures and algorithms. Students step forward and backward through a program’s actions at their own pace. Custom visualizers are deliberately used rather than a debugger. The visualizers emphasize understanding as a whole, whereas debuggers are optimized for investigation of details.

Good visualization of analytic results addresses only a portion of the effort required to make good tools in support of software engineering. The field of human-computer interaction covers the full range of training, usability testing [31], documentation [1] [71], input devices [72] [85], and command format. Even seemingly innocuous features such as the undo mechanism come under scrutiny [8]. Some researchers are even working on tools to post-process displayed information for trend detection [47].

1.3.1 Graphical Representation without Visualization

In some cases, researchers have even tried to extract the benefits of visual representation without providing graphical interaction with their tool. One example is a cliche-based program editor [78]. Programmers think in terms of larger constructs than simple statements. A skilled software engineer thinks in terms of entire loops, blocks, and similar idioms. Or at an even higher level, a programmer might consider whole modules at a time. The cliche-based editor exploits this by representing commonly used constructs, cliches, as plan diagrams.

Plan diagrams depict computations as nodes in a graph. They connect computations with arrows representing data flow. Plan diagrams are similar to flowcharts. Cliches are drawn by putting a box around a plan and placing ports at the top and bottom for input and output respectively. Ports are represented by small filled circles, and they are analogous to arguments passed into or returned from a procedure. When
one plan nests inside another, any unbound ports in the nested plan are promoted to being ports of the enclosing plan.

A user instantiates these plans in an editor. The ports are filled in with the particulars of an implementation. Usually, this implies assigning input and output ports to variables in the program. However, the editor itself is entirely textual. The editor represents the cliches internally in a data structure that is consistent with the graphical notation of plan diagrams, but the user only interacts with the textual programming language input and output of the tool.

1.4 Simulation

Visualization of simulation data can be conducted in two ways [7]. In the first, the simulation feeds data to the visualization module. This direct method often includes controls that feed back from the visualizer to the simulation. These controls allow the user to influence the simulation as it proceeds. Active controls also allow others to join and depart viewing of the simulation collaboratively [65]. Collaboration and control are not possible in the second method, offline visualization of collected data. The simulator runs, recording information in a data store. The visualization tool reads the data store and renders the data on screen. This prohibits control of the outcome of the simulation, because the simulation is no longer running. However, greater flexibility and control is provided over the visualization. The user is free to move forward or backward in time, changing views on the data.

Ideally, a hybrid mode may be used. The simulation proceeds in real time, giving the user opportunity to guide or change it. Afterward, the collected data persists, permitting more detailed and flexible viewing. An advanced user interface would allow a user to switch between modes. The user could remain in the past of the simulation and study some portion in detail, then skip the intervening time to catch up to the present to continue guiding the course of the simulation.
1.4.1 Design as a Motivation

Software design errors are particularly costly, yet Guindon claims that design remains unsupported by methods and tools [30]. Although this claim came before the object-oriented “methodology wars” of the late nineties, the cost of design errors remains as high. Part of the difficulty of constructing design support tools comes from the fuzzy and heuristic nature of design. Software design experts do not think in terms of individual programmatic statements. Expert reasoning is also not rule-based, to the detraction of rule-based expert systems. Expert software engineers seem to use case-based reasoning. They work by decomposing problems into smaller problems. When the smaller problems resemble familiar ground, well-known solutions may be adapted to fill the new role. These solutions are procedural plans to accomplish a particular task [86]. Break-down in a designer’s ability is associated with stretching beyond the range of known solutions or requiring too many to be combined to solve the larger problem.

Several suggestions focus on augmenting the designer’s ability to work with a larger range and number of solution patterns [30]. This assistive work creates opportunities for good visualization and navigation. The first suggestion is a library of design schemas. The library requires a good navigation system to browse and choose appropriate designs. The self-named “patterns community,” a group of software engineers who have rediscovered the idea of reusable design elements (in an object oriented style), describe their biggest stumbling block as a failure to categorize patterns into a useful library organization. The second suggestion is a design journal, a document that could be maintained in software. Another suggestion is the convenient display of constraints that a software engineer must keep in mind. Tracking constraints and issues frees up short-term memory for other details. The goal is to prevent a designer from forgetting a crucial issue while working on a partial solution.

The last suggestion calls for visual simulation tools. Graphical animation from executable specification languages should guide the production of software. Executable specifications provide a number of benefits beyond clarification of misunderstanding-
ings for developers. Simulation of specification behavior provides insight into what the software will really do. The effects of changes can be observed directly in the operation of the simulated specification. Errors in the specification and design can be caught by observing simulation of the specification before development begins. Catching defects early drastically reduces the cost of fixing the errors.
Chapter 2

HCI and Visualization

This chapter contains a survey of current work relevant to software engineering visualization systems. The survey begins with background on graphical display in general and draws from sources on general user interface design. All of these traditional issues present themselves in the design of software engineering visualization tools. Additionally, the survey covers recent work in the field of visualizing software systems. The chapter ends with a list of guidelines, features, and ideas distilled from the research discussed throughout. The rules of thumb in the list are meant to aid in the construction of software engineering visualization systems; they are tested on the design of a state-machine simulator in chapter 3.

2.1 Printed Graphics

Edward Tufte suggests that graphics should “reveal” data [73]; graphical displays should draw attention to distinctions of consequence. He presents a bullet list of qualities desirable in a data display:

- Show the data.
- Induce the viewer to think about the substance rather than about methodology, graphic design, the technology of graphic production, or something else.
• Avoid distorting what the data have to say.
• Make large data sets coherent.
• Encourage the eye to compare different pieces of data.
• Reveal the data at several levels of detail, from a broad overview to the fine structure.
• Serve a reasonably clear purpose: description, exploration, tabulation, or decoration.
• Be closely integrated with the statistical and verbal description of the data set.

Although they are meant to apply to printed material, these criteria are useful for computer displays as well. Some strong negative examples come from the ease with which computer displays are crowded, distractingly colored, or obfuscatingly sized and shaped. In [73], Tufte examines the history of printed graphics, beginning with data laid over geographical maps and progressing through train schedules, national debts, and dozens of other examples. He develops a series of guidelines that can be applied to user interface design as well as paper graphical presentation.

The concept of data-ink guides a display maker’s priorities. Ink on a page that directly and non-redundantly communicates a piece of information is data ink. Ink purely for decoration or redundant presentation of information is non-data ink. A high ratio of data ink to non-data ink indicates a good design. Quantitative calculation of the relevant areas can determine the ratio with good precision. This concept requires modification for use with user interface design. A concept of “data-phosphor” must account for painting the page background as well as data.

Tufte’s rule requires the presence of a relatively uniform backdrop for whatever images the application paints on the screen. The rule for chart design derives two rules for user interface design. First, user interfaces should make use of a plain and neutral backdrop. A word processor’s metaphor of white “paper” beneath black “ink” typed by the user exemplifies a good choice. Bad choices can be found on any number of winter holiday web sites using green text on red backgrounds – color-blind users
find them hard to see and inaccessible. Muted and consistent shades of grey work well as backgrounds [74].

Graphical elements that serve multiple purposes are encouraged. On paper, elements achieve multiple purposes by showing more than one element of data at a time. For instance, placing numbers on a line rather than boxes shows the exact value of the number as well as its relative position to other data points. Presenting both value and trend information with the same marking increases the power of the graphic.

Visual interfaces admit a greater variety of purposes. Unlike the passive environment of pen and paper, visual elements can transform in real time to reveal additional data. Tool tips provide an example of additional information a GUI can provide in response to mouse motion. Further, visually communicating data is only one of several purposes for display on a computer monitor. Other widgets provide navigation through a browsing environment and editing control. An object can display a summary of data and respond to a mouse click by switching views to an elaborated window showing more detail. Similarly, double clicking on an item on-screen often starts an editor for it. Computers excel in the area of giving items multiple purposes.

"Multiples in space and time" [75] are another powerful idea of graphical design. The same shape replicates, showing many panels of nearly the same image. The variations in shape, coloration, or decoration reveal the data. Looking at each panel provides information about a time slice, sample, or other individual measurement. Looking across the panels encourages comparison of the differences in similar structures. A row of screen shots of an application, each depicting a different stage of execution, exemplifies this design technique. Each image shows how the application looks at that point. Comparison across screen shots reveals more information: the sequence during a particular task. Computers improve on the idea of small multiples with temporal sequencing. Computers show animations that illustrate, through time, the evolution of phenomena. Without animation, the presentation of data that changes in both space and time is very difficult [74].

Computers are also well suited for working with micro/macro readings [74]. For pen and paper graphical displays, the display should give the reader an overview of a
wide data set by displaying a regular and repeating structure. This allows summary at a glance. But the details of the image should reward close attention with more information. An architectural blueprint is a good example. The larger features of the plans allow readers to see the rough shape of building areas. A glance reveals the relative sizes of rooms, hallways, and open regions. However, plans drawn to scale also yield information under careful scrutiny. Measurement of the paper representation provides details, such as the number of seats a conference room will hold or whether the distance from offices to the stairs will meet fire codes.

The principle of layering and separation [74] helps to explain why some interfaces seem busy or cluttered. Strong, heavy, graphical elements such as thick lines and boxes suffer from a problem dubbed, “1 + 1 = 3 or more.” Consider two thick parallel black lines on a white piece of paper. Although each black line is only a single item, visually they create three elements: the two black lines and a perceived white line (un-inked paper) running between them. Computer displays are particularly susceptible to this problem, as rules and boxes are drawn around buttons, fields, images, menus, check boxes, and almost every other graphical element.

2.2 Editors

Editors allow a programmer to create, browse, and alter text files. These text files are often source code files, but any documentation or other files that can be edited in text form are also subject to the editor. Editors are often enhanced to help in managing the organization of code. These enhancements involve scanning the code and relating various parts of the program to one another [32].

Automatic formatting of code is an easy feature to incorporate. There are a number of pretty printers, code formatters, and books of stylistic guidelines available on the market. Unfortunately, authorities on coding style often contradict each other. All extol better readability as their virtue, but few, if any, attempt to justify why one practice is preferable to another [53]. Varied aesthetics become a problem when a project team cannot agree on coding standards or tools to support them.
An editor can internally represent code as an abstract syntax tree using one of the incremental compilation algorithms available from language research. Within that context, the editable code presented on screen is as much automatically generated from the AST as it is independent text. Such an editor removes the problem of deciding on stylistic coding conventions. Each programmer may configure the editor to render the source in a pleasing style. Because code is parsed on the way in, and generated on the way out, no semantic meaning is lost. Comments and other extraneous details may be tracked within the context of the AST. But the source itself is formatted in any way desirable.

Concurrent editors such as MACE [52] introduce additional concerns. The user interface must provide access to the identities of all users editing a file together, but this information should not get in the way. The editor must be capable of broadcasting updates in real time, but it must also have privacy features to allow individual authors to conceal incomplete draft changes. Most importantly, the creators of MACE found that users would not use their editor if it did not support a command set, look, and feel similar to each user’s preferred editor. For example, users accustomed to emacs demanded emacs style key bindings before using the program. Consistency in user interfaces is important, as is user preference. Key bindings to commands should always be remappable. Preparing pre-set templates that mimic similar applications is desirable, where permitted by intellectual property law.

Layout consistency carries the same importance as input consistency. Many image editing applications have a tool palette. This palette is a small window with toolbar style icons to represent common operations. The window is detached from any other frame in the application and may be placed anywhere on the user’s desktop. Most other applications use only a single window, lining controls in toolbars and menus across the top. Often these toolbars are detachable into palettes, but few users do so. The single window tendency is even stronger for documents. Most applications present a single window that holds one document. Additional documents replace the view of the first in that window. Very few applications allow an arbitrary number of documents to be opened in separate window frames. A multi-window style certainly

23
provides the user more flexibility to minimize, resize, tile, iconify, or expand documents and palettes of tools as appropriate. However, enough applications have set precedent for a single-window environment that it would be jarring and disorienting to many users to work with a multi-windowed application. For ease of use, a user interface must meet the expectations of a user: the principle of least surprise. The more flexible, powerful style is not necessarily beneficial.

Editors offer more than pretty printing and concurrency. Editors also provide the best place for in-source visualization tools. At its simplest, a comment is a tool for understanding source. The comment explains the source code around it. Graphic visualization can be used in the same way. The GRASP project provides an example of a graphically enhanced editor [34]. Graphical notation in the left margin accompanies source code text. The graphical notation uses symbols familiar from flowcharts and other design notations. The set includes small diamonds next to if statements, snaking lines that indent at each nesting of flow control, dots that continue straight down to any “else” statement, and leftward arrows noting return statements. Glancing at the symbols yields a sense of the overall flow and structure of the code, without reading through it. When reading in detail, the symbols at the left of individual lines provide a sense of context.

To prevent visual clutter, GRASP permits disabling individual annotation elements. With further insight, the authors realized that often one wants to see the major flow control structures of a program without reading the intervening code. GRASP includes a facility to “fold” code [35]. Folding removed the code inside a block from display. Clicking on an icon left in place of the code restores the missing source. This feature folds up code that one does not wish to consider, emphasizing the control structures wrapped around the code at that point. In addition to removing the distracting detail of the folded blocks, the extra space brings a larger portion of the program into view on the editor screen.

Folding is only a small portion of a larger mechanism. In essence, folding removes something from immediate display and presents a token for retrieving it. In this case, code is removed from view and replaced by a token that allows the user to retrieve the
code. These screen icons could easily generalize to allow retrieval of almost any kind of data. For instance, one could link a full document describing a department of defense coding standard to the line of code strongly affected by that standard. Programmers could use data formats other than text; for example, a software engineer might record voice or video instructions to document a passage of code. That voice recording would be referenced by an icon in the code. When clicked, the message would play. The general guideline here is that unobtrusive icons can stand for much more varied and complicated data. So long as the number and types of icons are not overwhelming to the user, they are an excellent way to make information available without adding visual complexity.

Another visualization offered by GRASP is the CPG, or “Complexity Profile Graph”. For this view, GRASP evaluates every line of source code with a complexity metric. For purposes of the visualization, the metric could be arbitrary. In this case, the metric is a combination of four factors designed to measure complexity of a line’s content and context [33]. A graph charts source code line numbers on the x-axis and complexity metrics on the y-axis. Doing this produces a graph that displays complexity throughout the program. A software engineer can find spikes and plateaus in this graph indicating areas of high complexity in the source code. These areas can be targeted for simplification during code maintenance or for particularly intense study while coming up to speed on the code.

A completely different tool for visualization of code would be a call-chain simulator. Many times, programmers have cause to wonder, “Who called this procedure?” or “What procedures is this one likely to lead to being called?” Answering these questions requires tracing through the code to find the relevant references. A simple visualization technique for this problem is possible with a code editor. The editor should allow users to select procedure names and choose to open the procedure selected in a new window (preserving the old window as well). Additionally, in a side window at the edge of the screen, the editor should append the name of the procedure selected to a list. In this way, the list at the edge of the screen is identical to what might be a valid call chain through the program. A user might click on previous
entries in the chain to pop back up through the call stack and make other choices for further investigation.

2.3 Visual Languages

True visual languages have graphical representations for almost all aspects of language syntax [14], including basic flow control constructs and operations on primitive types [59]. This definition contrasts such products as Visual Basic or Visual C++, in which the visual mechanisms relate solely to GUI creation. Formalisms for these syntaxes have been created analogous to the formalisms for textual grammars. Relation grammars represent sentences with symbols and spatial relationships between those symbols [21]. Symbols have productions and relations have evaluation rules. In many cases, procedures form the atomic graphical units; they are connected by lines that indicate data flow [61]. Many visual languages use a visual model based on data flow and or flowcharts [11], using lines to connect data output from one module as input to another [7].

“What should be graphical?” is a central concern when developing a visual language. Coarse grained features of a program often display more easily as graphics. Module dependency, procedure and function dependency, and flow control all have traditional graphical views. Visual languages can borrow these notations, originally meant for documentation, and put them to work as syntax in a new language. However, some constructions in programming languages are more awkward when manipulated as graphics, such as arithmetic expressions [80].

Another concern is the lack of re-usability of visual programming editors. In general, a visual programming language requires an editor capable of manipulating the symbols used in that language's syntax. By comparison, text editors may be re-used as-is to edit any textually based language. Applications such as Palette [27] address this problem by providing environments in which one can build any set of graphical primitives needed for different visual languages.
2.4 Debuggers and Profilers

Visual debuggers present a graphical rendition of an executing program and its data. Data and code are the subject of visualization, although sets of interesting events, such as particular procedure calls or messages passed, can be rendered independently. There is little abstraction involved in the visualization. Debuggers do not present summary data in the way that profilers do; individual lines of code and values of data are presented for inspection. The user can behave actively or passively with such mechanisms as breakpoints and data changes [7].

Debugging and performance analysis tools divide into two functional modules [41]. The first component monitors; it is responsible for collecting data from the program under study. The second component analyzes or processes data for display to the user. For large software applications, a substantial amount of data is collected during debugging or profiling. Mapping that information into an understandable display is challenging.

2.4.1 Event Graphs

Event graphs aid in visualizing communication between nodes in a distributed system [41]. The graph consists of horizontal lines, reminiscent of a musical scale. The horizontal axis is a time scale, with time advancing from left to right. At the left margin, a label annotates each horizontal line, indicating the process associated with the line. An event on the graph is an “action without duration”, an instant in which the state of a process changes. Event graphs used to debug large parallel systems focus on message passing events: send and receive statements. These events are represented by circles on the horizontal lines. A line with an arrow points from the “send” event on the source process line to the “receive” event on the destination process line.

Event graphs are suggested for diagnosing faulty behavior in message passing and for finding performance bottlenecks. A software engineer may scan across a horizontal line in search of an isolated event: an event with no arrow to or from it. An isolated send event may indicate that a sending process is filling a buffer that will never be
emptied. An isolated receive, particularly a blocking receive, would indicate that a
process is blocking, waiting to receive a message that never arrives. Another error
may be diagnosed if a joined pair of sender/reciever events disagree on the number
of bytes that were in the message. This might indicate an error in the protocol or
message handling.

Further processing on event graphs provides an example of a formal analysis tech-
nique simplified by software automation. Event graphs can be used to detect race
conditions. An analytic method identifies a candidate set of events that could have
their ordering altered in successive runs of the program. These events are highlighted
on the screen. A programmer may investigate what alternate patterns would look like
by dragging the events along their horizontal lines. Repositioning the events gives the
programmer direct control to change the visualization to other possible traces. Often,
the consequences of changing the ordering of messages are apparent when examining
the event graph.

The basic event graph takes on a number of additional roles as features are added
[41]. Periodically, a small symbol appears on the time lines indicating that the applica-
tion has been checkpointed. The application may be started at any such checkpoint,
mid-stream. Also, when a process blocks on an event, a rectangular bar appears,
starting at the time of the block. The bar terminates at the circle that indicates an
event. By examining these bars, a programmer can survey the performance of the
application in terms of the time it spends blocking. Finally, each event circle links to
the line in the source code that generated the event.

Event graphs must control a problem of scale. The display becomes visually
crowded as the number of processes, nodes, and messages increases. Grouping similar
things reduces visual complexity while preserving meaning. For example, many multi-
process programs employ a few master processes that create, coordinate, and destroy
worker processes. There may be thousands of worker processes handling tasks given
out by the master processes, but only the master processes are of significant interest.
Collapsing the worker threads onto one aggregated line solves the problem. The loss
of distinction between the worker processes is unimportant while focusing on the
master processes.

A similar technique compresses data along the horizontal axis – collapsing a common pattern in time. Such patterns might be a broadcast message from one process to all others or a single back and forth exchange between two processes. These interactions can be replaced by a single vertical rule line labelled with an identifier for that pattern. A double benefit is afforded by this practice. Near misses to a pattern of message exchanges often indicate subtle bugs. If the developer sees a series of messages that should have been abstracted into a pattern, looking for the difference that caused the pattern match to fail may aid in finding a bug.

Filtering also reduces the complexity of the display. Hiding some processes, their events, or the communication arrows between them helps to unclutter the display when viewing the event graphs. This is similar to the pattern matching mechanism, but pattern matching leaves a token to indicate that the pattern is present. Filtering just removes items from display.

Irrelevant information must be collapsed into representative categories. The visual display of these representative categories must draw less attention from other data than the expanded members. Semantic meaning is preserved wherever possible. It is always preferable to collapse data in such a way that it is grouped with like data, allowing display of the aggregate or a summary. For example, collapsing all the slave processes together means that events and communication can still be displayed for the group of slave processes. Similarly, when identifying patterns of communication to collapse along the time axis (such as broadcast communication), the visualizer is replacing a busy display with a quiet token representing the same semantic meaning.

Visually, the display is made clearer by aggregation only if the aggregated groups draw less attention than the remainder of the screen. For example, collapsing a few hundred slave processes into one grouping may not be entirely successful. If the density of communication from that group to the rest of the horizontal process lines is overwhelming, the programmers attention will be drawn from items of interest into the thick nest of communication links. In that case, it might be necessary to hide the communication links to and from the slave process group. This conflicts
with the desire to preserve information where possible. A better option would be to display slave communications in a muted color and shade: something similar to the background of the display, but perceptibly different. With this solution, the content of the information is preserved, but it does not distract from the intended focus of attention.

2.5 Design Aids

A number of tools have been developed for visualization of program design. These programs vary from flowchart editors to applications for laying out module dependency relationships. Recently, the introduction of the Unified Modeling Language [24] [10] [64] made support of UML a trendy focus for graphical design tools [57].

The Argo project introduced a novel way for software engineers to interact with their designs [62]. Argo is based on a few theories of the cognitive processes of designers.

**Reflection-in-action** states that designers are best able to evaluate their designs while making their decisions, not afterward. If only some information is available at that time, then the information that is available will receive a bias of attention.

**Opportunistic design** suggests that designers work by their perceptions of priority or ease of task. They do not follow a predictable process when addressing design issues.

**Comprehension and problem solving** proposes that designers use many mental models of a system. Each model is tailored to the design issues a software engineer needs to explore at that moment.

Argo contains an infrastructure that hosts a number of software architecture critics. These critics are threads that run in the background, while the designer works. When the critic’s heuristic indicates that the design violates its principles, the critic...
will highlight the problem area. Because of the principle of reflection-in-action, critics are meant to operate on incomplete designs. They note troubles while the designer is working, rather than waiting for a finished product. Because critics are small, easy to write threads of code, hosts of critics can be written to pore over all kinds of domain specific issues – modularity, visibility, stylistic awkwardness, and maintainability are just a few. Each critic can be enabled or disabled to prevent an overwhelming amount of irrelevant feedback.

Each critic focuses on narrow issues, so its heuristic will likely incorrectly identify some problems. Because each critic is light-weight, there are likely to be many in parallel operation. The design feedback from the critics must not interfere with the designer’s efforts. Argo’s user interface scales well up to a large number of critics. Messages from critics stack up in a “to do list” format. When selected, a critic’s message highlights the portion of the design that triggered the critic. The designer can resolve the issue either by modifying the design or overruling the critic.

At the other end of the life cycle, Arch helps to reverse engineer existing designs rather than producing new ones [67]. The tool targets maintainers of legacy systems. The central abstractions in Arch are the subsystem tree and the constraint graph. A subsystem tree partitions the system into subsystems, which are in turn composed of subsystems. A constraint graph is a multi-colored, directed graph. The nodes in the constraint graph are nodes in the subsystem tree; edges represent relationships between the nodes. Coloration determines the type of relationship. Types of relationships might be “includes,” “exports,” etc. Arch scans source code and automatically constructs a visual graph of the subsystems. To cut the visual density of the graph, a user can abstract the details of sub-components into a single link from one larger subsystem to another. The Arch tool further allows architects to introduce constraints on nodes in the graph. By doing this, system modification is controlled to avoid undesired coupling of separate modules. An architect might determine that one module should have no reason to import another and explicitly prohibit it. If that prohibition is violated, an error is generated during the build process. In the case of such errors, Arch permits comparison of the present state of the system to
2.6 Algorithm Animation

Much experimentation in visualization techniques occurs in software engineering education [81]. Educational institutions have tried a number of sophisticated systems for algorithm animation. The goal of these systems is to help students understand basic data structures and algorithms by allowing them to step forward and backward through the program’s actions. This is done with a visualizer rather than a debugger because a visualizer can be built to aid in understanding. Debuggers operate at too low a level of abstraction [7].

Educational use of algorithm animation has shown that a high degree of user control is necessary [84]. These visualizations depend on the user watching a sequence of displayed actions. Context is important. If a user loses track of his place in the animation, he or she may not gain any insight from the remainder of the animation. To address this problem, time-animated graphics must provide a means to rewind, pause, slow-down, and speed-up the pace of the animation [7].

Many applications for playing video files provide a slider bar at the bottom of the viewing window. Dragging the token on the bar repositions the viewer in the file. Streaming media handling depends on the type of stream; some data formats do not allow repositioning. And one cannot slide past the end of the data that has been downloaded. Analogously, a viewer may animate an algorithm that is executing in real time. Particularly with a distributed algorithm, it may not be possible to fast forward beyond the currently reached point in the computation.

Algorithm visualization often involves two mechanisms [7]. First is the event trace. The event trace is a depiction of what happened during an algorithm run. The events of interest vary from algorithm to algorithm. The event trace is a time series recording of interesting actions, such as interprocess messages, network communications, or file accesses. The second mechanism, the data trace, is a variant of the event trace, and it records changes in the state of data. It is distinguished from the event trace in
that tracing state changes rather than point events requires that more information be collected.

2.7 Graphical Elements and Representation

Large amounts of data can be more easily navigated with graphical display and visualization tools. The complication of these displays is primarily in the size of the data sets. The trick to displaying these data is to aggregate background data in order to highlight relevant data. Each datum is a relatively simple measured quantity. For example, event graphs simplify understanding of complicated messaging systems. Individually, though, each event is simple to understand. Using graphical elements to represent the concepts of software engineering is not always so easy a task.

Visualization techniques are more difficult to apply when each datum being represented is more complicated. For example, visual programming languages are much more difficult to represent. Graphical notations must capture scoping, procedure invocation, control flow, data flow, and type information. Textual programming languages borrow from natural linguistics. This background already provides means of determining when one symbol modifies another. Indentation provides a sense of containment, as do parentheses. Reading left to right and down the page provides a convention of sequence. Programming languages simplify grammar to something regular and unambiguous. Graphics offer a more rich set of primitives. Spatial relation offers upwards, downwards, left-of, right-of, inside, outside. Variations of shape and coloration are almost limitless. But the semantics for combination have no precedent. How is one element positioned to indicate modifying, containing, shadowing, or following another?

This lack of precedent can also be seen in the context of declarative programming for GUI systems [51]. Direct manipulation is concrete; one changes a widget on screen and the code to produce that widget changes. Programming, in general, involves abstraction. Most visual programming techniques fail in that one cannot abstract from the concrete prototype of a user interface. There is no precedent for how
visual programming symbols might be more or less abstract. This problem has been attacked with a set of Prolog rules. The rules map relationships between programmatic objects. Firing these rules, in turn, leads to rules that map a programmatic relationship to a spatial relationship depicted on screen. For instance, a supervisor(X, Y) relationship may imply a vertical(X, Y) relationship, where a supervisor is graphically depicted above a supervised employee. Given this example, the system interprets one icon above another to represent a supervisor and supervised employee in each new case. The application of an old pattern to new data is, in essence, abstraction. This technique is called programming by example. One example provides the template interpretation for future elements in the same relationship. There is no canonical mapping of semantics to graphical elements, but structures as complicated as recursive data structures can be represented this way.

Another approach claims that the problem of abstraction is best solved by making it easy to add new visual primitive elements, removing the need to abstract at the visual level [9]. Most users are uncomfortable extending the provided library of components for visual languages. Thus, the complaint is not that relationships between components are difficult to map, but that creating components is complicated enough to intimidate users. The solution is to create a small set of reusable components that are easily combined into larger solutions. An editor for these basic components can also serve as the visual programming environment for end user applications.

Others have tried to attack the complexity of the data model itself. In general, good design is capable of decoupling the model of data from its display. In order to make a variety of analysis support tools useful, however, it would be desirable for those tools to share a representation [87] and a framework of execution.

2.8 Visual Feedback in Direct Manipulation

Direct manipulation programming covers the range of macro builders, spreadsheet programs, visual equation editors, and some visual programming languages [5]. In these environments, the system reacts instantly to changes made by a user, rather
than after a recompilation of code. The canonical, though simple, example is the recalculation of a spreadsheet when a new number is entered into one of the cells. The property of “liveness,” describes the immediacy with which a system reacts to users changes [82]:

**Level 1**: No semantic feedback is provided to the user.

**Level 2**: The user can obtain semantic feedback, but it is not provided automatically.

**Level 3**: Incremental semantic feedback is automatically provided whenever the user performs an incremental program edit, and all affected on-screen values are automatically redisplayed. For example, the automatic recalculation feature of spreadsheets supports level 3 liveness.

**Level 4**: At level 4, the system responds to program edits as in level 3, and to other events as well such as system clock ticks and incoming data from sensors.

One of the benefits forecasted for visual feedback in direct-manipulation is decreased debugging time. If programmers see the results of their changes immediately, they should more swiftly track down bugs and fix them. The correctness of that forecast has been challenged [82]. Behavior during debugging changes significantly with the live system. Programmers with a live system make smaller changes more often. The end results, however, do not show a significant improvement in the performance of debugging. Some bugs are solved faster with a live system, some slower. Performance varies significantly based on the type of bug and the individual doing the debugging [16].

Type of bug is the most cited factor in whether liveness improves or hinders debugging speed. The details of the visual system used for the test may have contributed to some bugs being easier to solve than others. The two programs to be debugged were a mathematical locking program and a graphical display program. Liveness aided in the
mathematical lock program more than the graphical display program. The language used for the study was Forms/3. It may be that Forms/3 yields more information in its graphical syntax relevant to the correction of mathematical formulae than it does about control flow, which dominated the graphical display program.

There is not yet a definitive answer on appropriate uses of interface liveness. The point is well taken for user interface design that one task, such as debugging, may require a number of different views on the same data. Until the appropriateness of each view for each purpose is determined, it should be easy to switch between views as needed.

2.9 Data Dictionaries

Global [17] is an example from the camp claiming that emacs can be made a user interface for anything. A data dictionary is similar to, though simpler than, a code slicing tool. It does not consider implications from one statement that might affect another – it cannot tell you all the lines that may affect, even distantly, a variable’s value. However, it can point out all occurrences of a variable or procedure in a large base of source code. This indexing is a particularly simple form of formal static analysis. The output is accessed by command line or through an emacs interface. Users type a portion of the name to be retrieved. The completed identifier and all the line numbers on which it appears pop up in a window. These line numbers (with associated file names) allow one to find all references to an identifier in a large project. Keyboard commands, in emacs, allow navigation around the list, directly taking the user to the lines of code referenced.

One could imagine enhancements to this interface, however. In large projects, filters for the displayed lines might be helpful. Some way to account for shadowed variable names would be preferable. A more powerful analytic engine could trace variables through possible procedure invocations to the argument names that would name them. Correspondingly, the user interface would have to allow display or suppression of these additional possibilities. Color coding might indicate density of an
identifier’s appearance in an area or special uses such as initialization or destruction of variables, recursive calls to procedures, etc. The possibilities for expanding the variety of information carried by the display are numerous. The one advantage the current interface has is simplicity. While it does not provide all the information it could, what it does provide is presented in a clear and unobfuscated manner. A list of lines of code is easily interpreted in the context of Global’s operation.

2.10 Extensions for Collaborative Systems

Sharing visualization systems among multiple participants requires several extensions to normal visualization systems [83]. Sharing data sets requires synchronization. Collaboration in large scale programming projects traditionally revolves around a repository of files. That repository has the machinery to incorporate differing versions of files via a three-way merge. Large projects with a variety of configurations, target platforms, and versions may have a sufficiently complicated configuration space to merit visualization tools for that purpose alone [79]. Changes made to such a project are preserved by merging with the baseline files.

This as an asynchronous system – collaboration in realtime requires fine-grained updates of content, making them synchronous systems [60]. Realtime collaboration requires merging and synchronization to be pushed as far forward as the editor itself. Different mechanisms for updating shared content have led from WYSIWIG (What You See Is What You Get) to acronyms like WYSIWIMS (What You See Is What I May See) [52] and WYSIWITYS (What You See Is What I Think You See) [70].

Participants in collaborative work also require data exchange. In addition to the primary user’s system of visualization, data from collaborators must be unobtrusively incorporated into the display. This may be as simple as setting up a data feed from the remote to the local location. More likely, it will mean structuring a way to read notes and datagrams or even view snippets of multimedia attached by collaborators.

In collaborative work on rapidly evolving documents, users must share control of the application. Shared application control has applications in user training as well.
There are several models for this. A guide may lead all other participants by taking control of the application alone while others observe. Alternatively, a relative anarchy could emerge if all participants act in an uncoordinated, unmediated fashion.

The MACE program solves these problems flexibly [52]. MACE allows concurrent editing of text files. Unlike a repository system, all users may edit the same file together in real time. The granularity of editing is arbitrary. In order to edit a segment of the document, the user places bounds on the region, locking that region. During the period of the lock, only that user may edit the locked portion of the file. Since lock granularity is a single character or space between two characters, a file may be shared among as many users as needed. Users may also permit realtime updates from their section to others. This permits peer review as the document is being written.

2.11 Guidelines and Methods

The following guidelines and practices are distilled from the above examples. These rules of thumb provide practical guidance in designing software engineering visualization tools. These heuristics overlap, and many categorization choices could be debated. However, the principles themselves are more important than whether one considers them to be more closely related to an architecture design concern or usability.

2.11.1 Process

- Prototype the displays. Prototyping the user interface with some user manipulation of widgets is even better [48].

- The design of visualization systems is well guided by models of cognition.
2.11.2 Architecture

- Where possible, build an editor that operates on the syntax and semantics of the document, rather than a simple text editor. The data view can be customized by different users.

- Bear in mind that text is exportable and universally editable. Graphical formats are often backed by a binary representation. Consider defining a text export format, such as with XML.

- Decouple the model from the display system. Share the model among multiple analysis components.

- Self-contained analysis modules, such as Argo critics, allow incremental increase in the power of a visualization system. Such extensibility is good design.

- When possible, design for a collaborative multi-user system up front. At the very least, if a program is to be used in software engineering, always provide a scriptable means of performing merges between differing versions of the data file. Compatibility with leading source repositories is also desirable.

2.11.3 Usability

- Try to keep the user interface as self-consistent as possible [68]. Navigation tools, in particular, should always be consistently located and used throughout the application.

- Carefully consider whether novelty in the user interface is needed. The learning curve for new users will flatten with similarity to other interfaces. For visualization displays, borrow from familiar paper plots such as map displays and time series plots.

- Applying an analogy to the design of a user interface, such as the “desktop metaphor” may make concepts easier for users to understand [46].
• Screen layout should direct the user's attention purposefully towards important material [15].

• Texture and color are taken in during pre-attentive processing by the visual cortex. These are often used in line charts of one kind or another. Shape, area, and containment are perceived by serial scanning of the eyes [44]. Thus, prefer line charts and histograms to pie charts and Venn diagrams when possible.

• Color cannot be used to highlight or contrast if much of the screen real estate is covered in bright colors. Use color sparingly on a neutral, muted background [15].

• Avoid heavy lines and rules bordering display widgets. Use thin lines in muted shades of the background color; they are easier on the eyes and do not draw attention from important content.

• Combine words and graphics [15]. Graphical icons alone do not convey meaning very well. When used with words, the icon provides a mnemonic shortcut, but the word ensures clarity. Visual languages often have particular trouble with this concept.

• Support alternative key binding. Where permitted by intellectual property law, provide template key bindings for similar applications.

• Always provide an annotation mechanism. Users are likely to have a pen and paper near their terminal, but attaching notes directly to the data that generated the thought is preferable.

• Context in animations is very important. Animation systems should include controls to fast forward, slow down, pause, and rewind the display.

2.11.4 Visualization

• Follow the Visual Information Seeking Mantra: “Overview first, zoom and filter, then details-on-demand,” [69]. Computer displays should present macro-level
summarization of data with options for closer inspection of details.

- Maximize “data-phosphor”. Include necessary controls and displays, but make the user interface as clean and unadorned as possible.

- Strive for multi-purpose graphical elements. Where possible, graphics should illustrate both a value and a trend. It is less advisable to couple control and navigation elements to information display elements. Doing so may hide the controls from users.

- Animation and small-multiple images (frames of an animation laid out in sequence) are desirable because they encourage comparison of values.

- Graphical explanations of source code can be particularly effective if combined with the display of the source itself. When providing such adornments, allow users to disable them piecewise.

- Complexity is a particularly thorny software engineering problem. Most visualization systems could incorporate some sort of complexity metric for data under visualization. Providing users a means of changing the calculation algorithm is desirable.

- Time series graphs are a powerful tool for watching the behavior of programs. Archetypes common to these graphs are parallel lines representing different objects of interest (such as processes on a node or nodes in a network), mark on those lines (for events of interest), and arrows (connecting related events). Grouping and filtering is useful for scaling these graphs to large systems.

- Dependency diagrams of various granularities provide insight into system structure.

- Providing both description and prescription modes of visualization allows software engineers to see “what is” compared to “what should be”.

41
• Beware of immediate feedback and direct manipulation. They encourage shorter
cycles from thought to action. Correspondingly, they often feel as though they’re
closer or easier. In reality, the benefits are dependent on the user and the
problem.

Memory and Navigation in Visualization

• Keep short term memory load to a minimum [68].

• Group related information, lest the user have to switch mental contexts to pull
information from the application [15].

• From [44]: tables don’t require mapping graphical symbols to semantic infor-
mation. Thus, use a table when the important information can be read straight
from the table. For identification or comparison of trends, however, a table re-
quires multiple lookups of values. All these values must be stored in short term
memory. Thus, prefer graphs over tables for trend analysis and interpolation.

• Take advantage of hypertext-style linking. But bear in mind that users must
switch mental contexts when following such a link. If a user is likely to use a
particular kind of link often, create a separate display that collects the linked
pieces of information in one place. For example, consider code slicing as a more
powerful addition to hyperlinks between procedures.
Chapter 3

State Machine Simulation

State machines are often used as specifications for embedded systems. State machines are simple, and they are based on well-founded mathematical properties. If a system is specified as a state machine, extensive formal analysis can guarantee properties of the software. Less formally, simulation is a useful technique for building insight into the operation of the system. Since state machines are executable, simulation is straightforward. In fact, generating code straight from the state machine would be simple; the only obstacle is that black-box code generated from a high-level specification or design often does not meet performance goals of the system. Digital designers have this problem as well [76].

Software engineers observing a state machine in simulation can correct flawed behavior in the state machine specification before it is translated into a software system. Catching mistakes early, such as during specification, is much less costly than during implementation [63] [43]. In the case of safety critical systems, removing as many errors as possible early on is very important in terms of lives and property saved as well as development costs.

Overstreet and Nance [54] present a clear series of definitions for terminology surrounding simulation. These definitions are paraphrased below:

A system is part of the world which we choose to regard as a whole, separated from the rest of the world for some period of consideration,
a whole which we choose to consider as containing a collection of components. This system may be real or imagined. The system may have inputs and outputs.

**Inputs** are items to which the system is somehow sensitive but which are at least partly beyond its control.

**Outputs** are items which the system provides to its surroundings; they may or may not be used in the system itself.

**An environment** may be undefined except to identify the inputs of the environment to the system and the outputs of the system to the environment. The environment views the system as a “black box,” as long as the inputs and outputs are adequately specified.

**A model** is an abstraction of a system intended to replicate some properties of the system. The level of detail and the type of abstraction depend on the properties the model is intended to replicate.

**The model objective** is the set of properties the model is intended to replicate.

**A model specification** defines what a model does.

**A model implementation** defines how the behavior is to be achieved.

There are a variety of types of models. State machines can be classified as “declarative models” [23], along with other finite state automata, Markov models, event graphs, and temporal logic models. The dynamic behavior of these models consists of state or event transitions. These models emphasize the structure of states or events, removing corresponding emphasis from the details of the transitions. However, in simulating the behavior of a state machine, it is the state transitions that require the most attention. The first observation that something is wrong may be that the system is in a wrong or unexpected state, but the path of transitions is the forensic trail that leads to identification of the error. Only after discovering the error can the engineer correct the state machine’s behavior.
3.1 Simulation Representation

The simulator uses a simple state machine representation. Ideas behind the simulator data representation are derived from joint work on a different intermediate form [3]. As the simulator itself may be implemented based on a standard tool framework [4], it is designed with the language of the framework in mind: Java.

3.1.1 Model

The Model class contains a single, simulatable state machine specification. It contains the following variables:

**identifier** The identifier is a variable of type Identifier. The identifier is a label for the model.

**attributes** The attributes variable is of type Group. The Group is a collection that contains variables of type Attribute. An Attribute is an identifier/value pair. These pairs are used to extend the capabilities of the simulator without drastic rewrites.

**inputs** The inputs variable is of type Group. The Group is a collection that contains variables of type Signal. Input signals come from the environment of the model.

**outputs** The outputs variable is of type Group. The Group is a collection that contains variables of type Signal. Output signals are returned to the environment of the model.

**states** The states variable is of type Group. The Group is a collection that contains variables of type Signal. State variables represent a network of states connected by transitions.

**subModels** The subModels variable is of type Group. The Group is a collection that contains variables of type Model. Sub-models are not necessary for correct behavior of the simulator. They allow state machine specifications to be modularized, developed separately, and integrated. This cycle allows for re-use of
old specifications. Attributes in the enclosing model and the sub-model connect inputs and outputs of the two models. Any inputs and outputs not connected between the two models are promoted to the inputs and outputs of the enclosing model. This is similar to the way in which free variables are promoted into an enclosing scope in some programming languages.

3.1.2 Signal

The Signal class defines the structure and behavior of signals to, from, and within the state machine model. Signals are the entities that fill the input, output, and state groups above. The decision to use signals to represent states is deliberate, borrowed from the fields of signal processing and control theory. In essence, a signal carries a value. Input and output signals carry values of varied types. State variable signals must always be enumerated types. That enumeration is the range of possible states that a state variable can be in.

For example, an input signal might be a temperature measurement. It could take on any integer value. If a Kelvin temperature, values below zero might not be meaningful. An example output indicator might be a status indicator, which would be a boolean, true or false. A state signal, however, must have a value from an enumeration. An example would be the state values for a traffic light: LightRed, LightYellow, LightGreen, BlinkingYellow.

The Signal class contains the following member variables:

**identifier** The identifier variable is of type Identifier. The identifier names the signal.

**attributes** The attributes variable is of type Group. The group contains variables of type Attribute. An attribute is an identifier/value pair. These pairs are used to extend the capabilities of the simulator without drastic rewrites. They may eventually describe timeout behavior, for example. There is one attribute currently in common use; a signal type attribute describes whether the signal is an input, output, or state signal.
**type** The type variable is of type `Type`. The type variable determines what type the value takes on. Possible types are boolean variables, integers, floating point numbers, and enumerated types.

**value** The value variable is of type `Object`, the most generic type in the Java language. The value is the current value that the signal takes on.

**specifiers** The specifiers variable is of type `Group`. The group contains variables of type `Specifier`. Specifiers determine the value that the value variable takes on. They are called specifiers because they specify the value of the value variable. Each specifier contains a group of conditions. If a Condition evaluates to true, then the value variable takes on the value associated with the Specifier.

For state variables, a group of specifiers defines the transitions the state variable can take among different states. Only one Specifier at a time will have a true condition, indicating the necessity of adopting the associated value. For state variables, each Specifier’s value is a single value out of an enumerated data type, so each value can indicate one state out of a state transition network. Thus, when one Specifier’s Condition is true, it indicates the necessity of transitioning to that state if not already in it. A model is not well-formed if more than one Specifier’s Condition may be true at a time; this will generate an error.

For input signals, the group of specifiers may contain only a single entry, with “true” as its only boolean expression. The “value” portion of the specifier may be a link to a source of data, such as a file or network connection. This allows dynamic generation of input data.

For output signals, the attributes of the signal describe the sink (often a file or network connection) for the data. The condition may contain only a single entry, with “true” as its only boolean expression. The “value” portion of the specifier may be a formula to compute the current output value. This allows dynamic generation of output data.
3.1.3 Specifier

A signal, be it input, output, or state, takes on a value based on a group of specifiers. The variables contained in the Specifier class are listed below:

**name** The name variable is of type Identifier. The name variable provides a name for the specifier.

**value** The value variable is of type Object, the most generic type in the Java type system. The value variable holds the value that will be taken on if the condition block is true.

**conditions** The conditions variable is of type Group. The group contains variables of type Condition. Each condition contains a group of boolean conditions. If every boolean expression in the group evaluates to true, the group of conditions as a whole is evaluated to be true. If the group of conditions evaluates to true, then the specifier will cause the signal to take on its value.

3.1.4 Condition

A condition is, in essence, an AND list of boolean expressions. When a group of Conditions is used in a Specifier, it forms an AND/OR block of logic.

**booleanExpressions** The booleanExpression variable is of type Group. The group contains variables of type BooleanExpression. Each BooleanExpression evaluates to true or false. BooleanExpressions may depend on the values of other signals in the model. BooleanExpressions may also depend on constants and literal values.

3.1.5 Miscellany

A handful of other classes augments this list. For example, Identifiers decompose into Java’s string primitive type. Attributes decompose into a few Java strings. There are, of course, a few nagging details in the type system of the simulator. Input signals may
come in as boolean signals, making them directly usable in Conditions. Alternately, an input may be an integer, which would require a comparison to some other integer to generate the boolean value that fits a BooleanExpression. However, type systems as simple as the simulator’s are well understood.

3.2 Simulation Operation

The background operation of the simulator is a simple loop. A system heartbeat signal pulses at a user-defined granularity. After each pulse, the system completes a round of processing. It polls input sources and updates the state machine. A special pulse frequency of “as often as possible” may be used if processor consumption is not an issue. In all cases, the pulse frequency must be sufficient to catch incoming data from any network sources that have been hooked up to input signals. The pulse frequency must also be high enough for any minimum data rate expected on an output signal.

Just after the heartbeat pulse and the poll for collection of input, the system updates the values of the state machine. This is a non-trivial problem because of timing issues between different state variables. Suppose that state variable \( A \) appears in the list of states before state variable \( B \), despite the fact that \( A \) depends on \( B \)’s value. Suppose further that state variable \( B \) depends on input \( C \), which changes value in such a way that \( B \) should change state. \( A \) will be updated first, and since \( B \) has not changed, its value will remain the same. \( B \), on the other hand, will see that \( C \) is changed and change states. \( A \) is now in an inconsistent and incorrect state until the next pulse. In a digital system, this is a property called propagation delay; it takes a propagation delay for a system to stabilize after input changes.

In an implementation derived from the specification, the behavior of the system would be dependent on the details of the implementation. Part of the specification might mandate that response time meet certain speed objectives, but otherwise the implementation is free to evaluate in any order. During simulation, the best alternative is to set the frequency of the heartbeat pulse to be sufficiently fast such that
a second pulse happens quickly enough to change state A to its proper value with no noticeable delay. So long as the heartbeat frequency is sufficiently faster than the rate at which inputs arrive, the system will have time to settle its behavior.

Without testing, it is impossible to determine whether the performance of this method of simulation will be adequate. Clearly, the time required to process each heartbeat pulse will increase linearly in the number of states and inputs. With current computational power, this may be a non-issue.

There is a ready avenue for improvement if optimization is necessary, however. Pre-processing the model could produce a data structure that maps each signal to all signals that directly depend on it. For speed, consider this mapping to be implemented as a hash table. A lookup on one signal will return a set of all signals that depend on it to determine their value. We also create a list, called the dirty list, that has a list of all the signals that need updating in the coming processing round. The simulator’s algorithm, which executes every heartbeat pulse, would now work as follows:

1. Check input sources associated with input signals. If new input is available, the input signal is appended to the dirty list A.

2. Create a new dirty list, B.

3. For each Signal S in dirty list A:

   (a) Compute a new value for S. For input signals, this means reading in the new value. For state and output signals, this means performing whatever computation the Specifiers require to compute the new value.

   (b) Lookup S in the dependency hash table. This will return a set of all Signals dependent on S.

   (c) Add all Signals in the dependent set to dirty list B.

4. Replace dirty list A with dirty list B.

The double system of dirty lists may, at first, seem a bit odd. Why put off till the next processing round dependent signals that could be processed right away? If two
signals are interdependent on each other, they will place each other in the dirty list over and over. If the simulator tries to process the dirty list all the way to the end, the computation will enter an infinite loop as two states keep appending each other to the list. By deferring processing till the next round, this problem is avoided.

Performance is improved because only potentially changing states are asked to re-calculate their values. Untouched states are ignored. If yet more efficiency is required, the static analyzer could be improved to prohibit interdependent states, allowing the use of only one dirty list. However, the restriction against cycles in the dependency graph limits the variety of state machines that can be simulated. This loss of generality reduces the effectiveness of the simulator as a tool.

Another approach would be to add signals to an already-processed list after their first round of processing, prohibiting them from being added to the current dirty list more than once. Ultimately, these approaches are more complicated, and thus error prone; they are not worth the effort unless using a higher frequency clock heartbeat is unacceptable or impossible.

### 3.3 Methods of Visualization

The state machine simulator is meant to provide insight into the structural and dynamic behavior of a state machine.

#### 3.3.1 Analogy

The overriding visual analogy for the program is that of a desktop. Java’s Swing user interface toolkit provides all the necessary primitive graphical widgets to handle a traditional desktop interface. The interface is familiar to users of countless Windows, Macintosh, and X-Windows programs.

The state machine itself is not presented with any particular metaphor or analogy. Most educational curricula for software engineering include courses on automata, including state machines. Digital hardware engineers are also well familiar with state machines. Other domain experts brought in for review of the specification need only
grasp the simplest aspects of state machines – that each state variable can take on any of its possible states and transitions between them are based on other signals. Understanding of the theoretical properties of state machines should not be necessary for use of the tool.

3.3.2 State Machine Display

The state machine model is displayed in a large central window. The central window has a neutral gray background to allow for highlighting important elements. The simulator makes a small analogy to block diagrams by placing all information about state signals inside a square box. Input and output signals are smaller boxes outside the state machine box. The boxes for input and output signals contain the names of their respective signals. Arrows point from the input signal boxes to the state machine box. Arrows point from the state machine box to output boxes. Optionally, next to the arrows, the simulator may list the signal's last value. Also, optionally, the simulator may list the time since the value was last updated.

Each state variable name is displayed inside the system box. There are several boxes beneath each state variable, containing the possible values of state variables. These boxes are created for state signals because they take on a limited number of enumerated values – input and output signals may take on any number of values based on their types, so their values are just optionally printed, as described above. When a state variable is in a particular state, that state value is highlighted; during operation it is possible to watch the progression of the simulator from state to state. The graphical layout manager underneath the display ensures that state variables are spaced evenly around the state machine box; this is important to prevent confusion about which state values belong to which state variables.

If the state machine is too large to fit on the screen, the viewer can zoom and scroll. If this is not sufficient, a user may select state, input, and output variables and hide them from the display. These hidden variables will still be used in the simulation, but they will not be displayed. This will allow the user to focus on details of interest. As a particularly advanced feature, it would also be desirable to pick a single state variable
and tell the simulator to display a slice based on that variable. The simulator would run a code slicing (or in this case, state slicing) procedure over the state machine and display only those signals that have the ability to alter (or be altered, user's option) by the selected signal. This slicing feature would be particularly useful in debugging a state machine specification, removing irrelevant signals from consideration.

3.3.3 Details on Demand

The basic display is sufficient to observe the behavior of the simulation in action. This will provide a global perspective of the sanity of the machine. If the system gets stuck in a state or never reaches one that it should, this will be apparent. However, as a diagnostic tool, we expect more from the simulator. Right clicking on a signal will produce a pop-up menu; this is the same pop-up menu that has the option to hide the signal or slice based on it.

One of the options on the menu will produce a detailed view of the signal. The detailed view will pop up in a separate window that can be moved, iconified, tiled, and resized independently of the main simulation window. This signal detail window will contain a list of the specifiers that control the signal. Next to the value that specifier makes the signal take will be the set of conditions that make that specifier true. (This set of conditions is arranged as an AND-OR block, similar to the AND-OR blocks of the SpecTRM-RL language.) Boolean expressions and AND-OR blocks are always rendered in a tabular fashion, following the guideline for data whose immediate value is of more interest than a comparison for trends.

While the simulation executes, the specifier that is currently determining the value of the signal will be highlighted. Further, the true conditional that makes the specifier true will also be highlighted.

This feature, based on the guideline of details on demand, allows the user to see the boolean expressions and conditional that are responsible for a signal’s value. Knowing why a signal has a certain value is useful for debugging purposes.
3.3.4 Texture, Color, and Shape

As noted above, shape is used only to offset the boundaries of different signals and to separate what is within the state machine specification from input and output signals. Use of shape is permissible here because the screen layout can be studied at leisure before the simulation. Events within the simulation are conveyed with highlighting and coloration. These changes are more likely to draw on the pre-attentive visual processing of the user, giving the user a better chance of keeping up with simulation activity.

Users may right click on arrows to and from output and input signals respectively. A pop-up menu will offer the option to add text next to the arrow’s line stating the last value the signal took. This text tag may also include the time since the last update of the value. As a last option, the user may choose to have the arrow highlight in a different color in a round where it is communicating with the state machine.

3.3.5 Timing and Navigation

Simulators are often run entirely in a realtime mode, where participants must be present throughout to view the results. Alternately, viewers can be run entirely as post-processors on simulation data. The state machine simulator is a hybrid. The simulation runs in real time, but the viewer may operate on a current simulation or a log of previous results.

**Simulator Control**

The state machine will execute in real time. During the processing of the pulse, each change of state is recorded, along with the round number, in a log. That log may be played back for the user by the visualizer. The log is written such that each entry gives the state value before and after the change. While replaying the log, the visualizer does not actually perform the calculations of the specifiers again, it merely reads the state changes in turn from the log. This allows the user to fast forward, rewind, slow down, pause, and single-step his or her view of the simulation at will.
Changes in speed may be particularly useful during critical state changes. Several state variables may change values on or near the same round. The software engineer may want to tease apart the transitions, in sequence, to see the exact ordering and timing of the changes. If the user pauses or slows the simulator, the simulator engine will continue to process the simulation at full speed. The visualizer will simply switch to working off of the logged events. If the user ever catches back up with the simulator, he or she will watch events in realtime again.

Finally, it should be noted that the simulator log is sufficient to initialize the state of the simulator to any point in the simulation. It is therefore possible to run a simulation to a key point, then use the log to return to that point, continuing with different sets of future inputs to evaluate alternative possibilities.

**Event Traces**

One of the guidelines is to group similar information, avoiding the requirement that the user make the mental context switch of watching the state transitions unfold in time. Watching through the animation requires the user to remember the sequence of state changes. One naive solution would be to have the highlighting of state values recede slowly. In this way, the most brightly lit state is the current state, while the more muted trail of states behind it slowly fades to the background color. This is not desirable. The human eye can only distinguish between a few shades at once. Having those shades reinforce another obvious trend, like a rising histogram line, is helpful. However, having the shading of the state variables clash with the natural tendency to read a vertical progression is likely to cause confusion.

A better approach is to create a new visualization view altogether: the event trace. The goal of this view is to lay out the sequence of state transitions. The user selects each signal to add to the visualization. When the simulation runs, the simulation log is produced, as normal. When the user switches to the event trace, the display is populated from the information in the log.

At the top of the event trace, each signal of interest is displayed as the heading of a column. At the left, each row is marked with a round number and a timestamp.
that comes from multiplying the round number by the heartbeat frequency used for the simulation. Rows are only created if one of the signals of interest changed value in that round. Intervening rounds where nothing interesting happens are omitted. For each row, in the column of the signal that changed value, the old value of the signal is printed on the left, followed by a right arrow pointing towards the new value taken by the signal.

The reason for reprinting the old signal is again to reduce the load on short term memory. The alternatives are not good. Without displaying the previous state in the column, the user would have to scroll up until the point at which the state last changed. This creates a burden on short term memory while flipping contexts, voiding the point of this method of visualization. The other option is to reprint the current state of the signal in every row. This will create a sea of text, in which finding the signal that changed will be difficult. For this reason, only entries involving a change are filled in.

The simulator must include another option in this view, however, to avoid requiring excessive burden on short term memory. As it stands, the event trace view only lists changes in signal value. In a trace with many columns, a signal may not change values often enough to be present on the page the user is scrolled to. If the user wishes to correlate one signal’s value with the transitions in another, the user may have to scroll back to previous transitions. It is necessary to provide the user a way of unhiding the signal values of any cell on the spreadsheet-like view, as well as columns, rows, and arbitrary rectangular regions. To do this, the user selects the row, column, cell, or region to be unhidden, and uses a menu option, “unhide all values”. When this is done, all the values for those cells will display, even if they are not transitions in value. Fortunately, the transition cells, as they have two values and an arrow, will still be visually distinct. A soft coloration of the arrow would further distinguish cells containing arrows, and thus transitions, from cells that merely have static values.
3.3.6 Annotation Mechanism

Small sticky-notes are invaluable for aiding the memory. Much of the rationale for choices made during system development is recorded in comments and margin notes throughout specification, design, and implementation. Because the simulator will be used to demonstrate, review, and evaluate a specification, a method for annotating the simulation results is important. A user has full control to pause a simulation during its display, stepping backward or forward at will. While the simulation is paused, the user may make a note. The note is a text buffer, though the text buffer may include URLs that link to further information. From then on, whenever that simulation log is viewed, the note will be present. When the time step that has the note is reached, an icon indicating the presence of the note will be added to the screen.

In the main window, which shows the graphical representation of the state machine, this note will appear wherever it was placed when created. (If the note was created outside this window, it will appear at the bottom left of the window.) In the time series event trace, the note icon will appear to the left of the table next to the row that contains the round in which the note was created.

Tracking when in the simulation notes were made allows developers to ignore notes that are intended to regard a different place in a simulation. This preserves the context of the notes. It is possible, however, that a user might want to review all notes without regard to the time at which they were inserted into the log. For this purpose, a menu option will produce a list of all notes, allowing a user to peruse them in any desired order.

3.3.7 Design Support

The simulator is set in an application framework [4] that permits components to share data representations. While it is not specific to the simulation to provide design critics, separate modules may be written that use the same intermediate form. If this were done, heuristic design criteria could be applied to the state machine under simulation, recommending improvements. One example would be a critic that
ensured that a state machine model did not contain any cycles of signals depending on each other’s value.

### 3.3.8 Configuration

All configuration options can be saved and loaded at will. Simple, but important, details under user controllable configuration include color schemes, key binding, and sounds.

### 3.3.9 Direct Feedback

One feature that’s distinctly missing from the simulator is its ability to act as an editor. A case could be made for the possibilities of direct feedback in such an environment. One could simulate a string of inputs, discover an error, fix it, and then continue simulating right on from that point. However, this feature introduces a great deal of complexity. The simulator would, in its log, have to track editing changes to the state machine specification. Additionally, there would be no assurance for the user that the state machine under simulation now was the same as the one under simulation a few moments earlier! This would create a strongly inconsistent environment. For these reasons, editing of the state machine specification is purposefully left out of the simulator. There are, of course, also numerous architectural and design reason for not blending editing and simulating functionality.

### 3.4 Contributions

Human-Computer Interaction is a broad field, covering all aspects of user interface design. As a result of this field’s work, there are general guidelines covering the development of user interfaces. Visualization for software engineering is a smaller topic; no comparable sets of guidelines have been developed. This thesis compiles a survey of current work in visualization support for software engineering. From this survey, the powerful ideas spread across current research are gathered into guidelines,
principles, and features. Similar guidelines from more general purpose user interface design and printed material design are adapted to software engineering visualization.

Ideally, these ideas would be lab tested. Human trials, however, are lengthy, expensive, and error prone. A more modest test of these guidelines and principles is to use them in the design of a sample application. This thesis contributes a sample design, in this case for a state machine simulator. The principles and guidelines developed here were used in the design of this sample application, with good results.
Bibliography


Innovation and technology in computer science and education, pages 111–114, Krakow, Poland, June 1999. ACM.


[39] Gregor Kiczales. What can programming languages contribute to software engineering, and vice versa? (panel). In *Proceedings of the fourth ACM SIGSOFT
symposium on Foundations of software engineering, page 2, San Francisco, CA USA, October 1996. ACM.


intelligent user interfaces, pages 141–144, San Francisco, CA USA, January 1998. ACM.


