Program understandig in Gforth/Forth

A graphical approach

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Mario Gastegger

Matrikelnummer 0726289

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Martin Ertl
Mitwirkung:

Wien, 1. Jänner 2001

_____   _____
      Mario Gastegger   Martin Ertl

Program understandig in Gforth/Forth

A graphical approach

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Mario Gastegger
Registration Number 0726289

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Martin Ertl
Assistance:

Vienna, 1st January, 2001

| Mario Gastegger | Martin Ertl |

Erklärung zur Verfassung der Arbeit

Mario Gastegger
Ameisgasse 15/9, 1140 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Jänner 2001

_____

Mario Gastegger

Danksagung

Ihr Text hier.

Acknowledgements

Enter your text here.

Kurzfassung

Ihr Text hier.

Abstract

200-250 wordsasd or 10-15 lines Enter your text here.

Contents

List of Figures

List of Tables

Introduction

Motivation

Software systems underlie continuous changes throughout their whole life cycle. They evolve from the beginning of development until their release and maintenance phase. Large software systems[1] and most of all, software classified as type E [CHLW06], get more complex over time.
If there are more than a few developers/development teams involved or the developers/development teams are spread allover the world, there exists more foreign code than self written code. This makes it necessary to understand foreign code. Most of all when it comes to making changes, enhancements or fixes, there is a very high level of understanding of the software at hand necessary[Boe76][SLea97]. Due to [CZvD+09], "... up to 60% of the maintenance effort is spent on gaining a sufficient understanding of the program ...". These facts emphasize the need for computer aided methods to improve program understanding.

This thesis addresses the task of improving program comprehension of the concatenative programming language Forth on several levels. The terms "program understanding" and "program comprehension" are considered synonym. Namely the reading of source code, static analysis, dynamic analysis and the assistance of writing readable and easy to understand source code.

Due to the nature of concatenative languages, it is possible to write source code which reads very similar to natural language. There are no hard boundaries to the structure of the source code(custom defined loops and control structures) as in most other languages. Since Forth directly operates only on stacks and memory, the information which is immediately needed to follow the program execution is limited to those structures. In

---

[1]"The term large is, generally, used to describe software whose size in number of lines of code is greater than some arbitrary value. For reasons indicated in [leh79], it is more appropriate to define a large program as one developed by processes involving groups with two or more management levels."[LR03]

contrast, in object oriented languages there is also object state, object life cycle and concurrency to keep in mind.

Problem statement

There has been done plenty of work on the task of program comprehension in object oriented and procedural languages[CZvD$^+$09], but nearly none for concatenative languages.

Aim of the work

This work aims to give a brief overview on the field of program comprehension and its methods, to show existing aids to program understanding in Gforth/Forth and to study the applicability of some existing analysis and visualization approaches for other paradigms. Further more the suggestion of new methods or the modification of existing methods to meet the characteristics of Gforth/Forth.

Structure of the work

In Chapter 2, I will line out the methodological approach of this thesis. In Chapter 3, there will be a brief overview on the topic of program comprehension, an overview on the existing tools to improve program understand in general and on those specific to Gforth/Forth and afterwards I will present a selection of visualization approaches for procedural and object oriented languages. In Chapter 4, I will present some of the previously mentioned approaches for other paradigms applied on a real world Forth program, analyze their applicability and propose modification to those approaches. Afterwards I will present a prototype implementation of a program trace visualization enhancement to Gforth. Chapter 5 concludes the thesis with a summary and topics of further investigation.

Methodology

This chapter contains the methodological methods used to approach the topic of this thesis.

I will use a qualitative approach. In Chapter 3, I will gain an over view of program comprehension and it's techniques in general and I will summarize the currently available techniques for Gforth/Forth. The main resources for that was [CZvD$^+$09] and the keyword search on the ACM digital library[1]. Since I will investigate the applicability to Gforth/Forth of only a handful of concrete techniques, I used the following criteria for selection:

- Graphical visualization

- Applicable to imperative languages(procedural and object oriented)

- Behavioral analysis

- Suitable for trace visualization

One probable issue, introduced by the initial keyword search, is the lack of previous knowledge on the topic of program understanding, trace visualization and concatenative languages. Another one is the lack of experience with Gforth/Forth.

In Chapter 4, I will analyze the graphics which would be produced as a result of the selected techniques. First, I will introduce the software on which I applied those techniques. The software was chosen with the following criteria in mind:

- Size: The software should consist of at least hundred kilobytes of source code.

- Complexity: It should have at least a medium level of complexity.

- Practicality: It should solve a real problem.

---

[1]`http://dl.acm.org/`

Since software characteristics can vary greatly depending on the domain, the selection of only one program might not be meaningful enough to confirm, whether or not a certain technique is useful for arbitrary Gforth/Forth programs. Second, I will analyze the graphics in an exploratory fashion and propose modifications of these techniques to improve their usefulness. Due to the lack of automated implementations of the selected techniques for Gforth/Forth, I will produce these graphics manually. Third, I will introduce the prototype of an enhancement for the Gforth step-debugger which was implement in the course of this thesis and discuss its usefulness.

It is obvious, that due to the chosen methodology, there will be no quantitative confirmation for the usefulness of the techniques under investigation. Due to the limited selection of techniques, there may be other methods which maybe invaluable to program comprehension, but are not covered by this thesis.

Analysis of existing approaches

The focus here lies on so called E type software systems(E stands for evolving [CHLW06]). Most of the real world software systems are type E. These systems are of particular interest since they underlie continuous changes throughout their whole life cycle. Thus understanding existing code, which might not be well documented, is crucial for the maintenance of those systems.

3.1 Program comprehension

Program comprehension can be gained following various approaches. First of all, by reading the code. In [Bas97], Basili et al. approach the concept of reading on a very fundamental level. The natural way to learn writing is to learn reading first. The reading then forms a model for our writing. His research shows that reading is most effective compared to testing. This suggests, that readability of code does impact the efficiency of failure discovery. According to Basili et al., the most severe problem is the fact that programming languages are learned the other way round. We first learn to write code and then learn to read it. Furthermore, the ability to read code is not properly addressed in education. The syntactical flexibility, Forth provides, compared to other languages(and paradigms), allows it to achieve a very natural seeming reading experience. Thus our skills in natural languages could become in handy and make program reading and thus understanding even more efficient. This would in the end result in higher code quality in terms of failures and unexpected or unintended behavior.

There emerged several strategies on how to read and understand a program[SFM99][SWM97]:

Top down program comprehension

Using the top down strategy, the reader begins on the highest level of abstraction, the main purpose of the program and then builds a hierarchy by refining it into sub tasks until the lowest level of abstraction is reached.

Bottom up program comprehension

Using the bottom up strategy, the reader builds the mental model by grouping low level parts of code to build a higher level of abstraction until the whole program is understood.

Knowledge based program comprehension

The knowledge based strategy, allows both, the bottom up and the top down approach. The assumption is, that programmers have a certain mental model of the software, this model is evolved by both refinement and abstraction.

Systematic and as-needed program comprehension

This strategy embodies detailed reading as well as only focusing on the code necessary to fulfill the task at hand.

Integrated approaches of program comprehension

This strategy allows freely switching between the top down, the bottom up and the knowledge based approach.

As Storey et al. point out in [SFM99], there are certain factors which influence the choice programmers take. Thus programming environments should provide methods to support all of these strategies.

Since type E software evolves throughout its whole life cycle, also the before mentioned mental model of the reader has to evolve. It has to be kept in sync with the software system. This suggests, that it is essential to keep all types of artifacts(documentation, source level documentation, graphics,...) up to date.

3.2 Analysis to support program understanding

Besides reading of the source code and other textual documentation, there are also other methods to increase program understanding.

3.2.1 Dynamic analysis

Dynamic analysis is performed on the image of a program, executed on a real or virtual processor. The advantage is, that due to the availability of the data to be manipulated, the actual behavior of a program can be investigated. The major drawback however is that there exists only an incomplete view of the software system at hand[Bal99]. Dynamic analysis is a very efficient way to evolve or correct the mental model of developers, but not to create it. E.g. in large software systems, some scenarios simply might not occur during analysis.

6

3.2.2 Static analysis

Although this thesis focuses mostly on dynamic analysis, for the sake of completeness, also static analysis should be mentioned here. Static analysis is performed on the source code. Therefore, and in contrast to dynamic analysis, it has the capability to provide a complete view of the software at hands. The drawback is that there is no actual data present and thus there are no means of covering the actual data follow and the manipulation of data.

3.3 Existing approaches

3.3.1 Gforth/Forth

In this section I'm going to present the tools, Gforth/Forth provides to support program understanding and maintenance[1].

Examining data and code

Gforth provides several tools to display data and code, which supports program understanding and software maintenance.
For displaying data, the most important words are *.*, *.s*, *."*, *type* and *dump.*
. and *.s* simply display elements of the data stack, there are also words to visualize the other stacks. The words *."* and *type* display text and *dump* displays memory areas(address, hex and ascii). ~~ displays the location of itself in the source-file(file and line number) as well as the data stack.
All these words can be utilized to display logging information.
They are usable in interactive as well in non-interactive analysis.
There are also words to investigate the inner workings of other words. *see* displays the definition of words written in Forth. It can be used to quickly look at the behavior of words provided by Gforth without looking into source-files. The use of *see* and its relatives only makes sense in interactive analysis.

status.fs

Status.fs is included in Gforth[2]. It opens a separate xterm window and displays the current number base, the float stack, the data stack and the current search order. The view is updated after each ⏎ in the Gforth interpreter. Thus it is only useful for interactive analysis.

---

[1]For further tools see `https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Programming-Tools.html#Programming-Tools`
[2]Since version 0.7.0

Stepping debugger

dbg[3], the stepping debugger, supports among others, single step, step into, step over as well as break points. It displays the address of the word to be executed and the content of the data stack after its execution. *dbg* is only usable for interactive analysis.

Assertions

Assertions[4] can be used to verify, that the program, at a certain point, is in a certain state. For example pre-conditions and post-conditions of words could be implemented using assertions. It is also possible to prevent code from breaking during maintenance activities. The word *assert(* starts the assertion, the following words until the *)* are executed and have to leave a flag on the stack. If that flag is true, the asserted condition is meet, otherwise, the execution of the program ends and the location of the failed assertion is displayed(source file and line number).
There are several assertion levels and by setting the *assert-level*, assertions can also be deactivated.

Documentation

Thorough and up-to-date documentation is undoubtedly important, this also applies to concatenative languages. Besides behavior description of words, since Forth is by default an untyped language, there is also a special kind of documentation encouraged to ease the understanding of words, namely the stack effect comment[5]. It is written next to the name of the defined word and contains the number and type of elements on the stack which are manipulated by the word. These comments describe the state of the stack before and after the execution of a word. Within these comments, there is also a distinction to be made between interpretation, compile and run-time behavior.

Words and word lists

Like in any other language, words should be named expressively. However sometimes, it may not be avoidable to reuse names. Forth provides an elegant mechanism, called word lists[6], to address these issues and to organize words. With word lists, words can be defined in a certain context. Like in natural languages words can have a different meaning in separate contexts. Using word lists, developers can prevent name clashes and separate interface words from internal words. In large projects it might be necessary to define a naming strategy.

---

[3]For further information see: `https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Singlestep-Debugger.html#Singlestep-Debugger`

[4]See `https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Assertions.html#Assertions`

[5]See `https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Notation.html#Notation`

[6]For more information on word lists, see `http://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Word-Lists.html#Word-Lists`

Factoring

As in any other programming language, it is necessary to split the overall problem down into manageable, less complex sub problems. In concatenative languages this is also referred to as factoring. Factoring helps keeping definitions short(and thus easier to understand), reusable and easier to test[7].

Aliasing

In Gforth, there can be multiple aliases[8] for one word. Aliases can be used to use the same underlying implementation in different contexts, to make code more readable.

Emacs forth-mode

The emacs forth-mode(gforth.el, which is based on forth.el) provides many helpful features[9] to ease the writing of Forth. Most notable, related to program understanding:

- Word documentation lookup

- Jump to line from, error messages, debug output, failed assertions and ~~ output

- Highlighting

- Indention handling

Kgforth

There has also been an afford to integrate some of those tools into a graphical development environment. The project is called Kgforth[10], but its development seems to be discontinued.

3.4 Applicability of methods for other paradigms to concatenative languages

In this section I will present a hand full of visualization methods and discuss their applicability to Gforth/Forth.

---

[7]https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/
Factoring-Tutorial.html

[8]See https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Aliases.html

[9]For a more information see https://www.complang.tuwien.ac.at/forth/gforth/
Docs-html/Emacs-and-Gforth.html#Emacs-and-Gforth

[10]http://sourceforge.net/projects/kgforth/: "Kgforth is a simple IDE for the gforth interpreter/compiler for KDE 2.** It provides an editor, gforth window,debug and dump window, forth toolbar and menu."

Sequence diagram and scenario diagram

The Unified Modeling Language sequence diagram can be used to model event sequences on any level of abstraction. Since I left out object oriented Forth and there is no standard for concurrency, the word-only level provides little help. On this level, the actors would be represented by words and the messages[11] would represent word executions. Therefore the sequence diagram would degrade to a list. On the system level although, the sequence diagram can provide useful information on the interaction between system components. The scenario diagram as implemented by [KM96], is essentially the same as the sequence diagram.

Since system level visualization is applicable to almost any paradigm and word level visualization does not seem promising for dynamic analysis, I will not investigate the benefits of those diagrams in this thesis.

Hierarchical edge bundles

Hierarchical edge bundles as proposed by [Hol06] display hierarchic and non hierarchic relations between nodes. In context of Forth, word execution can be mapped to non hierarchic relations and directory/file tree and the definition could be mapped to the hierarchic relations. Like in object oriented systems, the proper organization of directories, files and word definitions are a requirement to make such a visualization really helpful. Another possible mapping could depend on word lists as hierarchic and word executions as non-hierarchic relations.

The hierarchic edge bundle can be used for interactive and non-interactive(real-time) dynamic analysis. In the following chapter I will analyze both possibilities using a real Forth software system as an example.

Information murals and massive sequence view

The information mural was initially proposed by [JS98]. A modified version, the mass sequence view, was later proposed by [Cor09]. It turned the horizontal scrolling into vertical scrolling and also included the hierarchic aspect more suitable. As with the hierarchical edge bundles, the usefulness of the hierarchical relations depend highly on the organization of the software system at hand.

As well as the hierarchic edge bundle, these two methods can be used for interactive and non-interactive(post-mortem) dynamic analysis. In the following chapter, I will analyze the massive sequence view using a real Forth software system as an example.

High-Level polymetric views

Polymetric views[DLB04] are a very interesting and promising approach to grasp the behavior of very large systems. In polymetric views, system attributes or measures are mapped to attributes of a graph. The attributes proposed by [DLB04], are position,

---

[11]The edges in UML's sequence diagram are called messages.

height, width, color and the relations(and the thickness) between rectangles, representing aspects of the software to analyze. In terms of Forth, the mapping to words seems most obvious. This would result in some kind of a word cloud with additional information attached. This could be most valuable, when used with the appropriate metrics, to analyze and optimize performance. Another advantage is, that there is no complete, but only condensed data[12] required.

This method is suitable for interactive as well as non-interactive analysis. In the following chapter, I will analyze a polymetric view of real software system as an example.

Fisheye views

Fisheye views were first proposed by [Fur86] and formulated by [SM96] and [SB94]. The essence of fisheye views is a principle also found in nature. It's basically described as a function, expressing the degree of interest of an subject, depending on an a priori importance and the distance from the current point of view. Growing distance lowers the degree of interest.

This method is related to the above mentioned polymetric views.

Execution pattern view

The execution pattern view, proposed by [PLVW98], visualizes large traces in a scalable manner and helps to identify execution patterns. It represents an interesting evolution of simple sequence diagrams and interaction diagrams.

Since it is somehow related to the massive sequence view, I will not pursue this approach in detail.

Method invocation view and taxonomy view

The tool GraphTrace[KG88] is meant to analyze object oriented programs. It provides a method invocation view and a taxonomy view. Although the method invocation approach of GraphTrace seems not practical for Forth programs, the authors mentioned two very interesting ideas.

- Displaying variable access:
  The idea of showing words which access variables and the other way round, showing which words access a specific variable is very interesting. Thus it would be easier to track the global state of a program.

- Concurrent views:
  The authors of [KG88] present a quiet interesting analogy. They compare the execution of a program with tennis and football, and refer to the multiple perspectives necessary to understand all aspects of a match and the whole outcome.

---

[12]There is no need to have a complete execution trace.

The variable access graph in a tree like visualization as in GraphTrace seems highly useful since variables represent a global program state, which can increase complexity. Although this approach seems to make only sense for in static analysis, I will analyze the variable access graph of real software system in the next chapter.

Frequency spectrum analysis

The frequency spectrum analysis as proposed by [Bal99] represents also an interesting approach. It is similar to polymetric views as mentioned before, but the visualization is not graphical but textual. The word execution frequency could provide valuable information about the actual performed work and incorporated with execution time, it could help in understanding programs and support performance analysis. In [Bal99], Ball shows the application of frequency spectrum analysis at the example of an obfuscated c program, but his approach should also be applicable to concatenative languages.

Since this is not the main focus of this thesis, I wont discuss this approach any further.

3.5 Methods to support code readability and to improve program understanding

In this section I will suggest methods to write better and more natural reading code as well as methods to ease understanding of existing programs.

A very important question is, how developers can be assisted to write more readable code. Concatenative languages are flexible enough to produce code very similar to natural languages, but how can this style be encouraged? In my opinion, the top down strategy and extensive use of factoring, is the key to produce readable Forth code in large software systems.

Going down from the top, the programmer writes description of the work to be done right out of the requirements documents(e.g. user stories). He defines single words or groups of words and describes them in more detail. This process is repeated until the technical level is reached and actual Forth words are written.

To encourage high code quality and the development environment should provide hints based on static analysis.

3.5.1 Common issues and possible solutions for writing readable code

It is not possible to make every word completely readable and the perceived readability also depends on the experience of the developer. At some point it always comes down to longer combinations of *nip, tuck, over, rot* and so on. This is hardly avoidable at the lowest level. Thus, proper documentation of words is essential. It is pretty obvious, that stack effect comments[13] in Forth, are a must have, but also the behavior of the words should

---

[13]See   https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Stack_002dEffect-Comments-Tutorial.html#Stack_002dEffect-Comments-Tutorial

be documented. Especially for complex or not very natural to read words[14]. Another advantage of word definition comments is the possibility of automated documentation generation.

Using this approach, the developers can stick to natural language like words until there is no more factoring possible and the words have to be implemented in low level Gforth words.

Another common problem are very long word definitions, they tend to increase the amount of brain capacity required to understand their behavior. To address this problem, the word hierarchy created by factoring, should be kept slim and deep. One approach here could be to display hints for word definitions, which exceed a certain amount of lines or words or different words, and suggest further factoring.

Another tool to make code reading more natural, is aliasing. By defining aliases for a certain word, its functionality can be used in different contexts and still read very natural.

3.5.2 Common issues and possible solutions to ease program understanding

Due to [RCM04], the systematic approach turned out to be most efficient to understand code. To ease the afford of finding the definition of words, used at a certain point, a hyperlink like referencing mechanism can be used.

As stated by Charles D. Moore in [BW09]: "... The challenge there is 1) deciding which words are useful, and 2) remembering them all.". When programs get grow, the amount of words can get huge. Thus it is suggested to have some sort of a dictionary to search the whole vocabulary by name, stack effect comment, word definition documentation and provide a reference to where they are used. Auto completion can also help a lot in finding words previously defined.

In software maintenance, a common issue is, when fixing a bug, first to identify the source code areas which are responsible for the malfunction. And second, to understand the meaning of those code areas to be able to fix the bug. The before mentioned massive sequence view can help in identifying those code areas in a large program trace.

Another issue in software maintenance is the identification of changes which are necessary to implement a new feature or modify an existing feature. In [SWFM97], Storey et al. presented an integrated approach, which combines some of the before mentioned visualization techniques. But instead of stopping at the file level, for Forth, it would be event more helpful to go down to the word level.

---

[14]Most notable
G in Gforth. See `https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Comments.html#Comments`

Results

In this chapter, I will introduce the chosen Forth software system. Afterwards, I will present the manually or semi automatically produced graphs of the selected visualization methods.

4.1 The software under investigation: Brainless

Brainless[1] is a chess-playing program written in ANS Forth. The source code consists of several files with an overall size[2] of 139497 bytes and 4108 lines of code[3]. This measure is somehow controversial, but since the files contain only a short header and are formatted in the usual manner, it seems appropriate for comparison. The code is organized in a flat structure. There is one directory with 30 files, which contain 663 words. There is only one custom word list defined. Thus the visualization of a word list hierarchy makes obviously no sense and is left out in the following sections.

The other software system, I took into consideration, was brew[4]. Brew is a 'playground for evolutionary programming', as the author calls it. Due to its size of 1062857 bytes and 36801 lines of code, this project seemed too large to be analyze manually in reasonable time.

For the following figures, I used a snapshot of an execution trace. Since the calculations of the computer-moves produce a huge amount of word executions, the example trace, was created by making only the player-move: *d2 d3 m* ↵ . The snapshot contains all word executions after and including the execution of the word *m*. It consists of 4709 word

---

[1]Brainless verion 0.1.2 is used, the source code can be optained on `http://sourceforge.net/projects/forth-brainless/`

[2]The command to calculate the size was *find . -name '*.fs' -maxdepth 1 | xargs wc -l*

[3]The command used to count the lines of code, was *find . -name '*.fs' -maxdepth 1 | xargs wc -l*

[4]brew version 0.2.0 was used. The source code can be obtained on `http://www.robertepprecht.ch/brew/index.html`

executions. Figures 4.1 and 4.2 show the state of the game before and after entering *d2 d3 m* ⏎.

4.2   The application of the previously presented visualization methods

Hierarchical edge bundles

Figure 4.3 shows the first 100^5 word executions of the before mentioned trace snapshot in a *hierarchic edge bundle.*

The outer ring labeled with *brainless-0.1.2* represents the directory of the source code, the middle ring represents the files and the inner ring represents the words. Words which haven't been executed in this snapshot, have been omitted. The word sectors are of equal angle, the number of different words divided by 360°. The file sectors' angle have been adjusted according to the number of files they contain. The lines in the circle represent word executions. Same as in [Hol06], the word on the green end is the caller and the word on the red end is the callee. The distance between the turning point and the center of the circle depends on the distance(in degree) of the two word sectors, words with great distance are connected near to center. Multiple occurrences of caller/callee pair appear as one line in this graph. The exact algorithms used to calculate figure 4.3 can be found in <appendix>.

Interpretation

The main problem in this manually generated picture is the limitation to 100 executions. Due to rendering time and memory requirement, it was not possible to visualize more executions with KTikZ[6]. Apart from that, the hierarchic edge bundle provides information on the complexity or importance of a word in this part of the execution trace(the number of words which are executed by one word, the number of distinct words which are executed by one word or the number of words which call one word).

Looking at the interaction lines, what stands out first, is the role of board.fs as caller and the role of threats.fs as callee. This leads to the assumption, that board.fs must contain controlling functions in this part of the trace. In contrast, threats.fs provides mainly utility words for board.fs.

Possible improvements

A further improvement would be to draw multiple occurrences of the same caller/callee pairs(the interaction lines) with a slight gap to visualize all the executions. Or, if there are too many interaction lines, to show the exact number of interactions e.g. on hover.

---

[5]The calculation of the interaction lines consume a considerable amount of system resources, which prevents longer traces due to memory limitation.

[6]Version 0.10

Figure 4.1: State of the game before entering *d2 d3 m*

Figure 4.2: State of the game after entering *d2 d3 m*

Another improvement would be to show the names of the words in the hierarchic part, or if there are too much, to show them e.g. on hover. Provided, that words are named expressively, the hierarchic edge bundle can also help to extract information on what happened during execution and how something happened.

Due to the limitation to 100 executions, the not involved words could have been omitted in the hierarchic part.

Information murals and massive sequence view

Figures 4.4, 4.5 and 4.6 show the trace snapshot in a *massive sequence view.*

The *massive sequence view* consists of the hierarchical part at the top of figure 4.4 and the interaction part which follows immediately(figures 4.5 and 4.6). The upper level of the hierarchical part, labeled with *brainless-0.1.2*, again, represents the directory of the source code. The middle level, the files and the lower level, the words. Again, words which haven't been executed in this snapshot, have been omitted. The interaction part shows the word executions as lines. As in [Hol06], the word above the green end is the caller and the word above the red end is the callee. The order of the interaction lines represents also the order of execution. The first executed word is represented by the upper most interaction line.

Interpretation

With the massive sequence view, it is easy to identify certain steps of the program execution like the drawing part at the end of the trace(interaction with drawing.fs shown if figure 4.5) and the file writing part(interaction from epd.fs at the end of figure 4.5), if one knows the words defined in those files.

Possible improvements

Figures 4.4, 4.5 and 4.6 clearly show the lack of interactivity. Without filtering, zooming, on-demand information on words(references to source code) and of course expressive naming, it is still hard to map sections of the trace to behavior. Besides these concerns, the massive sequence view seems to be well applicable to Forth program traces.

High-level polymetric views

Figure 4.7 shows a polymetric view of the snapshot. I used a circle to represent a single word. The radius reflects the number of executions, frequently executed words appear as larger circles. The position(distance from the origin) reflects the number of words executed within the word, fewer sub-word-executions result in a greater distance. If the number of sub executions vary, the maximum was used. The color of a circle reflects the io-behavior of a word. Red means, the word prints to stdout and yellow means it is reading from or writing to a file.
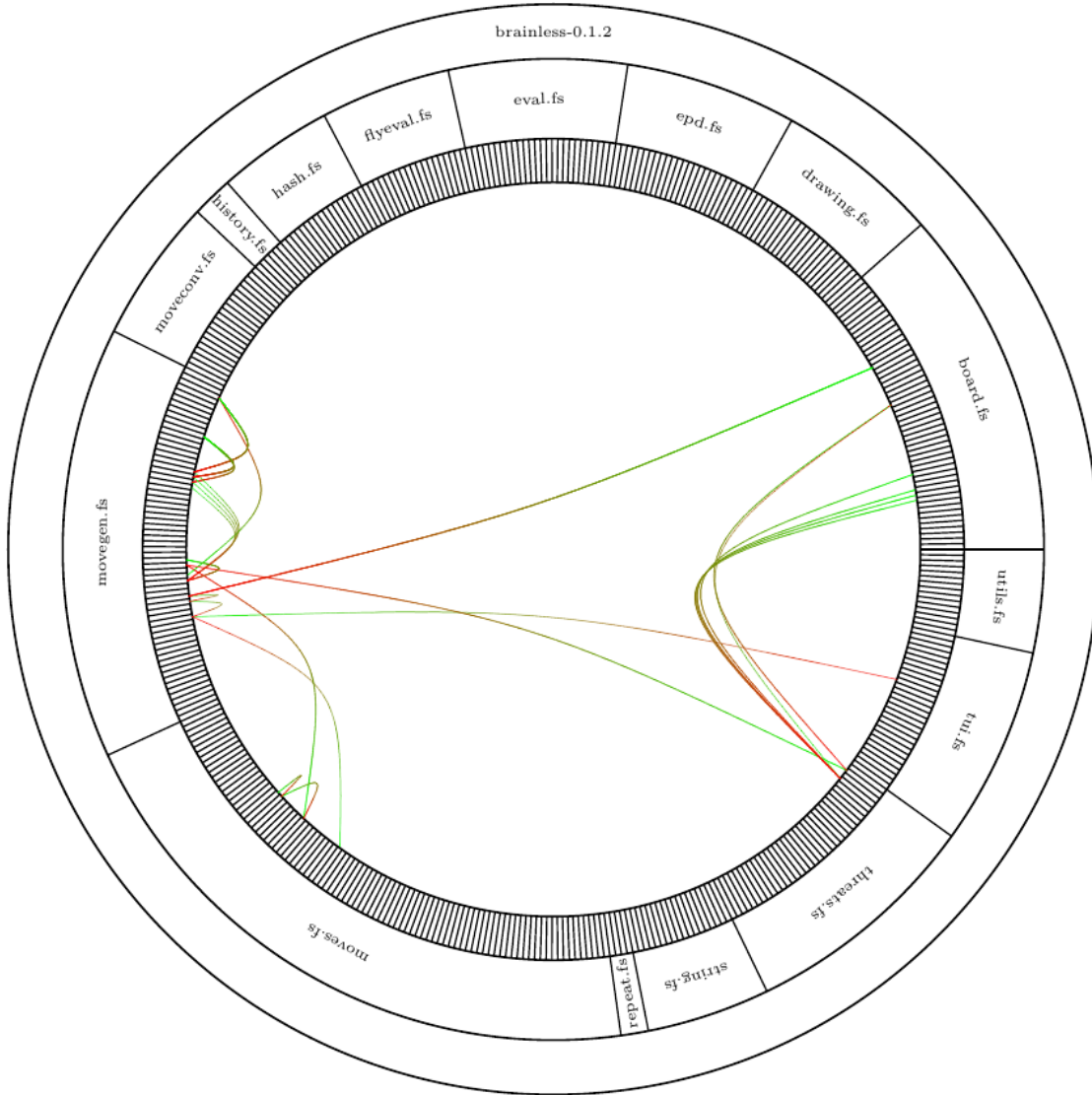
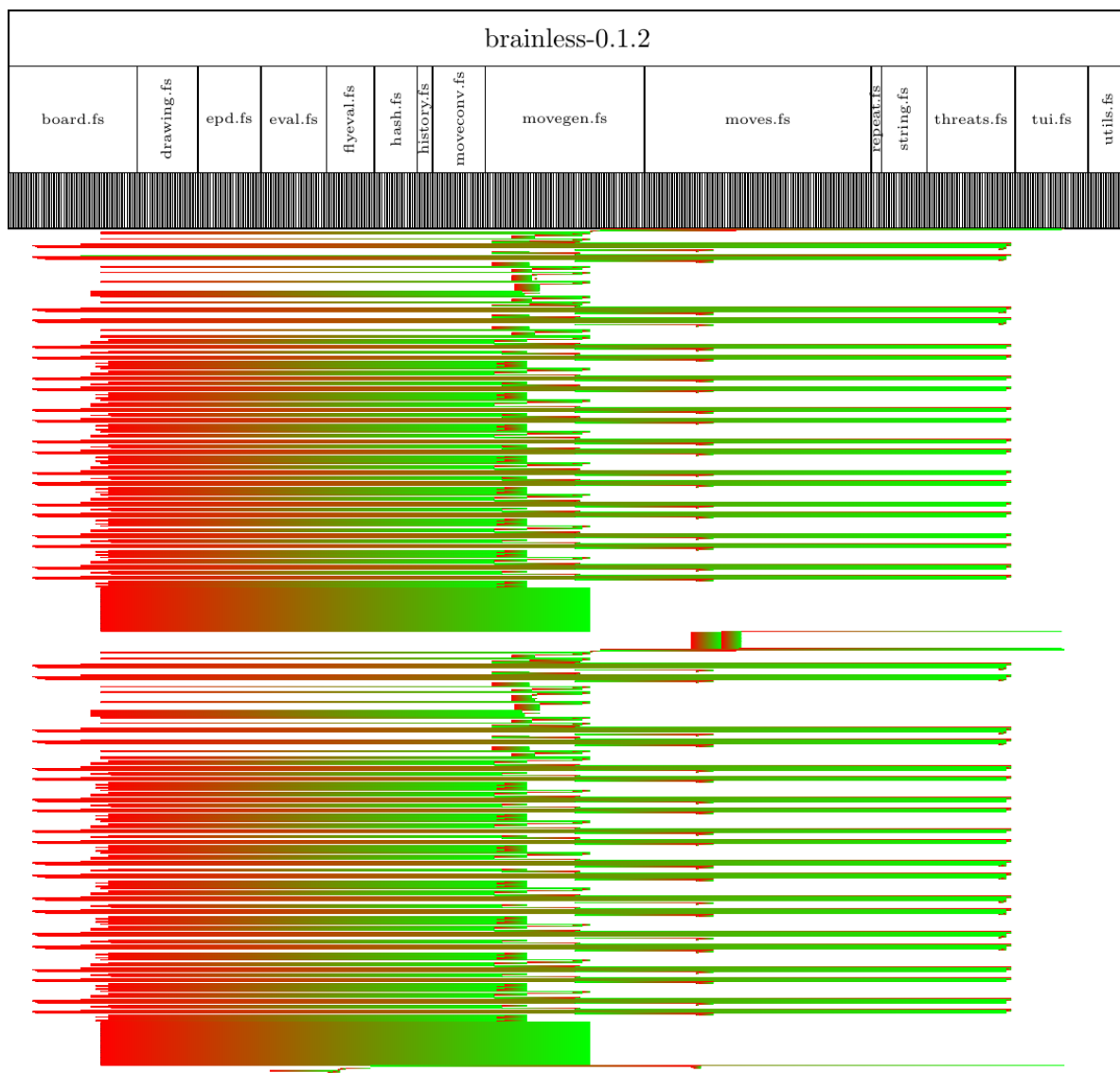Figure 4.3: Hierarchic edge bundle of a small snapshot of the trace of Brainless after *d2 d3 m*

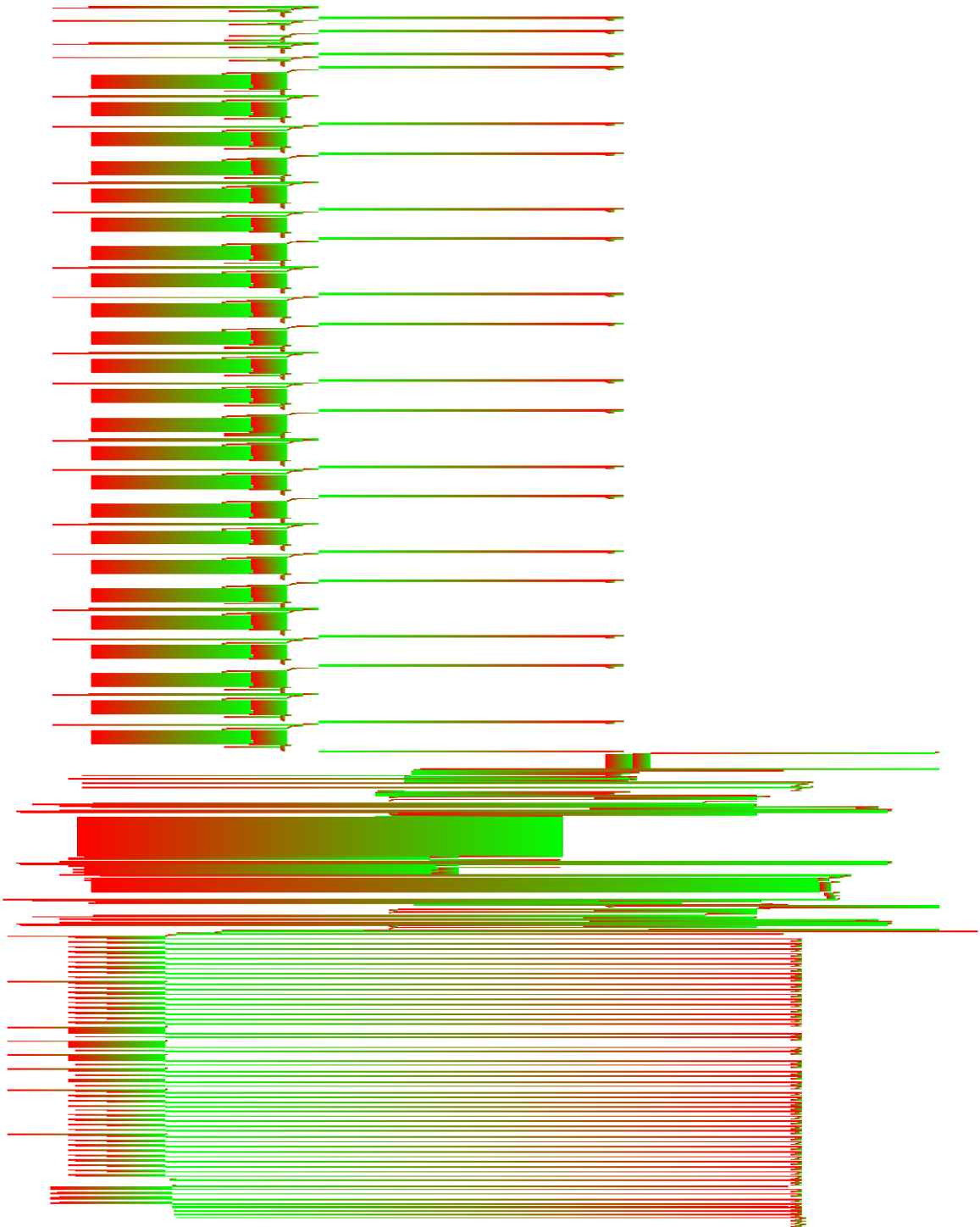Figure 4.4: Massive sequence view(Part 1) of Brainless after *d2 d3 m*

Figure 4.5: Massive sequence view(Part 2) of Brainless after *d2 d3 m*

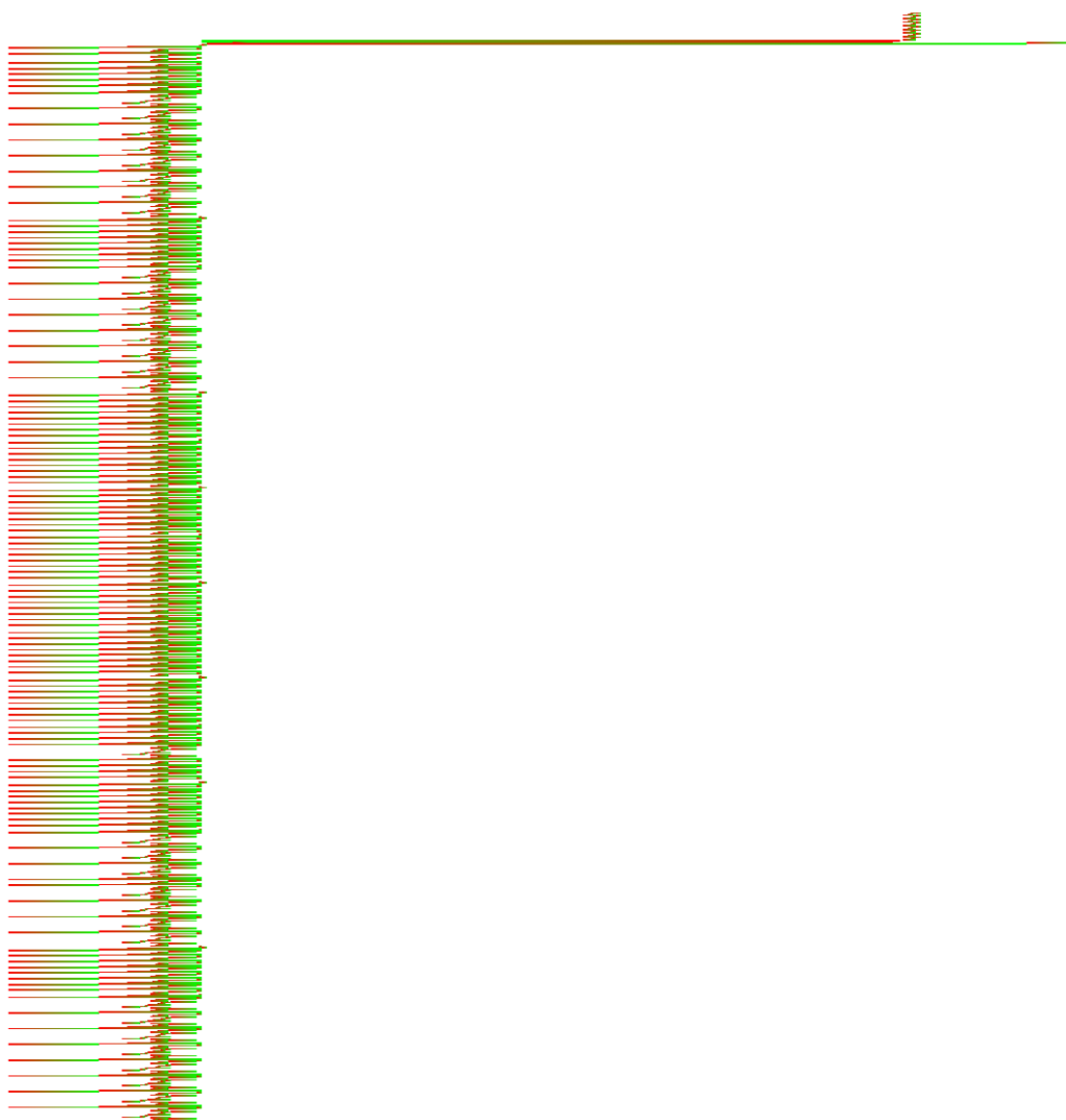Figure 4.6: Massive sequence view(Part 3) of Brainless after *d2 d3 m*

Interpretation

Figure 4.8 shows a snapshot of the whole graph, containing the most often executed word(*get-piece-masked*). This information can be valuable to identify words, suitable for performance improvements. Since optimizing words which are executed very frequently can significantly improve performance.

Through its short distance to the origin, the words *m*, *knight-moves* and *.board* can be identified as high level words. At this point, it seems misleading, that the word *knight-moves* is closer to the origin than *pawn-moves*, since the move d2 d3 was the move of a pawn.

Through the colors, it is easy to identify the words *epd-write-to-file* and *epd-append-to-file* as words with file interaction. The words *.hborder*, *field-spaces* and *display-move* can be identified as words responsible for drawing the chess board.

Possible improvements

This graph was generated of an execution trace but it could also be used to show information gathered from static analysis. For example, it would be interesting to map the number of words within one word to the radius of the word's circle. And use the color of the word circles to highlight very complex words(with too many words in their definition or too many conditionals or nested loops).

Memory access view

Figure 4.9 shows a small snapshot of the memory access of brainless. It shows the words and the memory locations they access. The words are represented by circles and the memory locations(values) by rectangles. The arrows represent the direction of the access. An outgoing arrow(from a word to a memory location) represents write operation and an incoming arrow(from a memory location to a word), a read operation. Since Brainless makes extensive use of *value*s, does not use **variable*s at all and uses custom defining words only occasionally, figure 4.9 shows only the *value*s.

Interpretation

Since the graph is very huge, it is not easy to keep track of the entire picture, but for debugging, it could be very helpful to see which other words manipulate certain memory locations to find the actual cause of a problem.

Possible improvements

It should cover *value*, *variable*, *2variable*, *fvariable* and also memory fields, allocated by custom defining words. Figure 4.9 shows only a small snapshot of the whole graph, because the whole graph would have been too large(the graph grows with the number of words, which access memory locations).
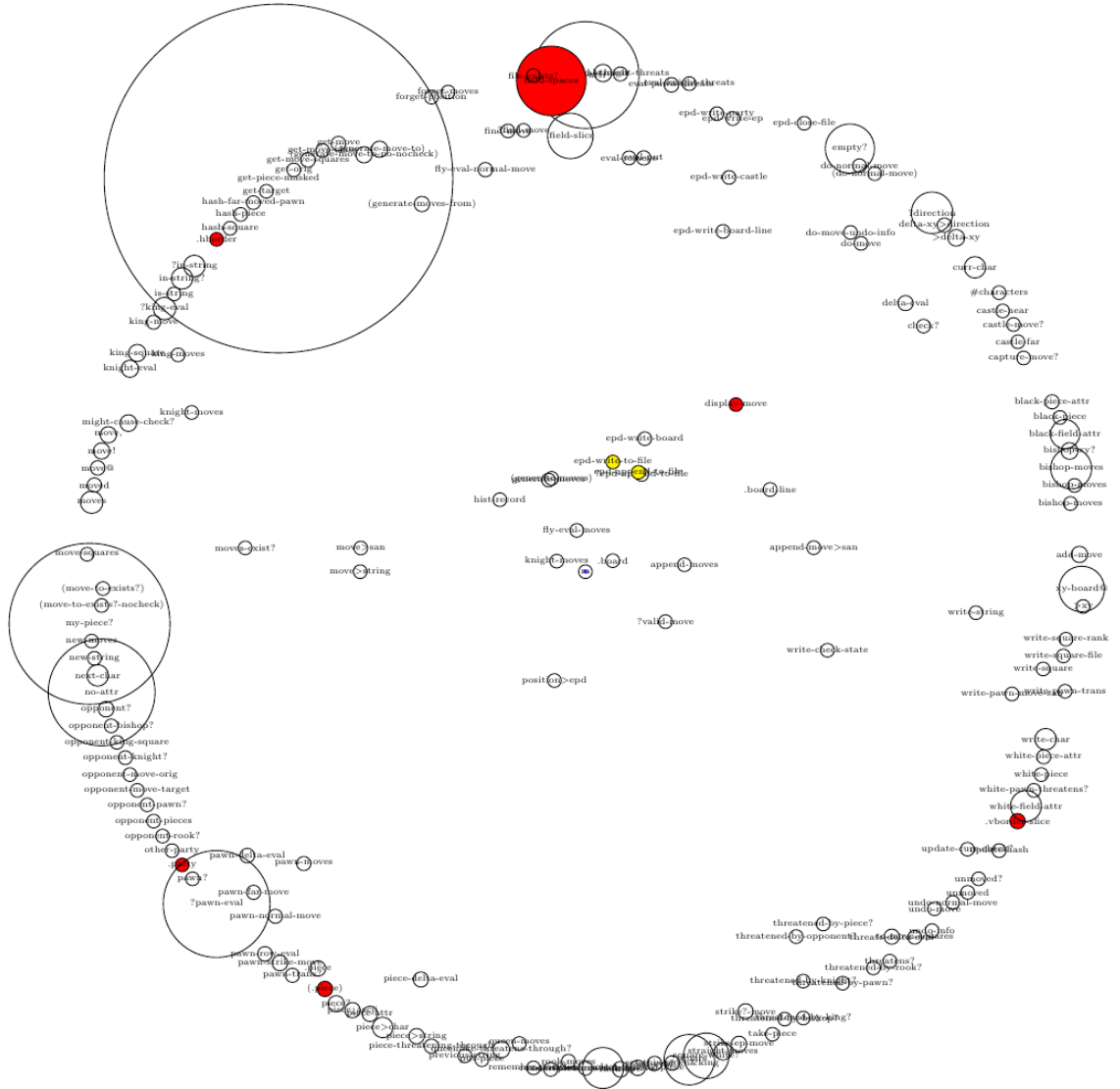
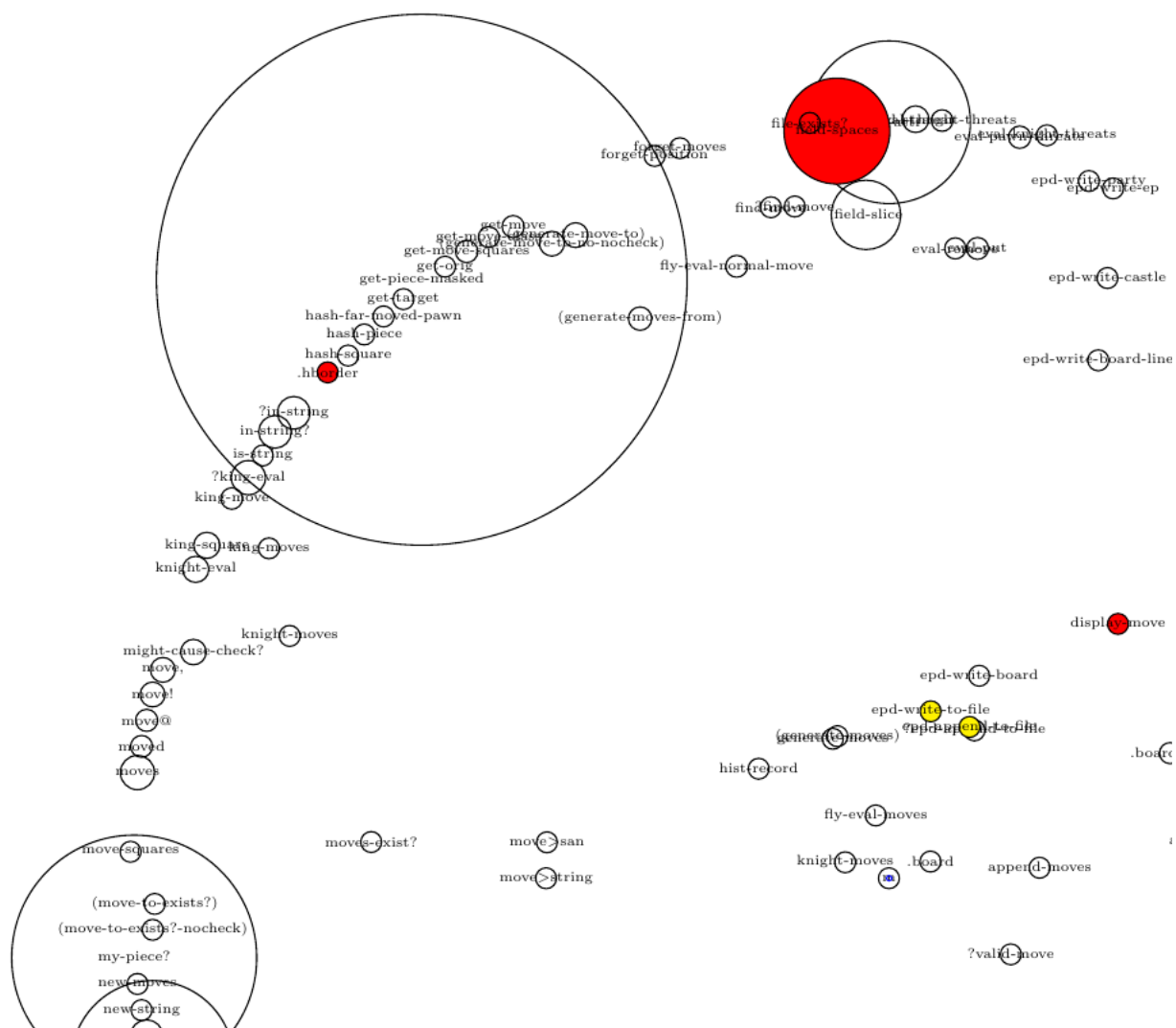Figure 4.7: High-Level polymetric view of the trace of Brainless after *d2 d3 m*

Figure 4.8: Sanpshot of the High-Level polymetric view of the trace of Brainless after *d2 d3 m*

4.3 gfvis - A trace visualization enhancement for Gforth

Gfvis was developed within this thesis, it is an enhancement for Gforths debugger *dbg*(See Chapter 3). Gfvis runs with a slightly modified version of Gforth(modified gforth-itc) and requires *gv*[7]. It consist of *gfvis.fs* and *gfvis.ps*. Both have to be in the same directory. *gfvis.ps* is a template file, written in postscript, it contains code to display the executed words and the state of the data stack, the floating stack and the return stack. *gfvis.fs* contains code which collects the information to be displayed and to create and update *trace.fs*. Gfvis is started by executing `<path to modified gforth>/gforth-itc ./gfvis.fs`. When the dbg is started(i.e. `dbg test`), trace.ps, a copy of gfvis.ps, is created and gv is started to display its content(See figures 4.11 and 4.12). From then on, every executed word, updates trace.ps with the name of the word and the state of the stacks. When `bye` is executed, gfvis terminates gv and gforth-itc.

Interpretation

In figures 4.11 and 4.12, I used the word *test* to demonstrate the use of gfvis. *test* contains conditionals, numbers, mathematical operations, sub word executions and it prints text to stdout. The source code of *test* can be found in figure 4.10. With gfvis it is easy to follow the inner working of *test.*

Possible improvements

The first problem of gfvis is the very inefficient use of screen real estate. It would be better to show only the current and the previous stack state. The complete history should be shown only on demand and the displayed stack depth could be limited to a few elements(e.g. the 5 uppermost elements).

The second problem is the growing rendering time of *gv*. With every new step, the whole files has to be re rendered, which takes a considerable amount of time for longer debugging sessions.

Another improvement would be to show not only the stack state, but also the content of *value*, **variable* and custom defined memory.

Displaying all allocated memory areas and keeping track of changes within those areas in a *diff* like manner, would be helpful.

Better visualization of beginning and end of control structures and especially loops would also be an improvement.

Currently the stack states and executed words are stored within postscript code. A further improvement would be to store those traces in a standardized data format like json or xml.

---

[7]GV is a postscript viewer for X displays. For further information see: `http://www.gnu.org/software/gv/`
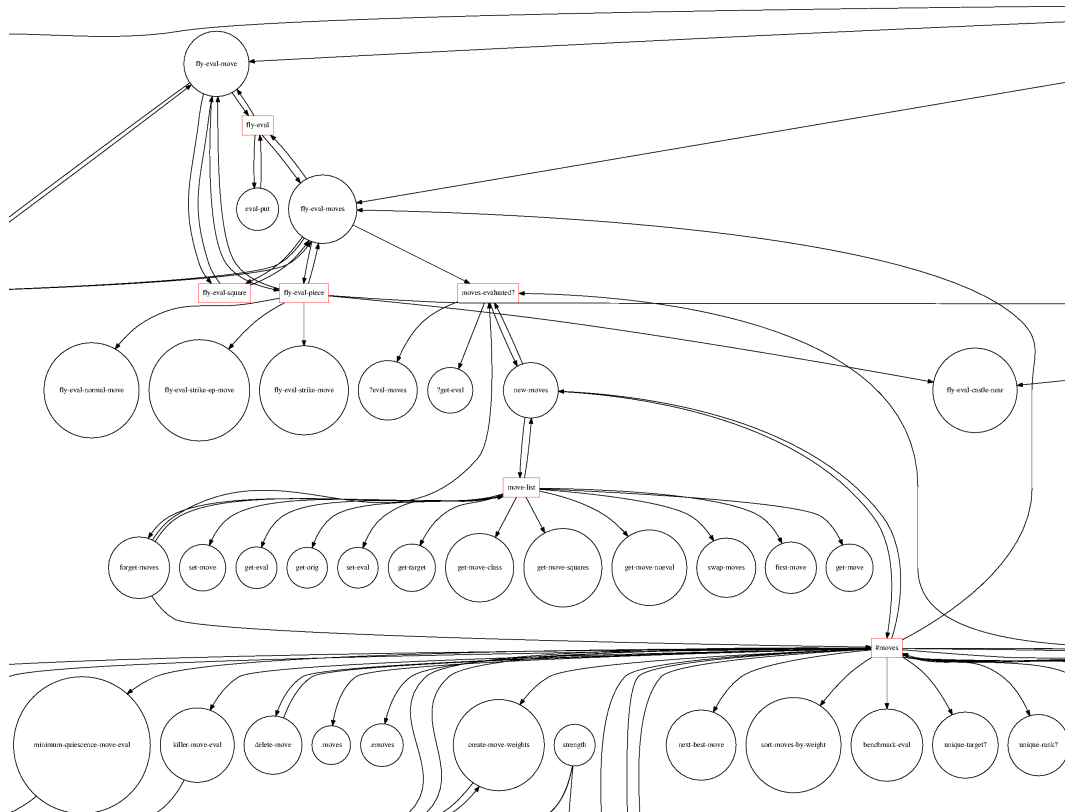
Figure 4.9: Snapshot of the memory access of Brainless

```
: test2 12 12 + 24 = if ." dubi dubi du" endif ;
: test 123 >r 1 2 3 + + 0<> if test2 else ." zero!" endif rdrop ;
```
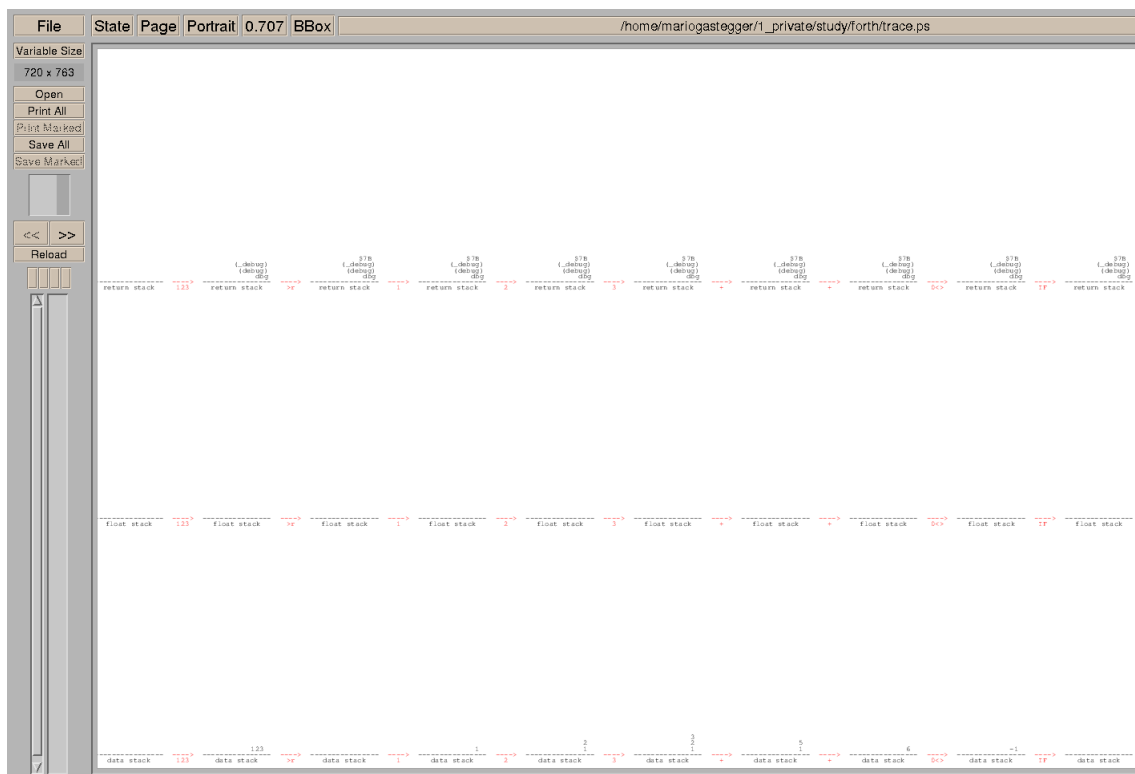
Figure 4.10: The source code of test

Figure 4.11: Output of trace.ps after typing *dbg test*(part 1/2)
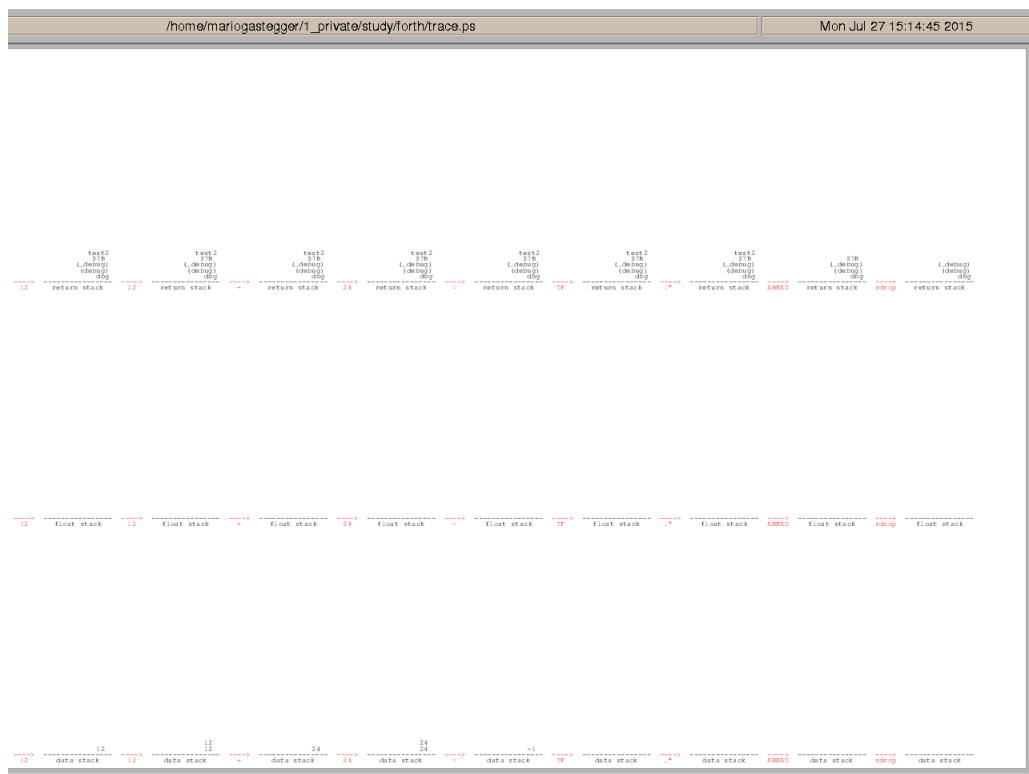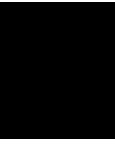
Figure 4.12: Output of trace.ps after typing *dbg test*(part 2/2)

Summary and future work

Hierarchic edge bundle

The hierarchic edge bundle could aid in program understanding, if the suggested improvements are implemented.

Information murals and massive sequence view

The massive sequence view can aid much in program understanding, especially with the suggested improvements. It is well applicable to Forth.

High-level polymetric views

But the key is still expressive naming.

Gfvis

Gfvis, the intended improvement to *dbg*, is only usable for fairly short execution scenarios and thus not as helpful as assumed.

5.1   comparison with related work

Quite some research has been done on the topic of program comprehension in the last decades, but most of it addressed object oriented or procedural languages. Since there is no standard way to model object orientation in Gforth/Forth, it is not possible to implement a general tool for visualization, similar to the existing methods. But these methods should be applicable in general.

The research on trace visualization has proven to be very similar for concatenative languages since a program is also a concatenation of words.

The closest thing to an development environment which provides interactive information from static analysis, is kgforth. It is an integrated development environment for Gforth/Forth, which provides separate windows for debugging and dump output. But its development has been discontinued.

5.2   discussion of open issues

Concerning the implemented demo, most obvious is the lack of usability. First of all the second window turned out to be rather annoying, it would have been better to include the visualization within the Gforth window and record the trace in a standardized data structure in a separate file. Another not yet addressed problem is the scalability of the view, the inefficient use of the screen real estate as well as the long rendering delay of postscript makes it unusable for very long traces. A possible solution would be to limit the displayed stack depth to a certain number since it is not encouraged to manipulate more than some of the uppermost stack elements. Or to limit the depth per word to the number of elements, which have been defined in the stack effect comment.

Another improvement would be the implementation of an interactive trace sequence view, like the "massive sequence" view implemented by [CZH+08], where in addition nested words as well as the stack state between arbitrary words can be hidden and displayed on demand. The introduction of "watch points" to reduce the visual noise and the size of the trace file could also address the scalability issue. But in general, postscript is probably not the best technology for displaying this data.

It is also not practical to compare two traces to each other. A visualization of several traces in a massive sequence view like manner, where the are synchronized by words and differences between them are visualized by colors, could solve this problem.

Due to the implementation of the trace recording within the debugger, it is not possible to collect traces of live systems. Neither is performance analysis possible while recording traces/debugging.

5.3   further work to be done

The pure exploratory approach did not provide any information on the actual impact of the implemented and suggested methods. The gathering of quantitative data and the formulation of hypotheses remains to be done in future works.

Another important question yet to be answered, is whether software maintenance in concatenative languages is conducted similar as in other paradigms. Only with accurate knowledge about how tasks are done, better methods to improve those can be developed.

Furthermore, it is yet to be determined how to orchestrate various methods, to fit in the development process. Whether or not some kind of integrated development environment or an independent set of tools is actually more helpful.

Concerning a Gforth IDE, a Light Table[1] like approach and an application of the idea of continuous programm understanding[MJS+00] would be interesting to see. By defining test input for words, the IDE would display the data flow/stack changes and respond immediately to changes.

5.4   further reading

There hasn't been done any work on program comprehension of concatenative languages, but [CDPC11] and [Cor09] provide good overview of the field field of program comprehension in general.

---

[1]Light Table is a javascript IDE, for further information see http://lighttable.com/

Hierarchic edge bundle source

graphics/hierarchic_edge_bundle-dir_file_word.pgf:

Massive sequence view source

massive_sequence_view-dir_file_word.pgf:

APPENDIX **C**

Gfviz postscript

The contents...

Bibliography

[AKG⁺10] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 53–62, New York, NY, USA, 2010. ACM.

[Bal99] Thoms Ball. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, October 1999.

[Bas97] Victor R. Basili. Evolving and packaging reading technologies. *J. Syst. Softw.*, 38(1):3–12, July 1997.

[Boe76] B. W. Boehm. Software engineering. *IEEE Trans. Comput.*, 25(12):1226–1241, December 1976.

[BW09] Federico Biancuzzi and Shane Warden. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. O'Reilly Media, Inc., 1st edition, 2009.

[CDPC11] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Commun. ACM*, 54(4):142–151, April 2011.

[CHLW06] Stephen Cook, Rachel Harrison, Meir M. Lehman, and Paul Wernick. Evolution in software systems: Foundations of the spe classification scheme: Research articles. *J. Softw. Maint. Evol.*, 18(1):1–35, January 2006.

[Cor09] Bas Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Wohrmann Print Service, 2009.

[CZH⁺08] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Syst. Softw.*, 81(12):2252–2268, December 2008.

[CZvD+09] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Softw. Eng.*, 35(5):684–702, September 2009.

[DLB04] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, CSMR '04, pages 309–, Washington, DC, USA, 2004. IEEE Computer Society.

[Fur86] G. W. Furnas. Generalized fisheye views. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '86, pages 16–23, New York, NY, USA, 1986. ACM.

[Hol06] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, September 2006.

[JS98] Dean F. Jerding and John T. Stasko. The information mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):257–271, July 1998.

[KG88] Michael F. Kleyn and Paul C. Gingrich. Graphtrace—understanding object-oriented systems using concurrently animated views. *SIGPLAN Not.*, 23(11):191–205, January 1988.

[KM96] K. Koskimies and H. Mossenbock. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. Technical report, 1996.

[LR03] Meir M. Lehman and Juan F. Ramil. Software evolution: Background, theory, practice. *Inf. Process. Lett.*, 88(1-2):33–44, October 2003.

[MJS+00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 47–60, New York, NY, USA, 2000. ACM.

[PLVW98] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *In Proceedings Conference on Object-Oriented Technologies and Systems (COOTS âĂŹ98*, pages 219–234, 1998.

[RCM04] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, December 2004.

[Rei95] S. P. Reiss. *Visualization for Software Engineering – Programming Environments.* 1995.

[SB94]   Manojit Sarkar and Marc H. Brown. Graphical fisheye views. *Commun. ACM*, 37(12):73–83, December 1994.

[SFM99]   M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *J. Syst. Softw.*, 44(3):171–185, January 1999.

[SLea97]   Janice Singer, Timothy C. Lethbridge, and et al. An examination of software engineering work practices, 1997.

[SM96]   Margaret-Anne D. Storey and Hausi A. Müller. Graph layout adjustment strategies. In *Proceedings of the Symposium on Graph Drawing*, GD '95, pages 487–499, London, UK, UK, 1996. Springer-Verlag.

[SWFM97]   M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Mueller. On integrating visualization techniques for effective software exploration. In *Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*, INFOVIS '97, pages 38–, Washington, DC, USA, 1997. IEEE Computer Society.

[SWM97]   M.-A. D. Storey, K. Wong, and H. A. Muller. How do program understanding tools affect how programmers understand programs. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, WCRE '97, pages 12–, Washington, DC, USA, 1997. IEEE Computer Society.