What should a Programming Environment for Threaded Interpretive Languages provide?

by

Mikael R.K. Patel

The Department of Computer and Information Science
Linköping University

PhD theses:

* Available at: University of Microfilms Intl., 300 N. Zeeb Road, Ann Arbor, MI 48106, USA.

( Linköping Studies in Science and Technology. Dissertations. )

No 97 Andrzej Lingas: Advances in Minimum Weight Triangulation, 1983.

Licentiate of engineering Theses:

( Linköping Studies in Science and Technology. Theses. )

No 73 Ola Strömfor: A Structure Editor for Documents and Programs. 1986.
No 108 Rober Bilos: Incremental Scanning and Token-based Editing. 1987
No 113 Ralph Rönquist: Network and Lattice Based Approaches to the Representation of Knowledge. 1987.
No 118 Mariam Kamkar, Nahid Shahmehri: Affect-Chaining in Program Flow Analysis Applied to Queries of Programs. 1987
No 126 Dan Strömberg: Transfer and Distribution of Application Programs. 1987
No 139 Christen Bäckström: Reasoning about Interdependent Actions. 1988.
What should a Programming Environment for Threaded Interpretive Languages provide?

by

Mikael R.K. Patel
E-mail: mip@liu.ida.se

Abstract: Given the low cost compiler and interpreter of Threaded Interpretive Languages, such as FORTH, an incremental and interactive programming style is more or less directly achieved. Traditionally program development may be regarded as a successive definition and testing of procedures and data structures. To reduce the complexity of the programming task and enhance reusage of code modules additional tools are required. This paper discusses some tools that should be available in a programming environment for Treaded Interpretive Languages. Some of the basic ideas from the advanced programming environments of INTERLISP and Smalltalk-80 are related and discussed.

This research is supported by the Teaching Faculty at the Department of Computer and Information Science, Linköping University.

Presented at the
8th Rochester Forth Conference on Programming Environments,

Postadress:
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

Mailing address:
Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden
INTRODUCTION

Threaded Interpretive Languages such as FORTH are powerful programming languages due to the fact that they allow an interactive and incremental style of programming. These two attributes allow programs to be tested almost immediately without the writing of specific test code. With the removal of traditional parsing, i.e., syntax and semantic checking, and relying on the reverse polish notation an extremely low compilation cost with regard to time and resources is possible. Compilation is, in principle, reduced to token scanning and symbol table searches. But the productivity of this environment relies on the incremental testing of individual definitions in a bottom-up fashion. The testing and development of larger systems is not directly supported.

This paper is a discussion of higher level support of the debugging phase when developing large software system with emphasis on reuseage of program modules. Debugging tools for static and dynamic analysis are discussed and exemplified with a prototype system that implementing some of the proposed methods. The distinction between static and dynamic analysis is here made based upon the properties of a program when analysing its definition and behaviour. Many of the methods and tools presented already exist in other programming environments such as Smalltalk-80 [Goldberg and Robson 1983] and INTERLISP [Teitelman 1978].

1. STATIC DEBUGGING TOOLS

Debugging a software system is a hard task, even for the experienced programmer. Given the problem of debugging a large software system which may be composed of a number of modules written by a number of programmers the complexity of the task grows rapidly. When composing a number of modules problems often occur on the interfaces between modules. New types of errors arise due to the fact that the number of components have grown, and that a detailed understanding of the interrelationships between the components is missing. In general it is difficult for humans to keep an overall view when dealing with to many interacting components.

The kind of support a programmer may need in these situation are typically information regarding different properties of the code the programmer is using. Tools such as Code Analyzers and Documentation Support are useful to increase the understanding of program modules and their interactions.

Often documentation of programs are on the level of functionality (and implementation) and not on the level of structure. Structural information is essential in order to understand the dependencies that may occur between sub-parts within a program module, e.g., between the inner definitions. Questions like am I using the definitions in the right order or how do I use this module may occur.
At this point a comparison with other engineering disciplines reveals that software engineering lack standardized methods for describing this kind of information. House architectures and hardware engineers deal mainly with the structure of the design and its functionality is often implicit or may be derived for the structural description.

In the following sections some tools for Code Analysis and Documentation Support are discussed in more detail.

1.1 Code Analysis
Let us now step back and investigate the scope of Code Analysis that would be interesting to be able to perform. Code Analysis may be regarded as viewing a set of definition, i.e., a program module, as a data base, and the analysis of it as different views of the data base. In principle, the threading of code creates a hierarchical data base of definition. What operations would be of interest to perform on this data base other than execution?

1.1.1 Decompilation
On form of Code Analysis is the regeneration of the source code, i.e., decompilation. This is maybe considered to be the simplest form of Code Analysis; To read the definition of a function, thus view its inner definition and "see" if the implementation meets its specification. The motivation to decompilation are then 1) the source code may not be accessible or 2) it may take too long time to retrieve compared to decompilation, and 3) the object code may differ from the source code because of transformations, optimization, and compilation.

1.1.2 Cross References
Another form of Code Analysis is to generate information about the usage of symbols, i.e., cross reference lists. These may be generated from the source code or from the object code (the data base). For many compiling languages such as C [Kernighan and Ritchie 1978] and Pascal [Jensen and Wirth 1985] this type of facility is often available. In an incremental environment the notion of source code and object code is not as distinct. Therefore the generation of this type of information is more appropriate from the object code level.

1.1.3 Call Structures
Yet another form of Code Analysis is the generation of the calling structure of a definition. This information may be presented in a flowchart form, with the definitions which are access and if appropriate their calling structure, i.e., the flow of control. Normally this type of information is both difficult and time consuming to generate and
keep consistent with the implementation, i.e. the code, thus tools to automatically generate this information either when needed or as a part of the documentation of a module.

Generation of a Call Structure Tree, e.g., flowchart, may easily be achieved by traversing the internal structure of the definition. The process become very similar to decompilation (unthreading).

1.1.4 General Code Analysis
In general there are three major questions that may be asked and answered by Code Analysis at this level;

- USAGE: Which other definitions are used to implement this definition?
- STRUCTURE: And in which control structure (flowchart) form?
- USES: Which other definitions use and are thus dependent on this definition?

INTERLISP provides these types of facility through a software package called MASTERSCOPE which builds a relational data base of the code. The programming environment of Smalltalk-80 also provides these possibilities except for the generation of control structure overviews.

1.2 Documentation Support
In general a software package usability may be considered to be a function of its documentation and functionality when compared to other software packages solving the same or sub-parts of the same problem. A software package which lacks adequate documentation does not really exist even if it performs a specific task faster or better than any other solution. As we all know documentation is maybe one of the most boring types of occupations within the spectrum of tasks when realizing software systems. The question is thus how can programmers be aided so that documentation is easily generated. Some of the suggested tools above for Code Analysis are useful to be able to generate overall information about a large software system. Control flowchart and Calling Structure are valuable sources of information when debugging and maintaining software. Standardization of documentation makes it possible to automatically generate the overall structure of a documentation from, for instance, the source code.

Let us last in this section on tools for static debugging consider some tools for documentation.

First the use of so called shadow screens for the documentation of FORTH source code written using screens has become one of the major accepted methods for attaching documentation on functionality to source code, i.e., definitions.

Secondly the use of index file to support on-line documentation, primary used to answer user questions like how many parameter does a word require etc.
And third the use of *Structure* and *Hypertext Editor* which will allow a programmer to attach a number of attributes to the source code such a documentation, alternative implementations etc, and thus ease the maintenance of the software system.

Many of the questions on how to deal with documentation are even more difficult to answer in a uniform way when considering that many FORTH systems today are used for target and meta compilation. The target machine are often considerably smaller than the host machine, and thus all code and documentation must in most cases be stored on the host.

### 2. Dynamic Debugging Tools

The behaviour of a large piece of software is not easily derived for a glance of the source code. The method of computation may be investigated in minor part but not as a whole. Given a program with "problems" how can we locate the problems and then fix them? In this section some tools for debugging the dynamic property of a program will be discussed.

#### 2.1 Execution Trace

When testing a program the tracing of a definitions execution can give valuable information about its behaviour. There are two major ways of performing this trace. The first where the execution of the inner steps of a definition are visualized and, second, the tracing of the entrance and exit of a usage, i.e., a call. These forms of tracing are normally address as the *white-box* resp. the *black-box* approach to testing. A *black-box* trace generates a somewhat less detail trace but gives information about the entrance and exit of a definition call.

#### 2.2 Break-points

An other form of dynamic debugging aid is the possibility of setting break points, e.g., to stop the execution at a point in execution and enter a command loop where the status of the execution may be analysed. The two main issues when using break points are 1) when to perform the break; always or conditional, and 2) what to be able to do when a break occurs. The first issue, when to perform the break, is easiest implemented if the break is to be performed always. Conditional break points give some problems as the environment of the execution of the break condition have to be determinated. A conditional break point may, for example, be on effect of the number of times a definition is called. The second issues, what to be able to do when a break occurs, is a question of mode. What type of information is relevant and necessary?

We propose the following "command" as a minimum set; first the possibility to *examine the environment* of the call, e.g., the state of the parameter stack, and the caller
of the definition. Second the possibility of simulation of the definition of the broken
definition, which may yet not be defined, e.g., returning after dropping parameters and
depositing resulting values. Third the execution of the definition, with or without, a
return to the break "command" loop. And last an exit out to the normal interpreter so
that simulation of the behaviour is possible.

2.3 Observing
Normally when a programmer starts to analyse the performance of a large software
system tracing and break-points may generate to much information and be confusing.
An alternative to these debugging tools is the idea of observation. By putting a
definition under observation a programmer may learn the context in which definitions
are used, e.g., who calls the definition and when. While traces and break-points are
restricted to definitions, observation may be performed on any type of object in the
dictionary.

Tools for trace, break-points, and for observing object and definitions may easily be
realized in FORTH by temporary changing the code field point. The actual function to
perform the trace, break-point or observation is written in the same fashion as a high
level object creator.

2.4 Measuring execution
When a solution to a programming problem has been implemented some additional
questions may arise. Questions like how can I improve the code? Where is most of the
time spent in the algorithm? Given the rule that approximately 90% of the computation
time is spent in 10% of the code this section of code is interesting to find. Recoding of
this section would give a major improvement on execution speed.

Programming environments like UNIX™ provide tools profiling of programs. A
possible way of realizing this mechanism in FORTH would be to redefine the entry and
exit of definitions (functions) so that a reference counter and a timer may be maintained
at run-time.

2.5 Problems with dynamic debugging in FORTH
A number of programming environments such as INTERLISP and Smalltalk-80 allow
temporary redefinition of a function or procedure at a break-point and then to resume
execution. This type of incremental development of code is especially interesting if the
test case or execution environment is difficult to regenerate.

Realizing this mechanism in a FORTH environment gives some difficulties as the
redefinition must revise the existing definition. One method would be to allow a form of
patching to be performed. This would be similar to deferring (forward definition).
Many of the dynamic debugging tools are easily implemented in FORTH but do not give so much information as FORTH lacks naming and typing of parameters. The interpretation of data on the stack at a break-point is left to the programmer. The use of named parameters as found in a number of FORTH dialect remove some of the problems but not all.

2.6 Multi-task dynamic debugging
Even though debugging of multiple task software system is outside the scope of this short paper the problem within this domain should be recognized. The debugging of a real-time system with several interacting tasks is an even harder problem to find appropriate tools and methods for [Fritzon 1983]. Tracing of the individual instruction streams are not meaningful on this level but the tracing task communication and synchronization via semaphores, monitor, message-passing or other task mechanisms are relevant. A number of aspects on how to deal with this problem domain may be found in, for instance, hardware simulation and design where the idea of probing is a well known concept. In general the validation a programmer may wish to achieve on this level of debugging in on the interaction between tasks and resources.

3. IMPLEMENTATION
A prototype system has been written in FORTH-83 implementing the following of above described methods and tools;

- Black-box tracing of definitions
- Break-points on the entry of definitions
- Observation of constants, variables and definitions
- Decompilation of definitions
- Plotting of call structure flowchart
- Information about which definitions use resp. are used by a definition

The dynamic debugging tools are realized mainly by temporary redefinition of code field pointers while the static debugging tools are performed using tree traversal techniques on the dictionary. A typical question on what a definition needs for its implementation takes only a few seconds even though the question may result in a total traversal of the code in the dictionary.
ACKNOWLEDGEMENTS
A special thanks to my wife Eva, Göran Rydqvist for reading and commenting this paper, and last to all the rest of the members of the CADLAB research group at the Department of Computer and Information Science, Linköping University.

REFERENCES


What should a Programming Environment for Threaded Interpretive Languages provide?

Mikael R.K. Patel

Abstract: Given the low cost compiler and interpreter of Threaded Interpretive Languages, such as FORTH, an incremental and interactive programming style is more or less directly achieved. Traditionally program development may be regarded as a successive definition and testing of procedures and data structures. To reduce the complexity of the programming task and enhance reusage of code modules additional tools are required. This paper discusses some tools that should be available in a programming environment for Treaded Interpretive Languages. Some of the basic ideas from the advanced programming environments of INTERLISP and Smalltalk-80 are related and discussed.

Keywords: Treaded Interpretive Languages, Software Engineering and Programming Environments.
<table>
<thead>
<tr>
<th>Citation</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiTH-IDA-R-88-34</td>
<td>Arne Jönsson, Nils Dahlbäck: Talking to a computer is not like talking to your best friend. Also in Proc. of the First Scandinavian Conference on Artificial Intelligence, March 9-10, 1988, Tromsø, Norway.</td>
<td>Arne Jönsson, Nils Dahlbäck</td>
</tr>
<tr>
<td>LiTH-IDA-R-88-30</td>
<td>Erik Sandewall: Future Developments in Artificial Intelligence. Also in Proc. of European Conference on Artificial Intelligence (ECAI), Munich, August, 1988.</td>
<td>Erik Sandewall</td>
</tr>
<tr>
<td>LiTH-IDA-R-88-23</td>
<td>Johan Fagerström, Yngve Larsson: Two Contributions on Debugging Distributed Systems. Also in Proc. of ACM SIGPLAN/SIGOPS, Workshop on Parallel and Distributed Debugging, Madison, Wisconsin, May 5-6, 1988.</td>
<td>Johan Fagerström, Yngve Larsson</td>
</tr>
<tr>
<td>LiTH-IDA-R-88-19</td>
<td>Lin Padgham, Ralph Rönquist: From a Technical to a Humane Environment: A Software System Supporting Cooperative Work. Also in Proc. of the GDI International Conference on USER INTERFACES, Rüsselikon, Switzerland, October 20-21, 1986.</td>
<td>Lin Padgham, Ralph Rönquist</td>
</tr>
<tr>
<td>LiTH-IDA-R-88-18</td>
<td>Nils Dahlbäck: Mental Models and Text Understanding - a Commented Review.</td>
<td>Nils Dahlbäck</td>
</tr>
<tr>
<td>LiTH-IDA-R-88-16</td>
<td>Lin Padgham: A Model and Representation for Type Information and Its Use in Reasoning with Defaults. Also in Proc. of AAAI'88, American Association for Artificial Intelligence, 1988.</td>
<td>Lin Padgham</td>
</tr>
<tr>
<td>LiTH-IDA-R-88-15</td>
<td>Lin Padgham, Ralph Rönquist: LINCKS: An Imperative Object Oriented System. Also in Proc. of the 20th Annual Hawaii Int. Conf. on System Sciences, Hawaii, 1987.</td>
<td>Lin Padgham, Ralph Rönquist</td>
</tr>
<tr>
<td>LiTH-IDA-R-88-13</td>
<td>Erik Tengvall: Ett kartorienterat planeringssystem för autonoma farkoster, en design diskussion.</td>
<td>Erik Tengvall</td>
</tr>
</tbody>
</table>
organizes undergraduate and graduate studies in Computer Science, Telecommunication and Computer Systems, and Administrative Data Processing. Research activities have an emphasis on advanced software technology and computer systems design and are organized in a number of research laboratories:

- **ACTLAB** - Laboratory for Complexity of Algorithms, which is concerned with the design and analysis of efficient sequential and parallel algorithms, and complexity theory, especially in the areas of computational geometry, data structures on bounded domains and graph algorithms.

- **ASLAB** - Application Systems Laboratory, which studies design of advanced support systems for interactive use of computers, including tools for automated construction of applications software.

- **CADLAB** - Laboratory for Computer-Aided Design of Electronics, which concentrates its research activities around tools for integrated development of hardware and software, graphics-based modelling and simulation techniques.

- **LIBLAB** - Laboratory for Library and Information Science, which studies methods for access to documents and the information contained in the documents, concentrating on catalogs and bibliographic representations, and on the human factors of library use.

- **LOGPRO** - Laboratory for Logic Programming, which concentrates its research activities around foundations of logic programming, relations to other programming paradigms and methodology.

- **NLPLAB** - Natural Language Processing Laboratory, which conducts research related to the development and use of natural language interfaces to computer software.

- **PELAB** - Programming Environments Laboratory, which works with design of tools for software development, specific functions in such tools and theoretical aspects of programs under construction.

- **RKLLAB** - Laboratory for Representation of Knowledge in Logic, which covers issues and techniques such as non-monotonic logic, probabilistic logic, modal logic and truth maintenance algorithms and systems.

**Research Reports 1988**


