FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Titel der Arbeit

Optionaler Untertitel der Arbeit

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Pretitle Forename Surname Posttitle
Matrikelnummer 0123456

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Pretitle Forename Surname Posttitle
Mitwirkung: Pretitle Forename Surname Posttitle
Pretitle Forename Surname Posttitle
Pretitle Forename Surname Posttitle

Wien, 1. Jänner 2001

Forename Surname	Forename Surname

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Title of the Thesis

Optional Subtitle of the Thesis

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Pretitle Forename Surname Posttitle
Registration Number 0123456

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Pretitle Forename Surname Posttitle
Assistance: Pretitle Forename Surname Posttitle
Pretitle Forename Surname Posttitle
Pretitle Forename Surname Posttitle

Vienna, 1st January, 2001

Forename Surname

Forename Surname

Erklärung zur Verfassung der Arbeit

Pretitle Forename Surname Posttitle
Address

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

Forename Surname

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

CHAPTER 1

Introduction

1.1 Motivation

- OK software, due to its(steady growing) complexity [LB85](need to read)
  maybe better than the lehman85 cus available [LR03]
  structured programming http://dl.acm.org/citation.cfm?id=1243380

- OK maintenance [LS80] [ISO06]

Software under lies a continuous changes, throughout its live cycle. The evolution process from the beginning of development until its release and maintenance. Large software[1] and most of all software classified as type E [CHLW06] gets more complex over time. If there are more than a few developers/development teams are involved or the developers/development teams are spread allover the world, there exists more foreign code than self written.

Since changes, enhancements or fixes of existing code demand the developers involved to gain a high level of understanding for the software at hand[Boe76][SLea97]. Due to [CZvD+09] "... up to 60% of the maintenance effort is spent on gaining a sufficient understanding of the program ...". This task is referred to by the scientific community as "program understanding" or "program comprehension" and thus these words are considered synonym in this thesis. This thesis addresses the task of improving program comprehension of the concatenative programming language forth on several level.

- OK mental model(LaToza et al., 2006)
  read: @inproceedingsLieberman:1995:BGC:223904.223969, author = Lieberman, Henry and Fry, Christopher, title = Bridging the Gulf Between Code and Behavior

[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]

in Programming, booktitle = Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, series = CHI '95, year = 1995, isbn = 0-201-84705-1, location = Denver, Colorado, USA, pages = 480–486, numpages = 7, url = http://dx.doi.org/10.1145/223904.223969, doi = 10.1145/223904.223969, acmid = 223969, publisher = ACM Press/Addison-Wesley Publishing Co., address = New York, NY, USA,

- OK strategies as stated by [SFM99]
- OK dynamic analysis as defined by [Bal99] [CZvD+09]
- OK static analysis as defined by [Bal99]

Namely the the reading of source code, static analysis, dynamic analysis and the assistance of writing readable and easy to understand source code.

- OK concatenative languages -> forth, postscript, factor -> implications from the concatenative nature... ie potential to be more natural to read cause of reverse polish notation
  David Shepherd , Lori Pollock , K. Vijay-Shanker, Case study: supplementing program analysis with natural language analysis to improve a reverse engineering task, Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, p.49-54, June 13-14, 2007, San Diego, California, USA
- OK comparison to oo langs
- OK higher abstraction, hard structure boundaries
- OK paradigm promotes a single shared data structure of high importance and thus may simplify the task of putting all the necessary run-time information visually together(cite someone who says that its important to have all information visible at every point in time). Although there are several stacks, features like arbitrary memory allocation, the focus on stacks is clearly stated.

Due to the nature of concatenative languages, it is possible to write source code which ready 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 object oriented languages. Since forth directly operates only on stacks and memory, the information which is immediately needed to follow program execution is limited to those structures. In contrast, in object oriented languages there is also object state, object life cycle and concurrency of interest.

Darren C. Atkinson , William G. Griswold, The design of whole-program analysis tools, Proceedings of the 18th international conference on Software engineering, p.16-27, March 25-29, 1996, Berlin, Germany

1.2 problem statement (which problem should be solved?)

There is plenty of work done on the task of program comprehension in object oriented and procedural languages[CZvD+09], but nearly none on concatenative languages. The qualitative exploratory approach of this thesis does not encourage the formulation of specific hypothesis. Therefore the first question, is the applicability of existing methods and their visualization techniques. The second question to be answered, concerns new approaches, which may be exclusive to concatenative languages or gforth/forth.

1.3 aim of the work

TODO: maybe cut scope down to dynamic analysis or trace visualization

~~This work aims to better understand how program comprehension is performed in concatenative languages and how it can be made more efficient.~~ The secondary goal is the analysis of the applicability of existing analysis- and visualization methods and to provide modifications to existing visualization methods(and maybe suggestion of new methods). The forth programming language is used as a representative of concatenative languages.

1.4 structure of the work

At first, the available information of a forth program is identified. The next step is to characterize the information and its necessity for program comprehension is investigated. The differences of forth and object oriented languages are summarized and then the applicability of existing analysis and visualization methods is presented. Since there is no standard implementation of object orientation if forth, this thesis won't take any object orientation implementation into account. The last part of this thesis investigates probable enhancements and modifications to existing methods and proposes new approaches. After the conclusion, the thesis presents further suggestions to support program comprehension and further topics of research in this direction.

//TODO: move to methodology

- qualitative approach
- exploratory case study
- prototype
- sketches
- trying to understand programs developed withing stackbasierte programmierung vl?
- qualitative approach , exploratory approach(?)

- proposal
- Preliminary evaluations as defined by [CZvD+09]
- outcome is a subjectiv view of the available methods, and proposed enhancements which have been implementet
- case study of the implemented enhancement
- suggestions of further enhancements

CHAPTER 2

Analysis of existing approaches

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

2.1 program comprehension

Program comprehension can be gained through several approaches. First of all is 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 to read first, the reading the 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. The most important issues stated, 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 properly is not properly addressed in education. The syntactical flexibility, forth provides compared to other languages(and paradigms), allows 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.

Besides the actual reading, there emerged several strategies on how to understand a program[SFM99]:
evt hier nur auf
citeStorey:1999:CDE:308936.308940 oder
citeStorey:1997:PUT:832304.836998 verweisen?!

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 building high levels 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.[SFM99] points out, there are certain factors which influence the choice programmers take. Thus the programming environment should provide methods to support all the strategies.(See later section)

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

2.2 Analysis to support program understanding

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

2.2.1 Dynamic analysis

Dynamic analysis is performed on the image of a program, executed on a real or virtual processor. The advantage dynamic analysis is, due to the availability of the data to be

manipulated, that the actual behavior of a program can be investigated. The major drawback here is that there exists only an incomplete view of the software at hand[Bal99]. It is a very efficient way to evolve or correct the mental model of developers. I will focus on distinct here between two categories of dynamic analysis: TODO... ???

- interactive
  The characteristics of interactive(stepped debugging, interpretative execution) approaches, is that the program is still executing during the analysis and thus time is a critical factor. This makes some analysis methods like performance analysis impossible. In networked environment, timeouts can make interactive analysis very hard.

- non interactive(real-time and post-mortem)
  In real-time analysis(log analysis, trace analysis), data changes very quickly, which can make the task of tracking execution scenarios very resource intensive. In contrast, in post-mortem analysis(log analysis, trace analysis), the processing of very large files becomes a problem.

2.2.2 Static analysis

Although this thesis focuses 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.

2.3 Existing approaches

2.3.1 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 as well as code which support program understanding and support maintenance.
For displaying data, the most important words are *.*, *.s*, *."*, *type* and *dump*.
. and *.s* simply print out elements of the stack, there are also words to vizualize the other stacks. The words *."* and *type* print out text. These can be utilized to print logging information.
Last but not least, *dump* displays memory areas(address, hex and ascii).

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

~~ prints the location of itself in the source-file(file and line number) as well as the data stack.
These 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 contained in gforth(since 0.7.0), it opens a separate xterm window, displaying the current number base, the float stack, the data stack and the current search order. The view is updated after each execution in the interpreter. Thus it is only useful for interactive analysis.

Stepping debugger

dbg[2], 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. The debugger is only usable for interactive analysis.

Assertions

Assertions[3] 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. With the use of assertions, the code can be prevented from breaking during maintenance. The word *assert(* starts the assertion, the following words until the *)* is executed and has to leave a flag on the stack. If the flag is true, the asserted condition is meet, if false, the execution of the program ends and the location of the failed assertion is printed(source file and line number).
There are several assertion levels and by setting the *assert-level*, assertions can be deactivated.

Documentation

Thorough and up to date documentation is undoubtedly important, this also applies to concatenative languages. Besides word documentation, since forth is by default a untyped language, there is special kind of documentation encouraged to ease the understanding of words namely the stack effect comment[4]. It is written next to the name of the defined

[2]For further information see: `https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Debugging.html#Debugging`
[3]See `https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Assertions.html#Assertions`
[4]See `https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Notation.html#Notation`

word and contains the number and kind 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.There is also a distinction to be made between interpretation, compile and run-time behavior ...dont mention it!?.

Words and word lists

Like in any other language, words should be named expressive. However sometimes, it may not be avoidable or desirable to reuse names. Forth provides an elegant mechanism, called word lists[5], to account 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 project it may be necessary to define a naming strategy.

Factoring

As in any other programming language, to solve complex problems, it is necessary to split the overall problem down into 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(`https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Factoring-Tutorial.html`).

Aliasing

In gforth, there can be multiple aliases[6] for a 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 help features[7] to ease the writing of forth. Most notable, related to program understanding, are:

- Word documentation lookup
- Jump to line from, error messages, debug output and failed assertions. ~~ output
- Highlighting
- Indention handling

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

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

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

Kgforth

There has also been an afford to integrate some of those tools into an graphical development environment. The project is called Kgforth[8], but its development seems to be discontinued.

2.4 Applicability to concatenative languages

In this section I will present a hand full of visualization methods and will discuss the applicability to gforth/forth. In the following chapter I will present an example on the most promising methods and analyze the actual outcome regarding program understanding.

existing methods abstract(abstract like print debugging and stepping and so on) furthermore the abstraction of all those methods mentioned above to find similarities and then adapt them to fit the characteristics of concatenative languages.

Interaction diagram, 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 concurrency, th word-only level provides little help. On this level, the actors would be represented by words and the messages[9] would represent word executions. Therefore the sequence diagram would degrade to a syntax tree. On the system level although, the sequence diagram can provide useful information on the interaction between system components. TODO drop scenario diagram???
The scenario diagram as implemented by
cite Koskimies:1996:SUS:871313, is essentially the same as the sequence diagram and the interaction diagram(Jacobson, 1992) which has been proposed by.
Since the system level visualization applies to almost any paradigm and the word level don't seem promising, I will not investigate the benefits of the UML sequence diagram 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 the hierarchic relations. Like in object oriented systems, the proper organization of directories, files and word definitions has to be considered to make such a visualization helpful. Another possible mapping could depend on word list as words hierarchic and word execution as non hierarchic relation.

[8] `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.

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

In the following chapter I will analyze both possibilities using a real forth software as an example.

Information murals and massive sequence view

The information mural was initially proposed by [JS98]. A modified version, mass sequence view proposed by [Cor09], which turns the horizontal scrolling into vertical scrolling and also contains the hierarchic aspect could be more suitable. As with the hierarchical edge bundles, the usefulness of the hierarchical component depends highly on the organization of the software at hand.
In the following chapter I will analyze the massive sequence view using a real forth software as an example.

High-Level polymetric views

Polymetric views[DLB04] are a very interesting and 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, 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 more 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 required. Large forth programs tend to consist of no more 1000 words[cite], thus the complexity and visual size of polymetric views can be kept at a moderate level. ... citation needed!
In the following chapter I will analyze forms of polymetric views of real software 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 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. With growing distance lowers the degree of interest.
This method is related to polymetric views, as mentioned above.

Execution pattern view

The execution pattern view proposed by [PLVW98] to visualize large traces in a scalable manner and helps to identify execution patterns, represent 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. I 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 a tennis and football and refer to the multiple perspectives necessary to understand all aspects of a match and the whole outcome.

In the following chapter I will create a variable access graph inspired by the tree like visualization of GraphTrace. TODO: Glossry entry for GraphTrace....

- frequency spectrum analysis (Ball 1999)
- architecture oriented visualization (Sefika et al. 1996)
- architecture with dynamic information (Walker et al. 1998)

CHAPTER 3

Methodology

describe the actual flow of work

describe what what the goal is and how its completion is verified

ich schaue was es gibt und versuche es auf forth anzuwenden und probiere es exemplarisch aus, was raus gekommen ist und mutmaÃ§e die hilfreichigkeit

CHAPTER 4

Results

To properly approach the stated problem, the first thing is to understand if and when program understanding is required. Although intuitively obvious, this section will discuss both, since the approach which developers use to understand programs, can be very different during the life cycle of software. Next, the means of understanding programs itself is investigated. Third, the nature of concatenative languages and in particular gforth/forth will be investigated.

kind of an ~~ide~~ development environment
light table ide(js) continuous reverse engineering idea of [MJS+00] to provide immediate response of the systems output... although probably not applicable or very time consuming in setup(or not more than integration testing...) for most industrial scale software
eclipse ide(java)

4.1 suggested solution

A very important question in this concern is, how can developers be assisted to write readable code. Experienced developers may do that intuitively, but how can novice developers be encouraged and supported to write readable code. Concatenative languages are flexible enough to produce code very similar to natural languages, but how can this attribute be supported?
One answer is to provide hints based on static analysis.
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", this is hardly avoidable at the lowest level. Thus, proper documentation of words is essential. Its pretty The obvious, that stack effect comment[1] in forth, are a must have, but also the behavior of the word should be

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

explained if complex or not very natural to read words[2]. Another advantage of word definition comments is the possibility of automated documentation generation.
Very long word definitions tend increase the amount of brain capacity required to understand its behavior. The way to account this problem is to break down the overall task into manageable pieces. It is called factoring[3] in context of concatenative languages. An approach could be to place a hint on word definitions which exceed a certain amount of lines or words or different words and suggest further factoring.
Another tool to make code read more natural, is aliasing[4]. By defining aliases for a certain word, its functionality can be used in different contexts and still read very natural.
Expressive naming, although obvious, it should be mentioned that assigning expressive and fitting names for words is essential. This applies to any language[does it? cite...].
To understand code, the systematic approach turned out to be most efficient[RCM04].
To ease afford of finding the definition of words used at a certain point, a hyperlink like referencing mechanism can be used[cite the visualization paper with the hyperlink feature].
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 larger, the amount of words can grow big. 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 reverence to where they are used. Auto completion can also help a lot in finding words previously defined.

- other data structures and variables should be displayed
  - memory maybe like [Rei95] or [AKG+10] but since there is no underlying object orientation and no standardized oo system this would be hard do accomplish
  - fisheye or word cloud like display(tree or sugiyama as of [SWFM97])
- interactive program manipulation: state of the system before a word, after a word and by clicking on the word jumping to its definition or inserting it and there also providing those features
- stepping debugger mode: simply stepping through the whole code word by word
- goal-oriented strategy: the definition of an execution scenario such that only the parts of interest of the software system are analyzed (Koenemann and Robertson, 1991; Zaidman, 2006).
- code analysis and visualization facilities see chapter 2 TODO

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

[3] See `https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Factoring-Tutorial.html#Factoring-Tutorial`

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

4.1.1 software evolution

[LB85] and [LR03] (beide vllt kritisch zu betrachten und evt out of scope; wenn dann noch in den jÃijngeren citedbys schauen; die grund aussage hier kÃűnnte sein, dass E-Type software immer im wandel befinden wird und immer Ãďnderungen unterliegen wird()aus leh2003))

4.1.2 software maintenance

- types of maintenance
- find bugs and fix them
- find the right place to implement a new feature.
- find the right place to modify a feature.

4.1.3 program comprehension

- structured approach
- thorough reading is the most efficient[cite]
- about the mental model building
- keeping the mental model up to date
- keeping artifacts up to date

4.1.4 program comprehension strategies

- top down
- bottom up
- knowledgebased
- systematic and as-needed
- integrated approaches

4.1.5 analysis to support program understanding

Several analysis types

- source code reading
- documentation reading(everything except source code)
- static analysis
- dynamic analysis
- post mortem analysis
- realtime analysis

dynamic analysis

- about realtime/interactive vs post mortem
- actual behavior
- incomplete view [Bal99]
- observer effect
  Andrews, J. (1997). Testing using log file analysis: tools, methods, and issues. In Proc. International Conference on Automated Software Engineering (ASE), pages 157âĂŞ 166. IEEE Computer Society Press
- scalability
  Zaidman, A. (2006). Scalability Solutions for Program Comprehension through Dynamic Analysis. PhD thesis, University of Antwerp
- debugging -> different kind of paradigms and languages and tools
  see @incollectionreiss1993trace, title=Trace-based debugging, author=Reiss, Steven P, booktitle=Automated and Algorithmic Debugging, pages=305–314, year=1993, publisher=Springer
- about debugging
- dataflow analysis(Backward Analysis)(not sufficient in demo)
  Darren C. Atkinson , William G. Griswold, Implementation Techniques for Efficient Data-Flow Analysis of Large Programs, Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), p.52, November 07-09, 2001

static analysis

- OK complete view
- OK no actual data present
- OK architecture and design documents

Static analysis is ...[cite]... not running 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 is no mean of covering the actual data and follow its manipulation. ...[cite] This makes it a most valuable tool for architecture, design, and algorithm analysis. ...[cite]

4.1.6 applicability to concatenative languages

existing methods abstract(abstract like print debugging and stepping and so on) furthermore the abstraction of all those methods mentioned above to find similarities and then adapt them to fit the characteristics of concatenative languages. applicability for concatenative languages

4.2 visualization to support program understanding maybe some examples(and tools)

- sequence diagram
- circular diagram and interactive interaction sequance diagram [Cor09]
- interaction diagrams (Jacobson, 1992)/ scenario diagrams (Koskimies and MÃűssenbÃűck 1996)
- information murals (Jerding and Stasko, 1998)
- polymetric views (Ducasse et al., 2004)
- fisheye views (suggested by George W. Furnas, 1986, and formulated by [SM96] and [SB94])
- hierarchical edge bundling (Holten, 2006)
- structural and behavioral views of object-oriented program (Kleyn and Gingrich, 1988)
- matrix visualization and âĂIJexecution patternâĂİ notations [PLVW98] to visualize traces in a scalable manner(De Pauw et al. 1993, 1994, 1998)
- architecture oriented visualization (Sefika et al. 1996)
- a continuous sequence diagram, and the âĂIJinformation muralâĂİ (Jerding and Stasko, 1998)
- architecture with dynamic information (Walker et al. 1998)
- frequency spectrum analysis (Ball 1999)

4.3 comparison and summary of existing approaches

existing approaches for gforth/forth and relation to above mentioned stuff

- kgforth http://sourceforge.net/projects/kgforth/
- existing methods(actual methods)
  - factoring (http://en.wikipedia.org/wiki/Modular_programming https://www.complang.tuwien.ac
    html/Factoring-Tutorial.html http://www.ultratechnology.com/Forth-factors.htm)
  - aliasing
  - organization of word lists
  - source code documentation
  - other documentation artifacts
  - dump
  - ., / and type
  - dbg
  - see and code-see
  - ~~

4.4 implementation

- OK proof of concept by enhancement of stepping debugger on forth code level(cause it has turned out to be the fastest and simples approach) by showing additional data: the other stacks

As a first step, in addition to this work, the stepping debugger of gforth was enhanced. The existing implementation only shows the data stack and the word to be executed. The enhanced version is now able to record the debugging trace as a postscript(trace.ps) file which shows the executed words and the state of the data stack, the float stack and the return stack. Since the return stack contains addresses related to the executed words, the word names were displayed when possible. The algorithm to resolve the word names corresponding to the address was already implemented in gforth's back-trace and not developed by myself. The debugging trace is stored as a postscript file and can be investigated during the debugging session as well as after the actual execution is complete. The implementation involved the modification of some gforth internal files as well as a file(gfvis.fs) which has to be loaded with gforth. The display layout was implemented in postscript. Every trace-file is constructed from a template file(gfvis.ps) which contains the postscript code to layout the recorded trace. (Since the debugger only works with the itc engine, the debugging has still to be performed with this engine.????) There was also made an attempt to accomplish this on the c source code level, but it turned out to be rather complicated and therefore the forth only level was chosen.

CHAPTER 5

Summary and future work

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

summary of what has been done and the subjective conclusion

CHAPTER 6

Critical reflection

only on the implementation or also the suggestions?

6.1 comparison with related work

Quiet 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, it is not possible to implement a general tool for visualization similar to the existing methods, but these methods should be applicable on an object oriented model implemented in forth 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 most similar existing work is kgforth, a development environment using qt. Although the outline looks promising, it is not in development anymore and I was not yet able to run it.

6.2 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 makes it unusable for very long traces. A possible solution would be to limit the displayed float and data stack depth to a certain number since it is not encouraged to manipulate more than some of the most upper stack elements. Or limit the depth per word to the number of elements as 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 the size of trace file could also address the scale issue. It is also not practical to compare two traces to each other and it is very hard to understand the behavior of large systems. [A visualization of several traces like the massive sequence view where several traces are synchronized by word and differences are visualizes by colors could solve this problem.???might be the same as on long trace for server like software??]

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.

- nature of gforth
  - interpretation/compilation mix(how to integrate the adhook changes between modes '[]')
  - implementation within the executing system(??)
  - lack of dynamic information(return stack add -> wordname heuristic)

6.3 further work to be done

- how does software maintenance work in those languages?
- ide
- display of variable content
- display of allocated memory areas
- display of color diff with tooltip of previous values for stacks and memory areas
- (better visualization of loops and control structures) is this even possible?
- (display of the full program as a graph) is this even possible?
- (customizable inspection depth) ?
- type system for forth...see strongforth and the work of greg
- static code analysis

  v—- see previous chapter —-v
- using a standard data type to store traces
- stack depth per word

conclusion like what i contributed to the community!!

6.4 further reading

work on program comprehension of concatenative languages good overview of the field [CDPC11] and [Cor09]

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.

[ISO06] ISO. Software engineering – software life cycle processes – maintenance. ISO 14764:2006, International Organization for Standardization, Geneva, Switzerland, 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.

[LB85] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

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

[LS80] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[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.

Glossary

Charles D. Moore Charles D. Moore, is the inventor of the forth programming language.. 14