High-Level Debugging: facilities and interfaces.

design and development of a debug-oriented IDE

by: NICK PAPOULIAS

Free Software needs, Free Societies

HIGH-LEVEL DEBUGGING: FACILITIES AND INTERFACES. DESIGN AND DEVELOPMENT OF A DEBUG-ORIENTED IDE

BY NICK PAPOYLIAS

COMMITTEE:

Assistant Professor Michail G. Lagoudakis (Supervisor)

Assistant Professor Vasilis Samoladas

Assistant Professor Katerina Mania

Submitted to the Department of Electronic and Computer Engineering in partial fulfilment of the requirements for the degree of Diploma in Engineering

Technical University of Crete

October 2009

By Nick Papoylias: *High-Level Debugging: Facilities and Interfaces. Design and Development of a Debug-Oriented IDE,* Submitted to the Department of Electronic and Computer Engineering in partial fulfilment of the requirements for the degree of Diploma in Engineering, Copyright © October 2009. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation.

Dedicated to my family.

My father George, my mother Vagellia and my sister Matina.

ABSTRACT

While debugging in general is an essential part of the development cycle, debuggers have not themselves evolved over the years as other development tools have through the advancement of Integrated Development Environments. In this project we propose a way to overcome this problem by introducing, designing and developing a high-level debugging system.

High-Level debugging systems are systems that integrate a source - level debugger with other technologies as to extent both the facilities and the interfaces of the debugging cycle. We designed and developed such a system in a debugging-centric IDE. This IDE introduces among other things: *syntax-aware navigation*, *data-displaying and editing*, *reverse execution*, *debugging scripting* and *inter-language evaluation* through the integration of its source-level debugger (gdb) with a full-fledged source parser, data visualisation tools and other free software technologies.

When large numbers of non-technical workers are using a programmable editor, they will be tempted constantly to begin programming in the course of their day-to-day lives. This should contribute greatly to computer literacy, especially because many of the people thus exposed will be secretaries taught by society that they are incapable of doing mathematics, and unable to imagine for a moment that they can learn to program.

— Richard M. Stallman [1]

ACKNOWLEDGEMENTS

My work in science and engineering would not have been the least fruitful if it wasn't for the unconditional support of my family. *My father George, my mother Vaggelia and my sister Matina.*

All those years I 've spend in T.U.C a lot of people accompanied me and helped me in my pursuits. Endless hours of scientific, philosophical and political brain-storming and struggle. Some of them stand out of this long list.. Thank you Julie, Blasi, Kosta K., Kosta A., Dimitri.

I am also grateful to my supervisor and committee for the trust they showed especially in the early stages of my work, when these thousands lines of code were just a mere proposal.

I would also like to thank the free software community and the Greek student's movement for giving me the software and psychological tools to pursuit my goals.

And last but not least, many thanks goes to Misha the Bear (Moscow '80 Olympics) for being the great mascot that he is.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 THE DEVELOPMENT CYCLE AND DEBUGGING.

From the broad spectrum of computer science the work presented here in focuses in the field of *Programming Languages and Tools* and in that of *Human-Computer Interaction*. More specifically we concern ourselves with *Debugging Systems* and *Integrated Development Environments*, that is to say with some of the most important production tools that shape the development cycle of software engineering.

While the term 'debugging' itself is usually attributed to Grace Hopper creator of Cobol [2], debugging as a diagnostic process spans in many more fields than programming languages and even beyond computer science and engineering. This general applicability stems from the very nature of human-labour where an initial imaginary *goal* is constantly shaping and being shaped by the overall *process* of production. This interaction between the final goal and the process is generally realised via the reoccuring feedback in between the cycles of applying and evaluating in the presence of natural laws.

A debugging system is a tool that can monitor and control the execution flow as well as the evolution of data of an algorithmic process.

This evaluation and revaluation can be mediated by technical means as is the case of oscillographs in hardware manufacturing, or cardiographs in medicine. In our case this mediation comes in the form of a debugging system that can monitor the execution flow as well as the evolution of data in time of an algorithmic process.

Now depending on the nature of the flaws discovered during production we are presented with a wide range of tools, some of which can even automate the process of debugging and testing with - almost - no human intervention. That is the case with syntax checking (that is an integral part of language compilers), unit testing suites, static analysis tools and to some extent delta debugging techniques [3].

But for the most part logical errors from their very nature tend to be elusive of automated debugging tools, so technological mediation in

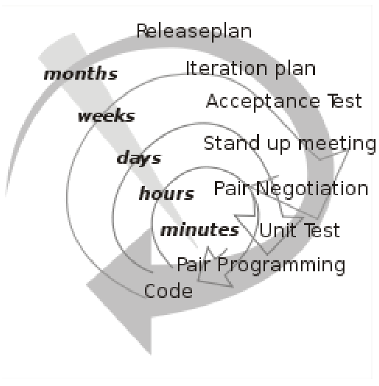


Figure 1. Development Cycle Example.

Figure 1 shows the development cycle of Extreme Programming software engineering paradigm, which focuses on feedback in between programming/debugging and testing cycles.

the form of human-computer interaction is needed for the programmer to debug his algorithms.

In this work we concern ourselves with the latter form of debugging tools, most commonly refered to as *source-level debuggers*, trying to *both* evolve them in the direction of better monitoring the execution of programs as well as in interacting more naturally with the programmer.

The significance of debugging tools in the overall development cycle can be seen in scientific publications concerning *effort estimation*[4] and *project management* which on average assert that testing and debugging cover roughly 50 % percent of the development time [5].

In our opinion though the most crucial aspect of debugging and execution monitoring is it's ability to widen humanity's grasp of automated processes of production. In our not so distant future it may even become a necessity for programming skills to be part of the standard primary curriculum, given our present state and momentum of automation. [1]

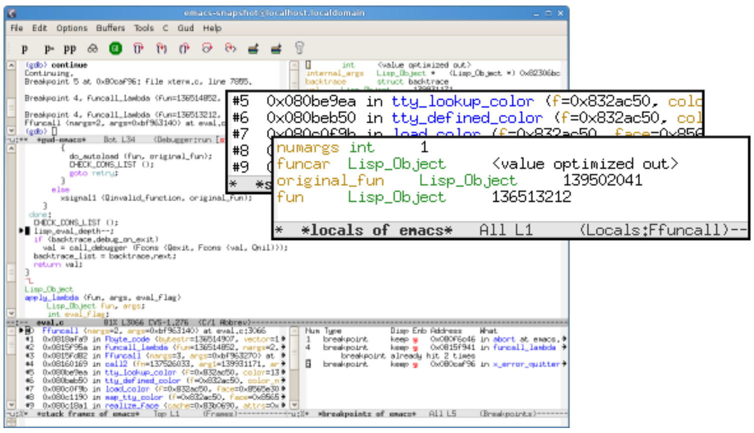
1.2 PROJECT WORKFLOW AND IDES.

Debuggers although so crucial for the development cycle are not the only tools of the development process. A whole range of software from

1 If of course in the meantime humanity will not destruct itself instead of socially and technologically advance towards producing goods in abundance and supporting access to scientific creativity for all human beings. [6]

compilers to project management scripts and from revision systems to profiling tools aid the developer to achieve his goals. We call the succession of processes and tools used to develop a project the *'project's workflow'*. Starting from the early days of operating systems, various environments where used to automated repetitive parts of this workflow and facilitate communication between the different tools used.

We can trace such environments back to the first command line interpreters that incorporated their own scripting language, such as the Bourne Shell [7] which is still in wide use even today. But when it comes to integrating *all* programming tools in one consistent environment, it is the Emacsen [2] and especially the Gnu-Emacs editor [1] that revolutionized the field. It was indeed here that *the editor, the console, the compiler, and the debugger* where first brought together in such a consistent way. Needless to say that this combination is what in general terms defines IDEs even today, although the field has since grown substantially as to incorporate many more technologies.


emacs-snapshot@localhost.localdomain
File Edit Options Buffers Tools C Gud Help

```
(gdb) continue
Continuing.
Breakpoint 5 at 0x80caf96: file xterm.c, line 7885.

Breakpoint 4, funcall_lambda (fun=136514852,

Breakpoint 4, funcall_lambda (fun=136513212,
Ffuncall (nargs=2, args=0xbf963140) at eval.c
(gdb) 
-u:** *gud-emacs*    Bot L34    (Debugger:run [
```

```
int          <value optimized out>
internal_args   Lisp_Object *   (Lisp_Object *) 0x82306bc
backtrace    struct backtrace
```

```
#5  0x080be9ea in tty_lookup_color (f=0x832ac50, colo
#6  0x080beb50 in tty_defined_color (f=0x832ac50, col
#7  0x080c0f9b in load_color (f=0x832ac50, face=0x856
#8  (
#9  (
*  *s
```

```
numargs  int          1
funcar   Lisp_Object       <value optimized out>
original_fun     Lisp_Object       139502041
fun      Lisp_Object       136513212

*  *locals of emacs*   All L1      (Locals:Ffuncall)--
```

```
            {
              do_autoload (fun, original_fun);
              CHECK_CONS_LIST ();
              goto retry;
            }
          else
            xsignal1 (Qinvalid_function, original_fun);
        }
    done:
      CHECK_CONS_LIST ();
      lisp_eval_depth--;
      if (backtrace.debug_on_exit)
        val = call_debugger (Fcons (Qexit, Fcons (val, Qnil)));
      backtrace_list = backtrace.next;
      return val;
    }
    ^L
    Lisp_Object
    apply_lambda (fun, args, eval_flag)
         Lisp_Object fun, args;
         int eval_flag;
--:--  eval.c        81% L3066 CVS-1.276  (C/l Abbrev)------------------
```

```
#0  Ffuncall (nargs=2, args=0xbf963140) at eval.c:3066
#1  0x0818afa9 in Fbyte_code (bytestr=136514907, vector=1
#2  0x0815f95a in funcall_lambda (fun=136514852, nargs=2,
#3  0x0815fd82 in Ffuncall (nargs=3, args=0xbf963270) at
#4  0x08160169 in call2 (fn=137526033, arg1=139931171, ar
#5  0x080be9ea in tty_lookup_color (f=0x832ac50, color=13
#6  0x080beb50 in tty_defined_color (f=0x832ac50, color_n
#7  0x080c0f9b in load_color (f=0x832ac50, face=0x8565e30
#8  0x080c1190 in map_tty_color (f=0x832ac50, face=0x8565
#9  0x080c18a1 in realize_face (cache=0x83b0690, attrs=0x
-u:%*  *stack frames of emacs*   Top L1    (Frames)------------
```

```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080f6d46 in abort at emacs.
4   breakpoint     keep y   0x0815f941 in funcall_lambda
        breakpoint already hit 2 times
5   breakpoint     keep y   0x080caf96 in x_error_quitter
-u:%*  *breakpoints of emacs*   All L5    (Breakpoints)-------
```

Figure 2. Gnu-Emacs integration of Gdb.

Speaking of modern IDEs and the current trends in their development today, we can see that the primary focus is more in the aforementioned integration of new tools - such as gui builders - rather than in the refinement and development of the more basic technologies that constitute the base of the environment, such as debuggers.

In this work from the perspective of the development of IDEs we shall focus in the integration and development of debugging tools trying to elevate the programmer's experience from the ground up. To do so we must first introduce the reader to some background knowledge about

2 plural, meaning class of Emacs editors

the inner working of debuggers and their relation with IDEs, starting with the following chapter.

2

BACKGROUND MATERIAL

2.1 SYMBOLIC DEBUGGING INNER WORKINGS.

We are mainly concered here with the inner workings of symbolic, source-level debuggers which are responsible for the debugging process of all non-kernel processes. For any such debugging system to perform its magic, access to scope and symbolic information from the source files and/or symbol table is required in the form of *debugging stabs*, as well as access to the underlying OS-specific debugging api. Figure 3 elaborates on these dependencies.

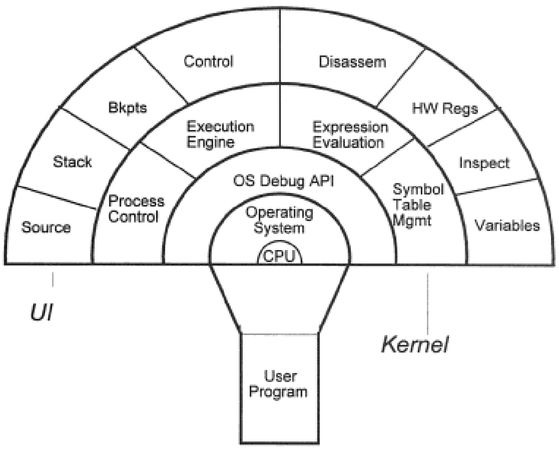


Figure 3. Symbolic Debugging Dependencies and UI.

This is as far as compiled languages go, although things are not all that different in the interpreted side, with the runtime-environment playing the role of the OS-specific debugging api while keeping record of symbolic equivalence between source code and intermediate byte-code. It could be expected that by their very nature debuggers of high-level interpreted languages should present more facilities to the programmer, given their environment's advantage for introspection. But oddly enough this is not the case, usually debuggers of such environments mimic a subset of the facilities presented to the C programmer, but this is a fact that we will address later on in chapter 3, when dealing with our *problem statement*.

We now turn our attention to the production of debugging information during the development cycle. The debugging information flow starts with the compiler. The compiler compiles C source in a '.c' file into assembly language in a '.s' file, which in turn the assembler translates into a '.o' file followed by the linker which combines the output with other '.o' files and libraries to produce an executable file.

Usually with some special directive upon invocation the compiler puts in the '.s' file additional debugging information, which is then slightly transformed by the assembler and linker, and carried through into the final executable. This debugging information describes features of the source file like line numbers, the types and scopes of variables, as well as function names and parameters.

For some object file formats, the debugging information is encapsulated in assembler directives known collectively as stab (symbol table) directives, which are interspersed with the generated code. Stabs are the native format for debugging information in the a.out and XCOFF object file formats, as far Unix environments are concerned.

The assembler adds the information from stabs to the symbol information it places by default in the symbol table and the string table of the '.o' file it is building. The linker merges the '.o' files into one executable file, with one symbol table and one string table. Debuggers use the symbol and string tables in the executable as a source of debugging information about the program.

We can see this information flow in Figure 4.

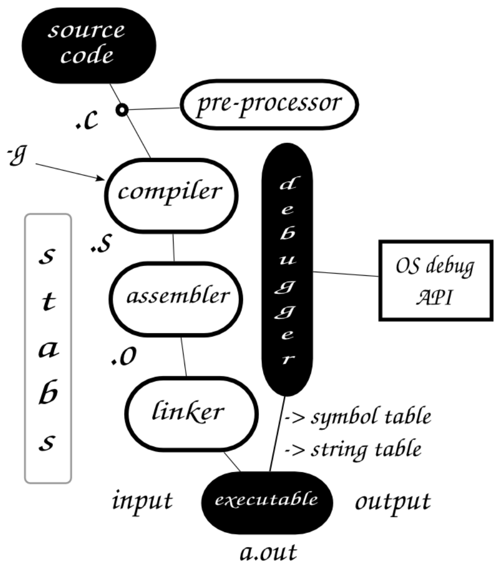


Figure 4. Debugging Information Flow.

More information on the subject of debugging stabs can be found in [8] and in [9].

2.2 SYMBOLIC DEBUGGING FACILITIES.

Now the most common descriptions for a symbolic debugger [10] follow more or less the following pattern:

1. A symbolic debugger, allows you to monitor what is going on 'inside' a program while it executes or what the program was doing the moment it crashed.
2. Start your program, specifying anything that might affect its behaviour.
3. Make your program stop on specified conditions.
4. Examine what has happened, when your program stopped.
5. Change things in your program so you can experiment with correcting the effects of one bug and go on to learn about another.

In terms of available facilities presented to the programmer - see Figure 3 - the most common of them are:

- *Line navigation*, per command-lines as well as naviating in and out of functions
- *Stack examination* upon program stop, in addition to frame and backtrace information.
- *Breakpoints*, as places of interest where the program should stop for further examination.
- *Watchpoints*, as varibles or expression of interest upon the change of which the program should stop for further examination.
- *Catchpoints*, as signals and os-specific events of interest upon the invocation of which the program should stop for further examination

- *Data viewing/editing* upon demand, for expression evaluation and editing.

Some additional facilities found at the current *'state of the art'* symbolic debuggers such as gdb and are still consider experimental are seperately examined in chapter 4.

2.3 DEBUGGING FRONT-ENDS AND IDES.

With the term debugging *front-ends* we are referring to the graphical (or some times textual) user interfaces that attempt to facilitate the use of a symbolic debugger (referred to as a *backend*) by providing a one-by-one access to the underlying facilities. This is the case for all major IDEs in current use today, such as Eclipse, Netbeans, Anjuta, Kdevelop or the proprietery Visual Studio e.t.c

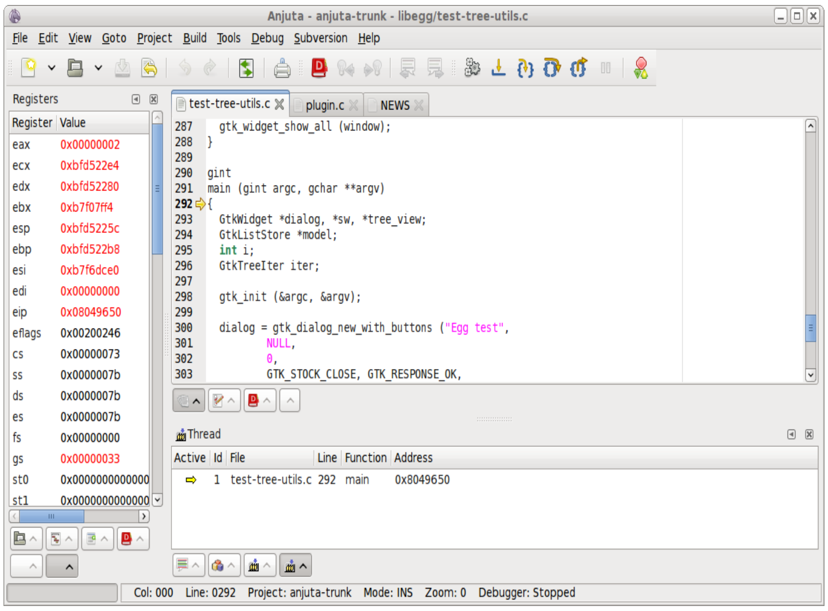
While we 'll be addressing this issue more closely up in chapter 3, this kind of *integration* for the debugger and other tools, does not convey new facilities but in effect only provides a visual summation of the development tools currently in use by the developer.

Nevertheless this kind of visual integration has gained symbolic debuggers wide spread use, which is one of the reasons that most of them have a special mode of operation, for communication with these *front-ends*. In the case of gdb that we 'll be discussing shortly, this communication is done via a domain-specific language, part of the gdb/mi (machine-inteface)[10].

2.4 DEBUGGING SYSTEMS.

Now debugging systems, in contrast with front-ends do not only provide access to the underlying facilities of the symbolic debugger but rather use them as a building block for larger systems that can provide execution monitoring. In this case the debugging facilities offered are a result of the integration of the symbolic debugger with other software, such as graph visualization tools, as is the case of the Data Display Debugger [11] (see Figure 6) and the Gnu Visual Debugger of the GNAT Programming Studio [12].

In this work we will elaborate on this kind of integration and visualization taking it some steps further, having multiple parts of the


Anjuta - anjuta-trunk - libegg/test-tree-utils.c
File Edit View Goto Project Build Tools Debug Subversion Help
Registers
test-tree-utils.c
plugin.c
NEWS
Thread

Register	Value
eax	0x00000002
ecx	0xbfd522e4
edx	0xbfd52280
ebx	0xb7f07ff4
esp	0xbfd5225c
ebp	0xbfd522b8
esi	0xb7f6dce0
edi	0x00000000
eip	0x08049650
eflags	0x00200246
cs	0x00000073
ss	0x0000007b
ds	0x0000007b
es	0x0000007b
fs	0x00000000
gs	0x00000033
st0	0x0000000000000
st1	0x0000000000000

```
  gtk_widget_show_all (window);
}

gint
main (gint argc, gchar **argv)
{
  GtkWidget *dialog, *sw, *tree_view;
  GtkListStore *model;
  int i;
  GtkTreeIter iter;

  gtk_init (&argc, &argv);

  dialog = gtk_dialog_new_with_buttons ("Egg test",
          NULL,
          0,
          GTK_STOCK_CLOSE, GTK_RESPONSE_OK,
```

Active	Id	File	Line	Function	Address
⇨	1	test-tree-utils.c	292	main	0x8049650

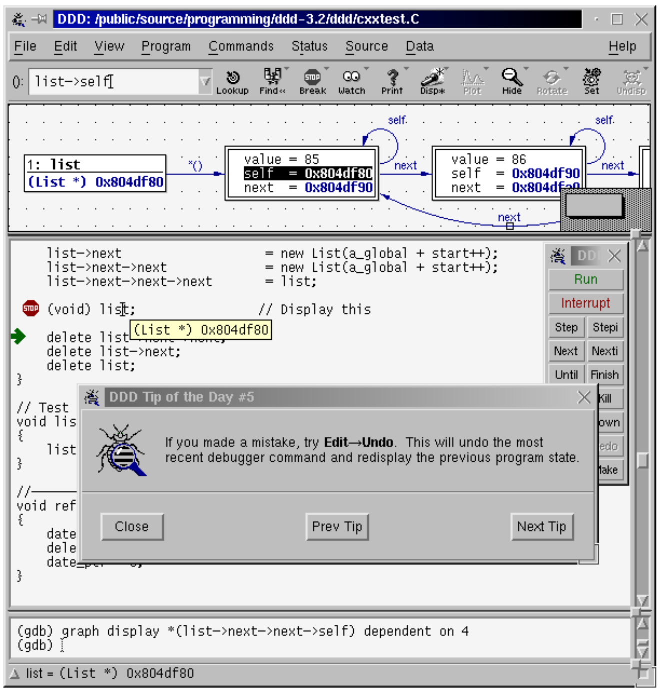
Col: 000 Line: 0292 Project: anjuta-trunk Mode: INS Zoom: 0 Debugger: Stopped

Figure 5. Example of a debugging front-end.

Figure 5 shows Anjuta the official Gnome IDE, which is a typical example of a debugging front-end.

debugging system interacting with each other offering many more options to the developer than before. In addition we will try to enrich the debugging information flow as to evolve debugging systems at a more fundamendal level.

But more on this in the following chapters, where we describe our solution approach to the problem statement that follows.



```
    list->next                    = new List(a_global + start++);
    list->next->next              = new List(a_global + start++);
    list->next->next->next        = list;

    (void) list;                  // Display this

    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//
void ref
{
    date
    dele
    date
}
```

(List *) 0x804df80

If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

list = (List *) 0x804df80

Figure 6. DDD a simple yet powerful debugging system.

3 PROBLEM STATEMENT

3.1 THE CURRENT STATE OF AFFAIRS.

Having introduced the reader to some background material, let us now consider the fact that today's advancement in IDEs although constantly offering new programming tools and levels of sophistication, has left debuggers where they were 20 years ago, mainly giving the programmer the ability to pinpoint source-lines of interest, stepping through subsequent lines of source-code, and monitoring certain expressions as he goes along. Of course the underlying technology offers some additional number of tools - in the same line of thinking - which are nevertheless rarely "embedded" in IDEs and used by the programmer, if - that is - any debugging tools are embedded or used at all.

Why debuggers have not substantially evolved over the years, regardless the interesting efforts that were occasionally proposed, is a very interesting question, which has surprisingly only two possible answers. The first answer, comes quite naturally, and is of course false stating that debuggers are "perfect systems" which have reached the end of their evolution potential. If that statement were true of course, we should all be living in a bug-free software world, with no frustration whatsoever about the development process and the problems it involves. The second answer states that we didn't had - not up until recently - the "means" to evolve debuggers, or more correctly, up until recently, in absence of the "means" that could eventually lead to the evolution of debugging systems, thinking about such an evolution was impossible and of course talking about it was meaningless.

Having already seen the importance of software monitoring and debugging we favour the second approach and propose - both theoretically and technologically - a possible route for their evolution that would hopefully meet the current needs of software engineering.

3.2 THE NEED FOR HIGH-LEVEL DEBUGGING SYSTEMS.

While this evolution can surely come in the form of rethinking the whole information debugging flow - and that's in part what *we* will do - it can also emerge in the form of a debugging system, that as we saw earlier can use current debugging technology as it's building block. This system though given the current advancement in computer technology and development tools should exhibit higher level of operability than similar efforts a decade or more ago[1].

So let us now introduce the notion of *"High-Level debugging"* systems - as opposed to "source-level" debuggers or legacy debugging systems - that are in general terms systems that can be build on-top of today's debugging facilities, not simply by giving *graphical access* to the underlying technology, but by offering more sophisticated methods of monitoring a software system, as well as new ways for a programmer to form his development cycle.

The term *"high-level"* in this approach can be thought of as a loan from the programming language world, the usage of which can be substantiated by the fact that "lower-level" debuggers can be used as building blocks for more complex debugging systems. As a matter of fact "lower-level" debugging languages do exist as the aforementioned Gdb/Mi [10] and can be used to build "higher level" debugging structures.

We should note here that although the term has not yet been standardised in computer literacy (being in its infancy as a subject) it - appears in more or less the same context - in a lot of works like *Golan and Hanson*, *[13]* , *Ducassé and Emde*, *[14]* and others that we will be discussing shortly after. Moreover we shall see that in these last couple of years a number of research initiatives concerning the development of debugging systems have appeared taking various approaches, and we

1 That is the case of DDD, which we saw earlier as an example of a debugging system. The main characteristics of this - then revolutionary - system, were forged in 1995.[11]

believe that this scientific interest proves that our initial observations in the beginning of this work about the future needs of this field were justified.

4 RELATED WORK

Before continuing we should take a brief look on related work, and current efforts on advancing debugging technology. We have divided our research of the field in two major categories one concerning bibliographic proposals, conference proceedings and published work and the other current trends in technological efforts towards new debugging systems which have reached their alpha or beta stage of production. In this last category we will find - besides the free software community - major players of the software industry like *Microsoft, Google, IBM and Wolphram Research Inc.* which our humble - in terms of resources - effort dares to challenge, and we believe that it does so with some significant success.

4.1 PUBLISHED WORK

In published work there is a great deal of references for high-level debuggers that are concerned with task specific debugging such as parallel execution. Here for example the main problem usually stems from the specific inability of traditional debugging schemes to monitor the complexity involved in concurrent execution, so in most cases a *higher* level of monitoring abstraction is proposed. This is the case of *Aral and Gertner, [15]* as well as *Cunha et al., [16]* where interesting propositions appear such as *variable tracing, debugging via assertions and scalable remote debugging*, although some of them are now pretty outdated. For reference a *State of the Art* report of this field can be found in *Huselius, [17]* . Our own approach though to high-level debugging systems, as we will see later, is more broad than the ones described above and not so task-specific, nevertheless the basic tools [1] for a future development in this area too, have been led.

1 such as thread and fork/exec monitoring

From these works we should separately examine *Cheng*, *[18]* where a separation from the source-line navigation approach in debugging was first proposed. The unique feature that *HDB*[2] introduced was that of debugging *checksums*, which were used to compress arrays and groups of variables without losing meaningful information. By using these checksums and their differences this article supports that it is possible and more convenient to detect misbehaviour of a program at a place near the source of the error. Now despite the fact that this approach did not make it to mainstream debugging, mainly because *unit testing* tools were widely introduced, we believe that the suggestion of navigation through larger and well defined portions of a program while monitoring it's execution is a worthy pursuit. A similar high-level approach is proposed in *Cifuentes et al.*, *[19]* from the completely different perspective of assembly code debugging and monitoring. More on this subject when we discuss *syntax-aware debugging and navigation* in Chapter 5.

Then there are *high-level* debugging systems that have been proposed and concern a *domain-specific extension language* that leaves on top of legacy debugging systems. This is truly an intriguing concept. That is the case with *Golan and Hanson*, *[13]* with *Duel* and *Ducassé and Emde*, *[14]* with *Opium*. In the case of *Duel* we have a high-level debugging system targeting the C language that during normal execution interacts with *gdb* providing new expression evaluations through a domain specific query language. In the case of *Opium* on the other hand the domain-specific query language (based on *Prolog*) analyses traces of program execution for post-mortem analysis. Both tools really provide higher levels of abstraction, extensibility and new ways for the programmer to monitor his running source-code. But we believe that there is a catch here given the fact that since their proposals debugging technology didn't catch up with these ideas even though for example *Duel* that was developed in Princeton is now part of *Microsoft Research* bibliography.

2 the tool that *Cheng*, *[18]* was based on

In essence a domain-specific language for debugging no matter how powerful and extensible, adds immensely to the complexity of the resulting development environment, and learning such a new language may seem like the last thing a programmer will want to do. In contrast maybe to a widely used and understood general purpose language that provides the same functionality without the burden of learning a debugging-specific one. This is actually where *we* are heading, and independently this is where *gdb* wants to go [20], but we will talk more about this subject later.

In addition there is the thriving field of *reversible and replay debugging*. We regard the ability to debug backwards in time one of the key components of high-level debugging systems, and so does the free software community [21]. A lot of different technologies have been proposed to accomplish this behaviour but all fall broadly in three major categories: *a)* remote execution in a virtual environment *b)* native execution via machine instruction monitoring and *c)* replay execution via language-dependent traces. In terms of published work some of these approaches can be found in *Narayanasamy et al., [22]* , *Lewis, [23]* and *Akgul and Mooney, [24]* .

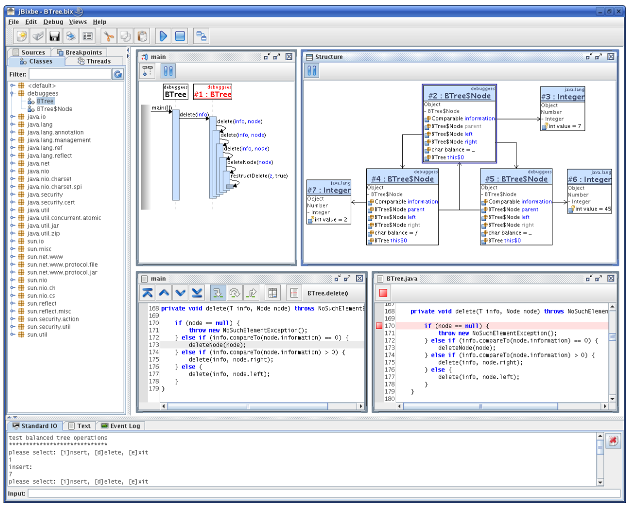
Last but certainly not least for reasons that will become more apparent in Chapter 5 and 6 our work is also related to the work of the Harmonia Project in Berkeley (see *Boshernitsan et al., [25]* and *Begel and Graham, [26]*) which deals with *high-level interactive software development*, *Language-Aware programming tools* and *programmer-computer interaction* although their work has yet to be expanded in debugging.

4.2 TECHNOLOGICAL ADVANCEMENTS

Beyond academic tools and proposals the last couple of years there were some advancements made directly on the technological end, in popular IDEs and widely used debugging tools. Although we must say that the overall adoption of new technologies is still incredibly slow.

Besides expanding basic multi-threading support which appears in both major source-level debuggers [3], the gdb development team has lately taken a step further giving a lot of attention in the aforementioned facilities of reverse/replay debugging, and scripting extensibility [27], [20]. Our work relies heavily on some experimental work done for gdb [28] for the first subject but we have taken a very different architectural approach on the second. Nevertheless this convergence on experimental choices strengthens our belief that we are on the right track.

We now turn our attention to advancement in debugging aids through IDEs. Starting with industry standard environments, some related and interesting work appears in *Visual Studio's data visualisers* [29] were data in html, xml or image form is according to semantics visualised to the programmer during debugging. This is a very interesting feature despite the fact that usually logical errors are found in the inner relationships of data-structures and not in the semantic representation of a type which usually appears in the running process – as is the case with images. Then there is the *high-level debugger* of Mathematica [30] which supports arbitrary computation at breakpoints in it's own language, including some visualisation of intermediate results, mainly mathematical formulae.



```
private void delete(T info, Node node) throws NoSuchElement

    if (node == null) {
        throw new NoSuchElementException();
    } else if (info.compareTo(node.information) == 0) {
        deleteNode(node);
    } else if (info.compareTo(node.information) > 0) {
        delete(info, node.right);
    } else {
        delete(info, node.left);
    }
}
```

```
test balanced tree operations
****************************
please select: [i]nsert, [d]elete, [e]xit
i
insert:
7
please select: [i]nsert, [d]elete, [e]xit
```

Figure 7. JBixbe call-graph and data visualisations.

In terms of replay/reverse debugging now, we have a lot of companies and groups implementing the basic feats, some of them on top of the remote machine interface of gdb, others on top of the *javaVM* like the *temporal debugger* [31] from *Cisco Systems* and *Bill Lewis'* ODB,

3 Gdb and MVSD

presented at GoogleTalks [32]. For the end an independent - but proprietary - project that we would like to mention is the high-level debugger JBixbe [33] which has some advanced capabilities in terms of *call-graph visualisation* and also a basic support for visualising data like *ddd* does, see Figure 7.

Having now scattered the field for related innovations and proposals we now turn our attention to the solutions we propose about high-level debugging and integrated development environments, starting with the next chapter.

5 OUR APPROACH

5.1 RETHINKING THE DEBUGGING INFORMATION FLOW

All features and facilities of debugging systems depend on the amount and nature of information that is available in the debugger and concerns the equivalency of source code with the running process. As we saw earlier, in the most such systems in current use today, this kind of information is usually embedded by the compiler or interpreted in the executable or intermediate byte-code respectively. This is done according to some predefined standard such as the *pdx* stabs format [8] which anticipates specific uses for the kind of information that it embeds.

In our work in order to support current development and future uses of debugging systems other than the ones offered by today's technology we *expanded* the nature and amount of information available to the debugging system by providing it with direct access to the semantically annotated parse tree of the source code. Our choice alters the classical debugging information flow which was presented in chapter 2 as follows:

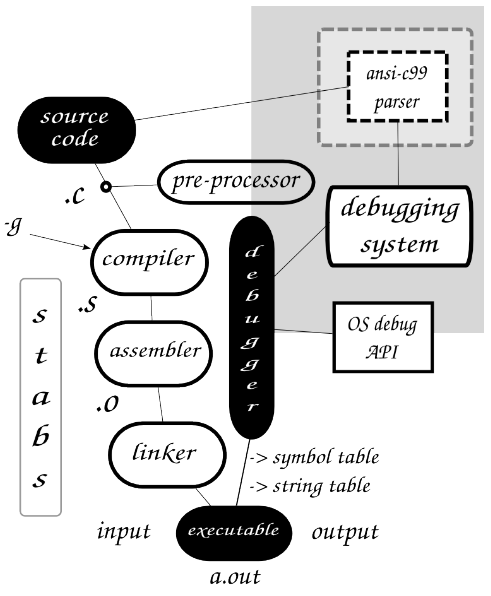


Figure 8. Expanding the information available to the debugging system.

Now as seen in Figure 8 in order to construct this semantically annotated parsing tree and provide additional information to the debugger, we designed and developed a seperate parser as part of our debugging system. This parser will provide us with the means to develop and support features like *syntax-aware navigation*, *reverse-debugging and tracing* among other things and potential future uses. To our knowledge this is the first time that debugging information is enhanced in such a way rather than simply being embedded in the intermediate or machine code.

In addition such a parser can also be used as a crucial building block for a lot of things that are in current use today in IDEs or have been proposed for their development. Some examples include *symbol-browsing, unit-testing, documentation extraction, syntax-completion and refactoring*. To some extend these other uses are the aim of the *Harmonia Project* in Berkeley [34], that we mentioned in chapter 4. Their architectural approach also includes a seperate parser to support these functions rather than using the first pass of the compiler itself. This desicion seems mandatory for the time being, due to the architectural structure of current compilers which favors syntax-trees in intermediate languages for optimization purposes. The [35] multi-language introspection tool [35] is also relevant in this context. In the future we may be able to support these functions directly from the compiler itself, see [36] and [37].

Keeping this new approach in mind, let's now take a closer look to the facilities, intefaces and other interacting parts of our *high-level* debugging system.

5.2 THE FIVE PILLARS OF HIGH-LEVEL DEBUGGING

SYNTAX-AWARE DEBUGGING: This first feature is intented to be the workhorse of the overall effort, and is based directly on the affornentioned extention of available debugging information. The implication here is that by using the parser to analyze source code, *debugging and execution navigation can take place in terms of specific syntactic* *structures* having different "template" information readily available to the user according to syntactic and semantic information of the target language. The programer is thus able to pinpoint structures of interest as a whole, and not just source-lines, while debugging can take place both as stepping through a "logical-unit" of evolution and as watching the execution flow over time, freezing the program when needed. The navigation through the syntax-tree operates in two modes *breadth and depth first* besides the classical single-line mode. In addition, through the general purpose extention language that we will examine later, conditional debugging as well as user-defined in-structure information can be supported.

DATA VISUALISATION: Greatly inspired by the work on DDD, data visualisation is an essential part of our high-level debugging system. Taking things a bit further than conventional approaches we have used and integrated software which is used for representing structural information as diagrams of abstract graphs and networks [38], and on top of that we have provided a comprehensive and generic API for visualising language-oriented datatypes *(containers, strings, integers, interlations)*. In addition we have developed from scratch a graphical widget for interacting with these graphs, which supports *editing and updating graph values, infinite graph expansion via menus and depth settings, layout capabilites* and other features that we will explore later on in chapter 7.

Futhremore, it is worth mentioning that our data visualisation solution intergrates nicely with our syntax parser. One example of this integration is the ability to examine identifiers that are *referenced* in code as far as the execution line, and not only local variables of the current scope that usually exclude global identifiers while including unreferenced uninitialised data. More details about the inner-workings of our data visualisation subsystem can be found in chapter 6.

GENERAL-PURPOSE EXTENTION LANGUAGE: Our third step was to integrate a general purpose extention language to our debugging system, which will be able to control our parser, the visualisation subsystem, the symbolic debugger as well as the "high-level" debugging facilities. We chose *python* which is a widely used and understood high-level language, distancing ourselves from the domain specific approaches that we saw earlier in previous chapters. Part of what we have achieved here (controlling the symbolic debugger via python) is also a *future* goal of gdb, which aims to use this extention language as a separate platform for writting usefull tools [20].

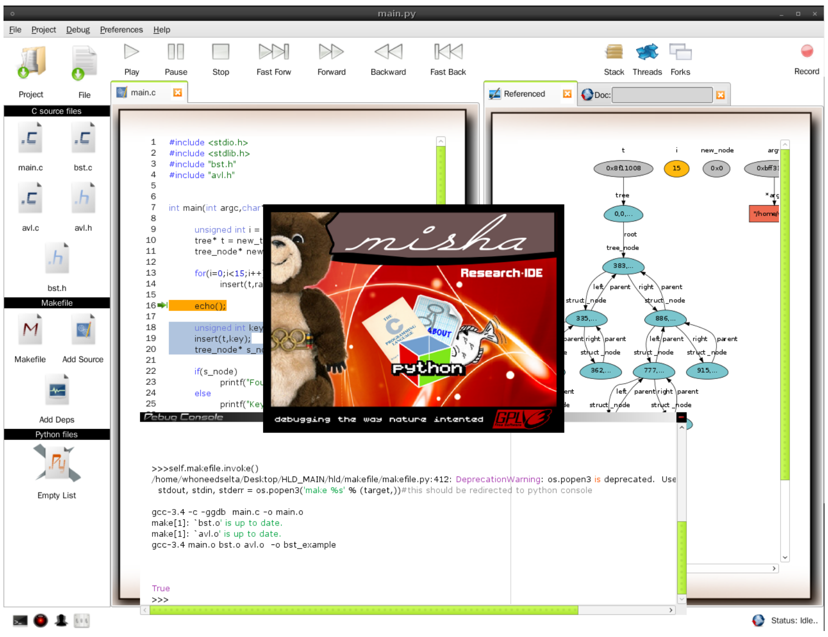
In our approach besides being able to control all of the different subsystems (and not just the symbolic debugger) from the debugging console and *in-project* python scripts, thus being able to extend both the debugging system and the IDE, there is the ability to *directly* and *seamlessly* call each project's C functions from within python. This feature besides being usefull for unit testing and code benchmarking purposes, encourages a multi-language approach in software enginnering which is a critical aspect of our future intentions for *Misha*.

REVERSE DEBUGGING: Stepping backwards in time while debugging is a valuable tool that cannot be absent from our research effort. It is also a community proposal, listed in the high priority project list of the *Free Software Foundation*. In responce to this interest and based upon the still exprerimental work done for *i386* native reverse execution [27], we integrated and enhanced the execution record facility of *gdb* with our syntax-aware navigation so that it is able to execute back in terms of complete syntactic structures, just as the programmer using the forward execution will have expected.

As we will see later with the integration of our parser with a recording execution system, memory issues that concern reverse execution can be addressed through static-code analysis. More on this on chapter 8, when we will deal with future work.

INNOVATIVE INTERFACES: Presenting the programmer with a lot of data and options all at the same time, is not always the best thing to do, but debuggers and IDEs from the very nature tend to demand their share of the desktop. In order to address these issues we developed *new graphic widgets* for the gnome platform, *innovative input facilities* that support speech and bluetooth devices, a *web-documentation system* with a dedicated crawler and a comprehensive *project management system* based on *gnu-make* that automates the search for library and source dependencies resolutions. We will examine these features with greater detail in the next chapter, where it will become obvious that although our research IDE is *debug oriented* it has also the potential of becoming a full-blown development solution.

So lets now take a look at the inner workings and the technologies that can make all these features happen.



```
#include <stdio.h>
#include <stdlib.h>
#include "bst.h"
#include "avl.h"


int main(int argc,char

    unsigned int i =
    tree* t = new_t
    tree_node* new

    for(i=0;i<15;i++
        insert(t,ra

    echo();

    unsigned int key
    insert(t,key);
    tree_node* s_n

    if(s_node)
        printf("Fou
    else
        printf("Key
```

```
>>>self.makefile.invoke()
/home/whoneedselta/Desktop/HLD_MAIN/hld/makefile/makefile.py:412: DeprecationWarning: os.popen3 is deprecated.  Use
  stdout, stdin, stderr = os.popen3('make %s' % (target,))#this should be redirected to python console

gcc-3.4 -c -ggdb  main.c -o main.o
make[1]: `bst.o' is up to date.
make[1]: `avl.o' is up to date.
gcc-3.4 main.o bst.o avl.o  -o bst_example


True
>>>
```

```
t           i     new_node
0x8f11008   15    0x0
tree
0,0,...
root
tree_node
383,...
left  parent  right  parent
struct_node      struct_node
335,...          886,...
362,...   777,...   915,...
```

Figure 9. Misha in action.

6 IMPLEMENTATION

There are over a dozen different technologies and software that were used for the development of our debugging system and IDE. For the purposes of this chapter we will focus in the major building blocks and technologies and we will describe their inner workings with some detail. *Five* interconnected sub-systems constitute the core implementation of our *debug-oriented* IDE:

- the syntax parser
- the symbolic debugger
- the graph visualisation sub-system
- the extension environment and language
- the project-management module

On top of them lies the *high-level debugging API* and the graphical user interface (editor, console, on-line help system) that uses it's facilities.

As we mentioned before, some additional technologies have also been developed to elevate the programmer's experience that include among other things *specifically tailored graphical widgets and input methods*.

The development model was generally multi-lingual with the processor and memory intensive parts written in *C* [39] - like the syntax-parser - while the graphical interface and all other extendible parts written in *Python* [40]. More detailed information about design, tools, libraries and technologies used, follows.

6.1 THE SYNTAX PARSER

Our syntax parser is an *ansi-c99* compliant parser with support for all the major gcc extensions and c-preprocessor directives. It is based upon the standard BNF notation for the language described originally in *Brian W. Kernighan, [39]* and revised here [41]. Besides syntax, it collects a variety of semantically interesting information for generic use in IDEs as well as debugging purposes in our particular case. We designed our parser with modularity in mind so that it can be used separately in other projects and for different circumstances and purposes. It was devised with standard and robust compiler construction tools – *flex* [42] and *bison* [43] – with the support of the standard c and gnome libraries (*libc* [44], *glib* [45]) for data-structures and miscellaneous services. Our resulting implementation in terms of context-free grammars is an *LALR(1)* parser. The syntax tree is an *n-ary tree* with uniform nodes. Typical structure and information stored per node can be seen in the following code-snippet.

Syntax node snippet

```
typedef struct syntax_struct_tag{//the tree is initialise with
    basic types
        syntax_type type;//syntax type
        gchar* name;//var,function,type name
        GNode* return_id;//var decl type, func decl type, typedef
            initial struct
        gchar* expr_text;//the text on for,if,while,do
        gint start_line;//self-explanatory
        gint end_line;//self-explanatory
        gchar* filename;//self-explanatory
        GList* refer_ids;//id list of referenced id's in this
            syntax structure
        GList* expr_ids;//id list of referenced id's in the syntax
            structures expression section
        gint dimension;//dimention of var,type
        gint refer_depth;//reference depth ** of var or type
        gboolean mirror;//True if it is a struct declaration
            awaiting definition
        GList* qualifier_list;//Data type qualifiers
        GList* ret_lines;//FUNC_DEF keep here their return lines (
            expr of return lines can be retrieved via line number)
        GList* misc;//for future reference, enumarations keep here
            the enumed strings, functions keep temporarily
        //their parametrs,vars keep their init assign status
}syntax_struct;
```

6.2 the symbolic debugger

In order to use gdb as a building block for our debugging system and expose its functionality to our general purpose extension language, we wrote a second parser for the *gdb-machine-interface*. This parser was then used to devise the low-level debugging api which is accessible via the extension language and the high-level debugging interface. We used python's *PLY* [46] for our second parser and managed to cover all of the underlying functionality of gdb as it is described here [10]. These python bindings for gdb, although architecturally different from what the gdb development team is currently planning [20] cover *today* a lot of ground of their planned future work.

6.3 the graph visualisation sub-system

As mentioned earlier we also developed a visualisation library for language-oriented datatypes using the *graphviz* package. Our recursively constructed datatypes are translated in the *dot-language* and accompanied tools which generate directed graphs as hierarchies using active research in the field of *graph visualisation* see *Koutsoos et al., [47]* .

Our visualisation API is fully customisable in terms of shapes, colours and presented interconnections, supports four generic datatypes (*numbers, strings, pointers/references and containers*) that can be further expanded through inheritance. The accompanied widget provides full navigation, customisation and editing for the graphs. Of course every aspect of the visualisation sub-system is accessible through our extension language. See Figure 10 for a sample drawing with *dot*.

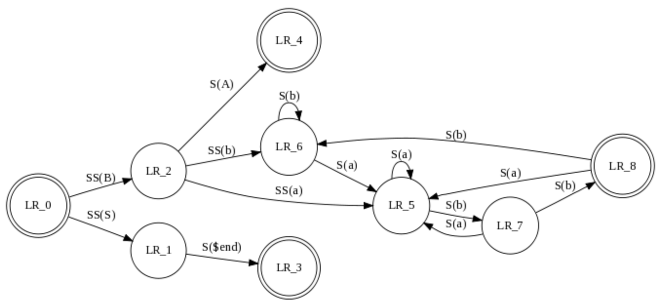


Figure 10. A sample drawing of a finite automaton described in the dot-language

6.4 the extension environment and language

We used *python* [40] as our extension environment and language and by using the language's introspection facilities we managed to provide both online (through our console) and on-demand (through python project files) extensibility to our debugging system and IDE. The scripting environment is failsafe and secured so as to prevent unexpected hang-ups of the application in case of errors or exceptions and has access to everything our system has to offer. From the high-level debugging API to the underlying symbolic debugger and the ansi-c parser, and from the data-display subsystem to the IDE gui itself. But first and foremost it can *directly* call the programmer's c project functions providing an unprecedented tool for testing, debugging and multi-lingual development.

This is achieved through *execution-time* translation of expressions and *textual evaluation* in the target language, either through the interpreter of a high-level language or through the symbolic debugger for low-level languages such as c, which have direct access to the symbol table of the executable. We will promptly see some examples of this technology in the following chapter.

6.5 the project-management module

The project management module is based on the versatile and ubiquitous tool *gnu make* [48], making project imports and exports to other tools and environments a breeze. Our IDE reads and writes directly to the project's *Makefile* storing there any information it might need *without* messing with other user specific directives the programmer might wish to add to his project.

In addition our *Makefiles* automatically detect *header file dependencies* directly from source code, while managing *libraries, compilation flags and include paths* without the programmer's intervention using the *pkg-config* [49] utility in the background. In essence with our approach, all the programmer has to do is *add* source files to his project and *choose* the libraries he wishes to use *by name* from a list of *automatically* detected installed libraries in his system.

6.6 THE HIGH-LEVEL DEBUGGING API

Our high-level debugging API is a neat example of OOP design. All the functionality of our *high-level debugging system* is wrapped up in a class which directly inherits our *ansi-c parser API* and the *symbolic debugger* interface, thus enabling the integration of the two in implementing new debugging facilities. Figure 8 illustrates this concept. A template header of some basic navigation calls from our high-level debugging API can be seen in the following snippet:

Syntax node snippet

```python
class High_level_dbg(Gdb,Parser):
...
    def play(self,options):
        pass
    def stop(self,options):
        pass
    def pause(self,options):
        pass
    def forward(self,options):
        pass
    def backward(self,options):
        pass
    def fast_forward(self,options):
            pass
        def fast_backward(self,options):
            pass
        def record(self):
            pass
        def reverse(self):
            pass
        def record_stop(self):
            pass
        def resume(self,options):
            pass
...
```

6.7 THE USER INTERFACE

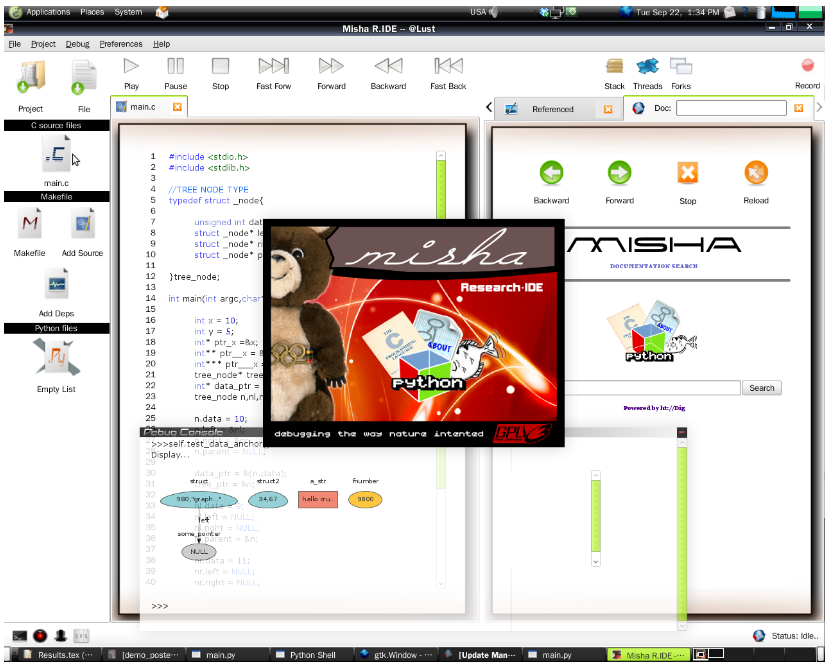
For our user interface we used the python bindings [50] of *gtk+* which is the standard gui library of the *gnome* [] platform. We developed from scratch new *interaction widgets* that include among other things a *tabbed two panel* interface a *graph editing* and a *console* widget besides enhancing existing technologies, such as the text and source editor that our toolkit provided.

Our on-line help system is based on the integration of a light *mozilla* [51] client inside our IDE which is served by a dedicated crawler software *htdig* [52] for specified documentation sites. Our *misha* terminal is a specifically initialised x terminal, while the speech recognition facilities are based on a *pocket-sphinx* [53] instance trained with the *wall street journal* corpora which is specifically tailored to recognise debugging navigation commands. Last but not least our icon artwork is part of the *nuoveXT* project [54] which is released under the *LGPL* license.

But let us now proceed to seeing the *Misha research I.D.E.* and our high-level debugging system in action.

RESULTS

We will be examining a demo project developed with our environment involving a standard implementation of a *Binary Search Tree* as well as an experimentation with the unsolved mathematical conjecture of the Collatz sequences, see [55] for more information. A full video presentation of this demo is available through the project's site at *launchpad* [56].



```
#include <stdio.h>
#include <stdlib.h>

//TREE NODE TYPE
typedef struct _node{

        unsigned int dat
        struct _node* le
        struct _node* ri
        struct _node* p

}tree_node;

int main(int argc,char

        int x = 10;
        int y = 5;
        int* ptr_x =&x;
        int** ptr__x = &
        int*** ptr___x =
        tree_node* tree
        int* data_ptr =
        tree_node n,nl,n

        n.data = 10;
```

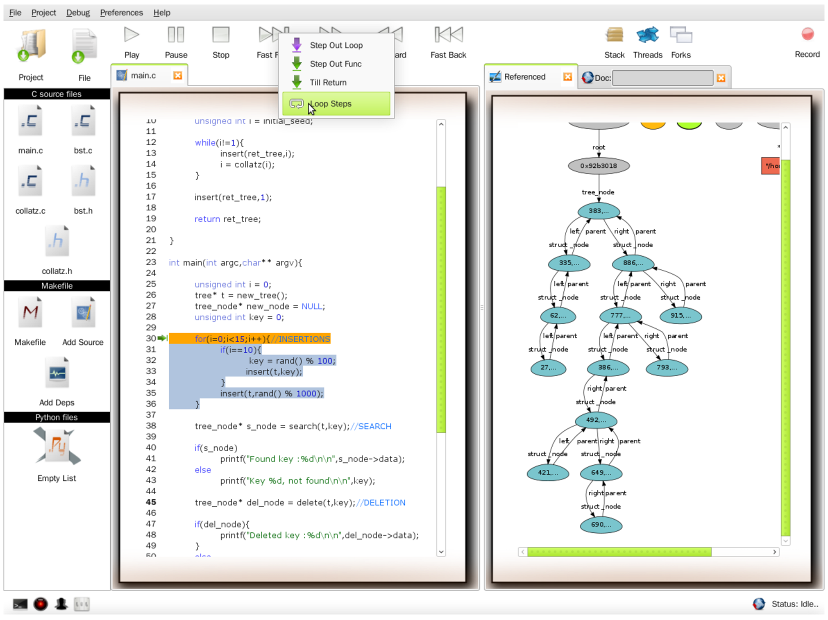
```
>>>self.test_data_anchor
Display...
>>>
```

Figure 11. Misha R.IDE welcoming screen

7.1 SYNTAX - AWARE NAVIGATION

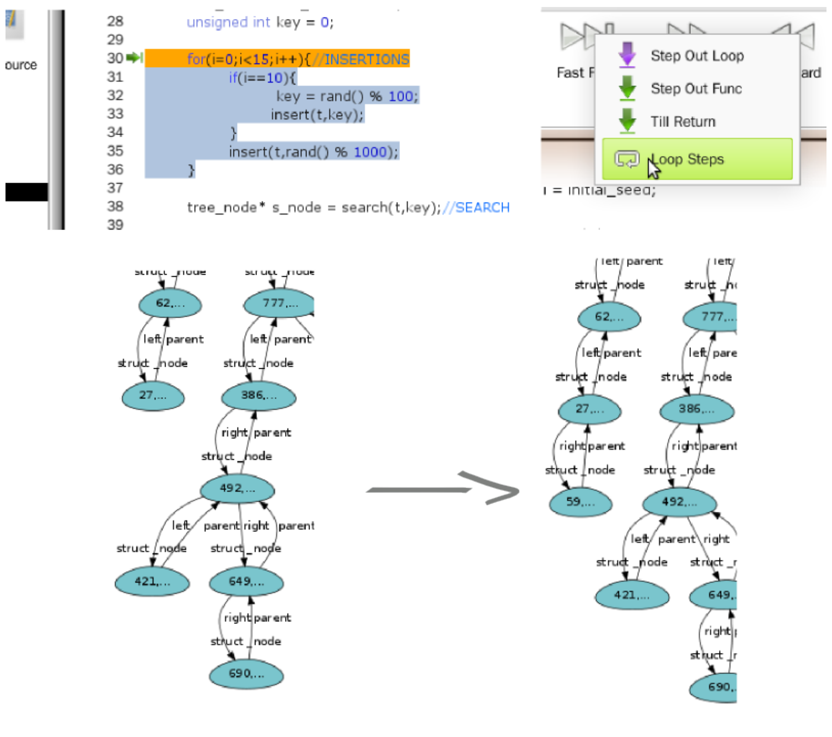
We will begin by demonstrating our syntax aware capabilities. As we can see in Figure 12 individual group statements, *if*, *while* and other syntax structures are blocked together to form *logical units* of execution that can accordingly be traversed. For example loop statements (either while or for loops) can be traversed as a whole structure, iteration by iteration or even in user-defined evolution steps. This traversal can be either *breadth-first* in a *bottom-up* fashion inside the syntax tree, or *depht-*

first while the debugger is automatically advancing through source code in *human-mode*.



```
        unsigned int i = initial_seed;

        while(i!=1){
                insert(ret_tree,i);
                i = collatz(i);
        }

        insert(ret_tree,1);

        return ret_tree;

}

int main(int argc,char** argv){

        unsigned int i = 0;
        tree* t = new_tree();
        tree_node* new_node = NULL;
        unsigned int key = 0;

        for(i=0;i<15;i++){//INSERTIONS
                if(i==10){
                        key = rand() % 100;
                       insert(t,key);
                }
                insert(t,rand() % 1000);
        }

        tree_node* s_node = search(t,key);//SEARCH

        if(s_node)
                printf("Found key :%d\n\n",s_node->data);
        else
                printf("Key %d, not found\n\n",key);

        tree_node* del_node = delete(t,key);//DELETION

        if(del_node){
                printf("Deleted key :%d\n\n",del_node->data);
        }
```

Figure 12. An aspect of syntax navigation.

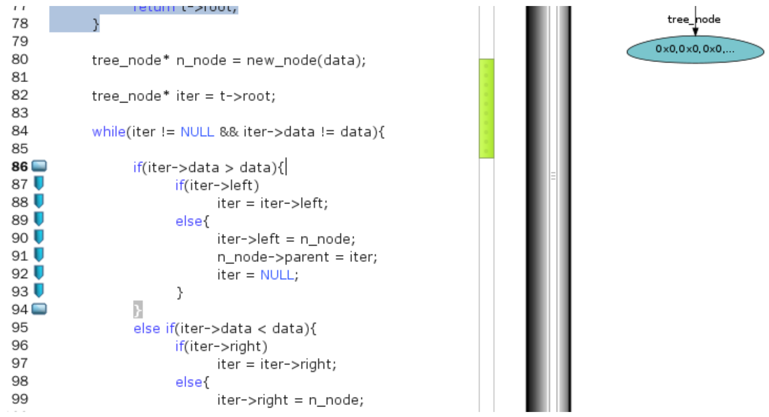


```
        unsigned int key = 0;

        for(i=0;i<15;i++){//INSERTIONS
                if(i==10){
                        key = rand() % 100;
                       insert(t,key);
                }
                insert(t,rand() % 1000);
        }

        tree_node* s_node = search(t,key);//SEARCH
```

Figure 13. Stepping through a logical loop iteration.

Moreover during syntax navigation various kinds of parsing information is used to ease the process of debugging and development. One such example is the selective display of variables that are so far (up to the current syntax structure) *referenced*, rather than simply displaying all set of local variables that usually include uninitialised data.

In addition now to standard source-line breakpoints, *syntax structure breakpoints* can be set that can more conviniETLY express the programmers' entire region of interest.



```
        return t->root;
    }

    tree_node* n_node = new_node(data);

    tree_node* iter = t->root;

    while(iter != NULL && iter->data != data){

        if(iter->data > data){
            if(iter->left)
                iter = iter->left;
            else{
                iter->left = n_node;
                n_node->parent = iter;
                iter = NULL;
            }
        }
        else if(iter->data < data){
            if(iter->right)
                iter = iter->right;
            else{
                iter->right = n_node;
```

Figure 14. Setting a regional breakpoint.

Of course *all* the standard navigational commands of the legacy *line by line* traversal, *step-in*, *step-out*, *step-over* are also supported, through the *forward* button depicted above.

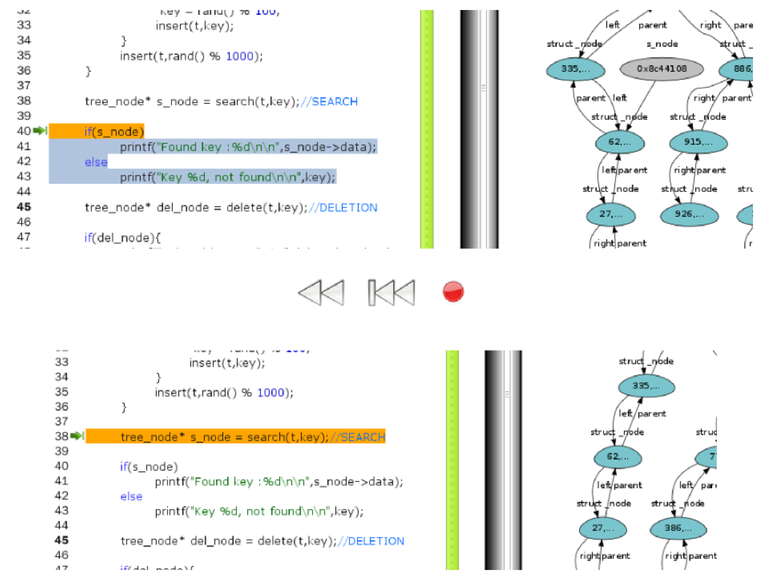
7.2 EXECUTING BACKWARDS IN TIME

As we can see in Figure 15 upon invocation of the record facility by the user, the ability to step backwards in time is enabled. Syntax navigation is also enabled in this mode, providing total control over program monitoring. In the example that follows, the programmer steps backwards from an if statement to review the invokation of the *bst search function*.

As above the equivalent reverse line by line navigation commands are supported via the *backward* button.

7.3 DATA - DISPLAYING AND EDITING

Now focusing on our data-displaying and editing facilities we examine the inner workings of the aforementioned bst *search* function. In the figures below, the iteration inside a bst tree is depicted, while the programmer is monitoring in execution time the evolution of data on the graph and through menus or even editing values as he goes along. The graph depth is variable and can be arbiteraly defined by the user,



```
                key = rand() % 100;
                insert(t,key);
        }
        insert(t,rand() % 1000);
    }

    tree_node* s_node = search(t,key);//SEARCH

    if(s_node)
        printf("Found key :%d\n\n",s_node->data);
    else
        printf("Key %d, not found\n\n",key);

    tree_node* del_node = delete(t,key);//DELETION

    if(del_node){
```

```
                insert(t,key);
        }
        insert(t,rand() % 1000);
    }

    tree_node* s_node = search(t,key);//SEARCH

    if(s_node)
        printf("Found key :%d\n\n",s_node->data);
    else
        printf("Key %d, not found\n\n",key);

    tree_node* del_node = delete(t,key);//DELETION
```

Figure 15. Stepping backwards, in time. Reversing the execution flow.

as well as the zoom factor for the entire data display through scrolling of the mouse wheel. The data display context menu, provides several facilities, including showing/hiding address values and evaluation of user-defined expressions among other things. All these facilities are illustrated respectively from Figure 16 to Figure 20.

7.4 DEBUGGING CONSOLE INTERACTION

Let's now take a look to our versatile *debugging console* interaction, that supports among other things syntax error reporting, data displaying in graphs, full scripting of the environment's API and direct python to c calls for the project being developed. This is where the main interaction with our general purpose debugging extention language takes place. The programmer can monitor and control literally every aspect of the environment from this very console and intrepreter. Some examples of this interaction follow, from Figure 21 throughout Figure 24.

7.5 MULTI-LANGUAGE SCRIPTING

As we saw earlier python to c calls apart from being a versatile tool for debugging in our environment, can also be used for unit-testing and multi-language software developement. Elaborating on this idea our IDE provides seamless inclusion of python source files to our c projects that can be invoked right upon entrance to our projects' main function. By convencion this happens automatically if a *'main.py'* file is present in the project

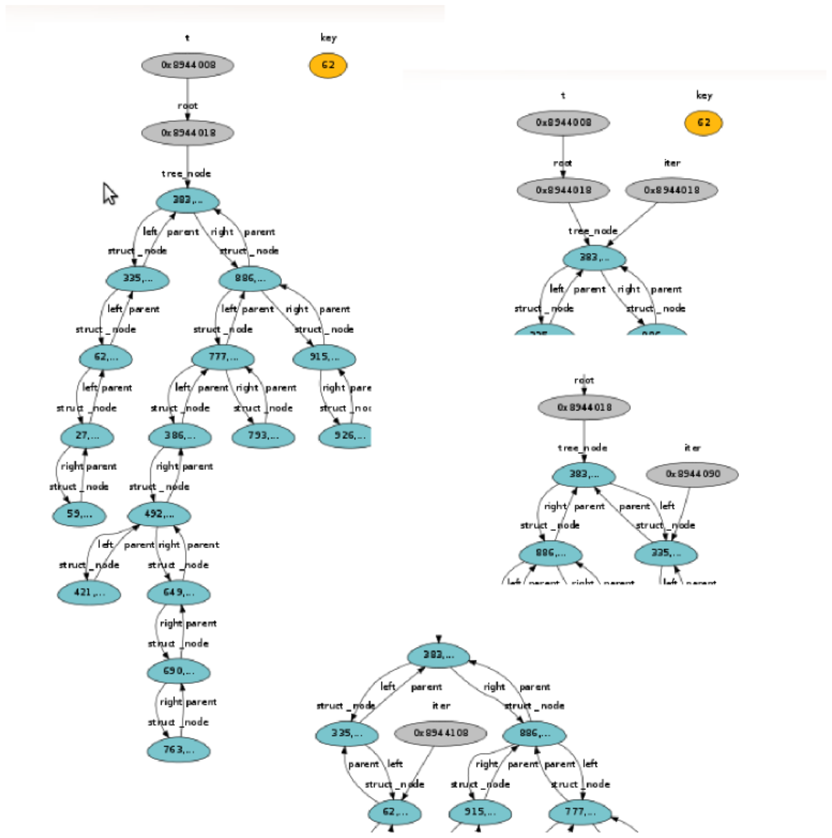


Figure 16. Iterating throught the BST data-structure for key integer value.

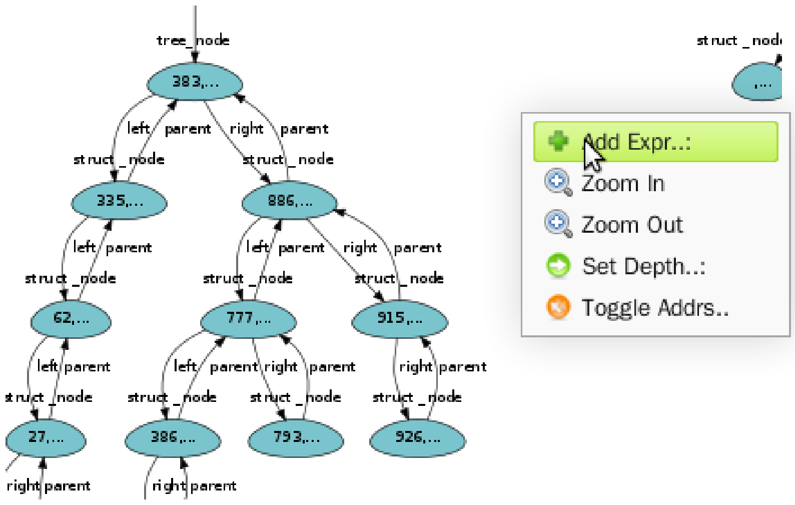
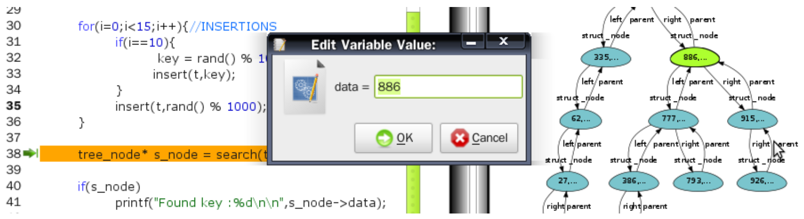


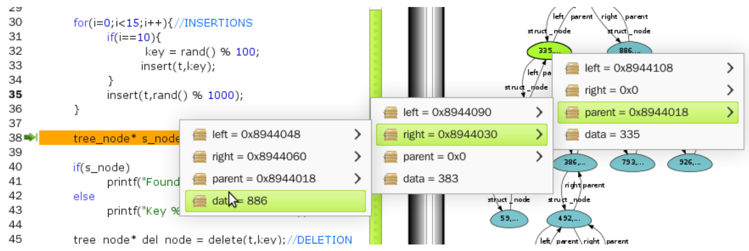
Figure 17. Data Display context-menu.

Moreover these scripts have access to the *extentibility API of the environment*, a feature that when combined with our multi-language development approach can provide *on-line* extentibility of our IDE both in c and in python during normal program execution.



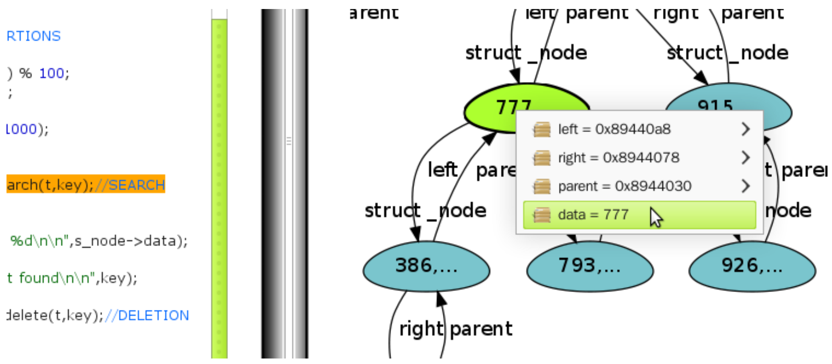
```
	for(i=0;i<15;i++){//INSERTIONS
		if(i==10){
			key = rand() % 1
			insert(t,key);
		}
		insert(t,rand() % 1000);
	}

	tree_node* s_node = search(t

	if(s_node)
		printf("Found key :%d\n\n",s_node->data);
```

Figure 18. Editing Data during execution time, through the data graph



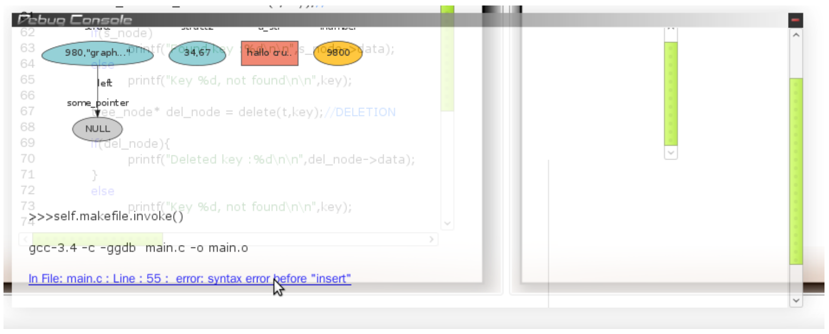
```
	for(i=0;i<15;i++){//INSERTIONS
		if(i==10){
			key = rand() % 100;
			insert(t,key);
		}
		insert(t,rand() % 1000);
	}

	tree_node* s_node

	if(s_node)
		printf("Found
	else
		printf("Key %

	tree_node* del_node = delete(t,key);//DELETION
```

Figure 19. Data graph menu navigation



```
RTIONS

) % 100;
;

1000);

arch(t,key);//SEARCH

%d\n\n",s_node->data);

t found\n\n",key);

delete(t,key);//DELETION
```

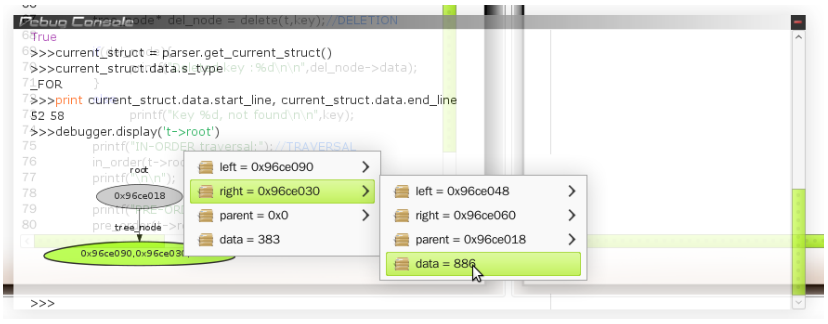
Figure 20. Zooming in a specified region in the data graph



```
	if(s_node)
		printf("Found key :%d\n\n",s_node->data);
	else
		printf("Key %d, not found\n\n",key);

	tree_node* del_node = delete(t,key);//DELETION

	if(del_node){
		printf("Deleted key :%d\n\n",del_node->data);
	}
	else
		printf("Key %d, not found\n\n",key);
>>>self.makefile.invoke()

gcc-3.4 -c -ggdb  main.c -o main.o

In File: main.c : Line : 55 :  error: syntax error before "insert"
```

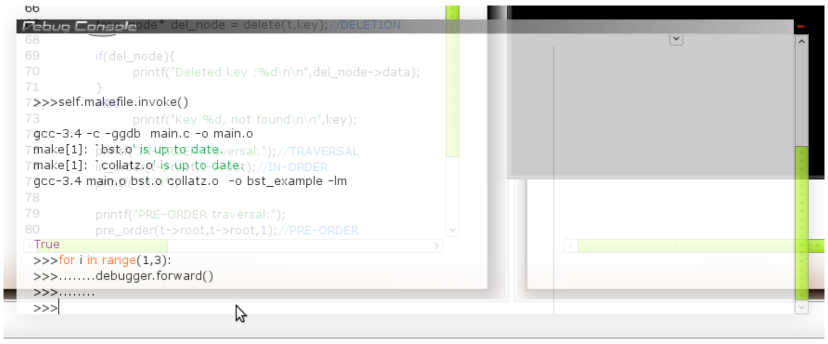
Figure 21. A snapshot of syntax error reporting and navigation through links

Figures Figure 25 to Figure 27 show two extention plugins in action, that use the projects' c funtions, the extentibility API, python and



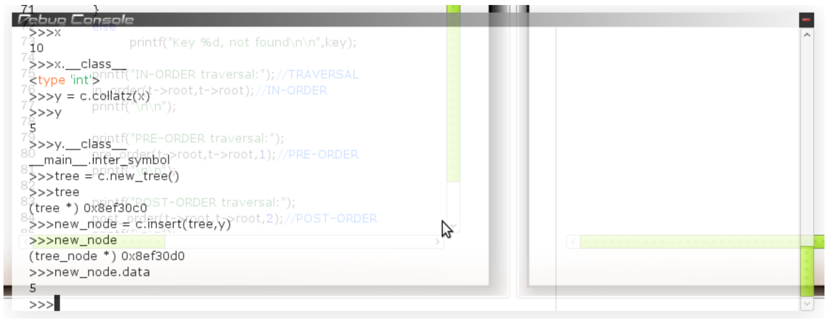
```
True
>>>current_struct = parser.get_current_struct()
>>>current_struct.data.s_type
_FOR
>>>print current_struct.data.start_line, current_struct.data.end_line
52 58
>>>debugger.display('t->root')
>>>
```

Figure 22. Graph expression evaluation, on the console.



```
>>>self.makefile.invoke()
gcc-3.4 -c -ggdb  main.c -o main.o
make[1]: `bst.o' is up to date.
make[1]: `collatz.o' is up to date.
gcc-3.4 main.o bst.o collatz.o  -o bst_example -lm
True
>>>for i in range(1,3):
>>>........debugger.forward()
>>>........
>>>
```

Figure 23. An example of code scripting debugging control.



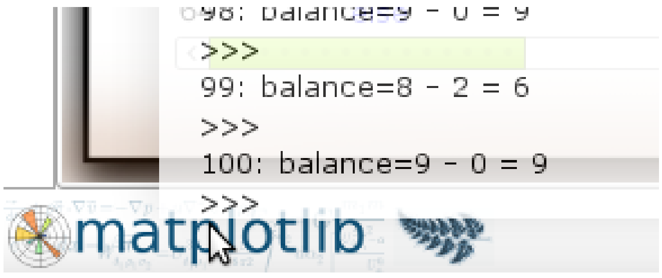
```
>>>x
10
>>>x.__class__
<type 'int'>
>>>y = c.collatz(x)
>>>y
5
>>>y.__class__
__main__.inter_symbol
>>>tree = c.new_tree()
>>>tree
(tree *) 0x8ef30c0
>>>new_node = c.insert(tree,y)
>>>new_node
(tree_node *) 0x8ef30d0
>>>new_node.data
5
>>>
```

Figure 24. Direct calling of c functions and code from python.

external libraries to visualise results of two simple *Collatz* sequence experiments.

7.6 project management and interfaces

Last but not least we present some of the peripheral interfaces that were developed to ease the usage of our IDE. From Figure 28 to Figure 30 we can see among other things, our *online documentation searching sys-*



```
698: balance=9 - 0 = 9
>>>
99: balance=8 - 2 = 6
>>>
100: balance=9 - 0 = 9
>>>
```

Figure 25. Two new buttons created, by invoking the *on-line* extentibility API of Misha.

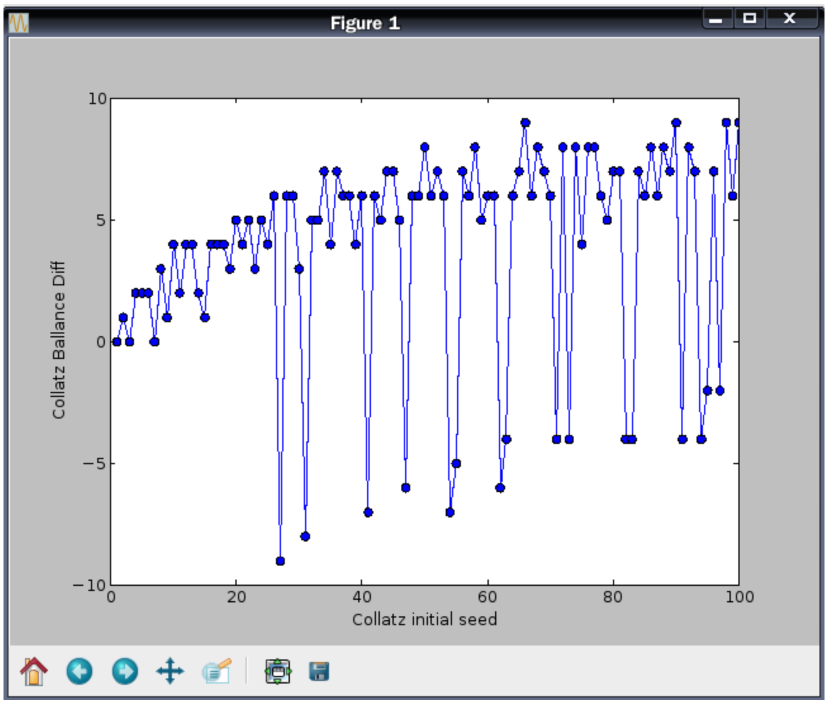


Figure 26. The first button's callback function uses c, python and external libraries to plot the results

tem, the automatic *dependency management* widget, and our bottom bar buttons from where our speech commands can be enabled.

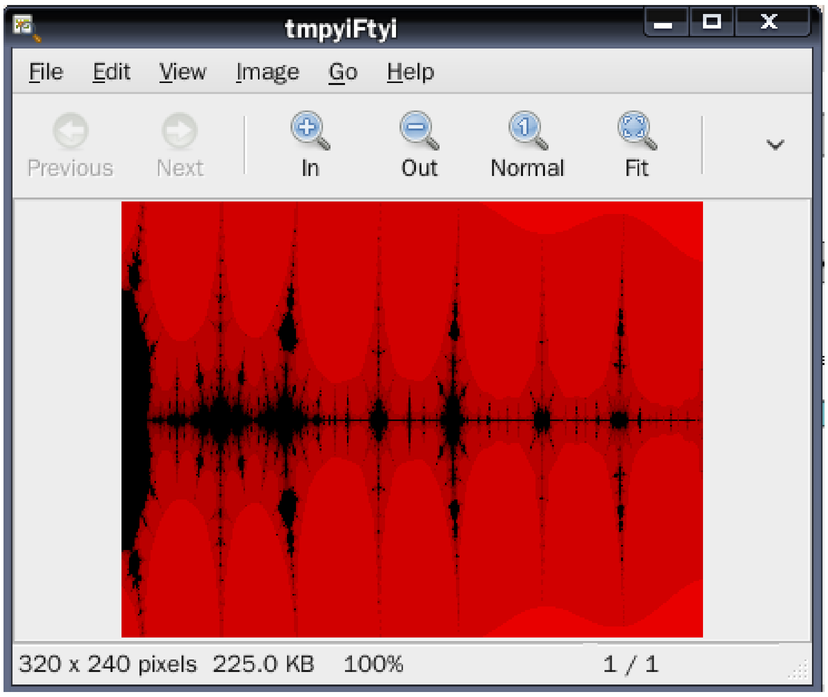
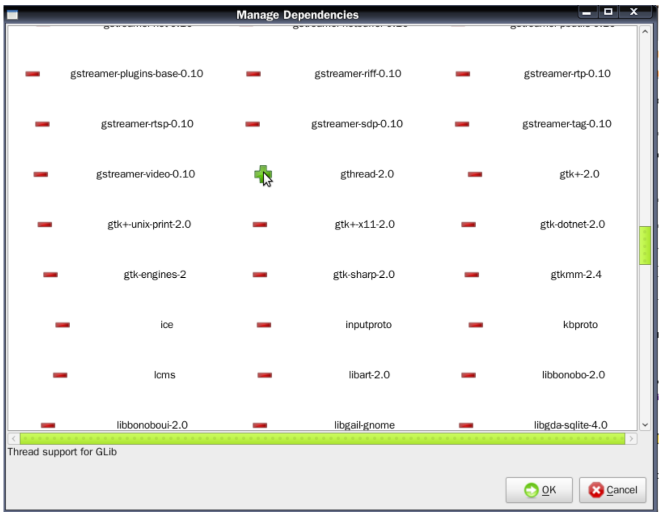


Figure 27. The second button's callback function uses c and python to visualise a collatz fractal



▬	gstreamer-plugins-base-0.10	▬	gstreamer-riff-0.10	▬	gstreamer-rtp-0.10
▬	gstreamer-rtsp-0.10	▬	gstreamer-sdp-0.10	▬	gstreamer-tag-0.10
▬	gstreamer-video-0.10	✚	gthread-2.0	▬	gtk+-2.0
▬	gtk+-unix-print-2.0	▬	gtk+-x11-2.0	▬	gtk-dotnet-2.0
▬	gtk-engines-2	▬	gtk-sharp-2.0	▬	gtkmm-2.4
▬	ice	▬	inputproto	▬	kbproto
▬	lcms	▬	libart-2.0	▬	libbonobo-2.0
▬	libbonoboui-2.0	▬	libgail-gnome	▬	libgda-sqlite-4.0

Thread support for GLib

Figure 28. The dependency management widget

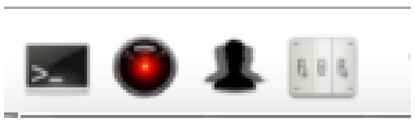
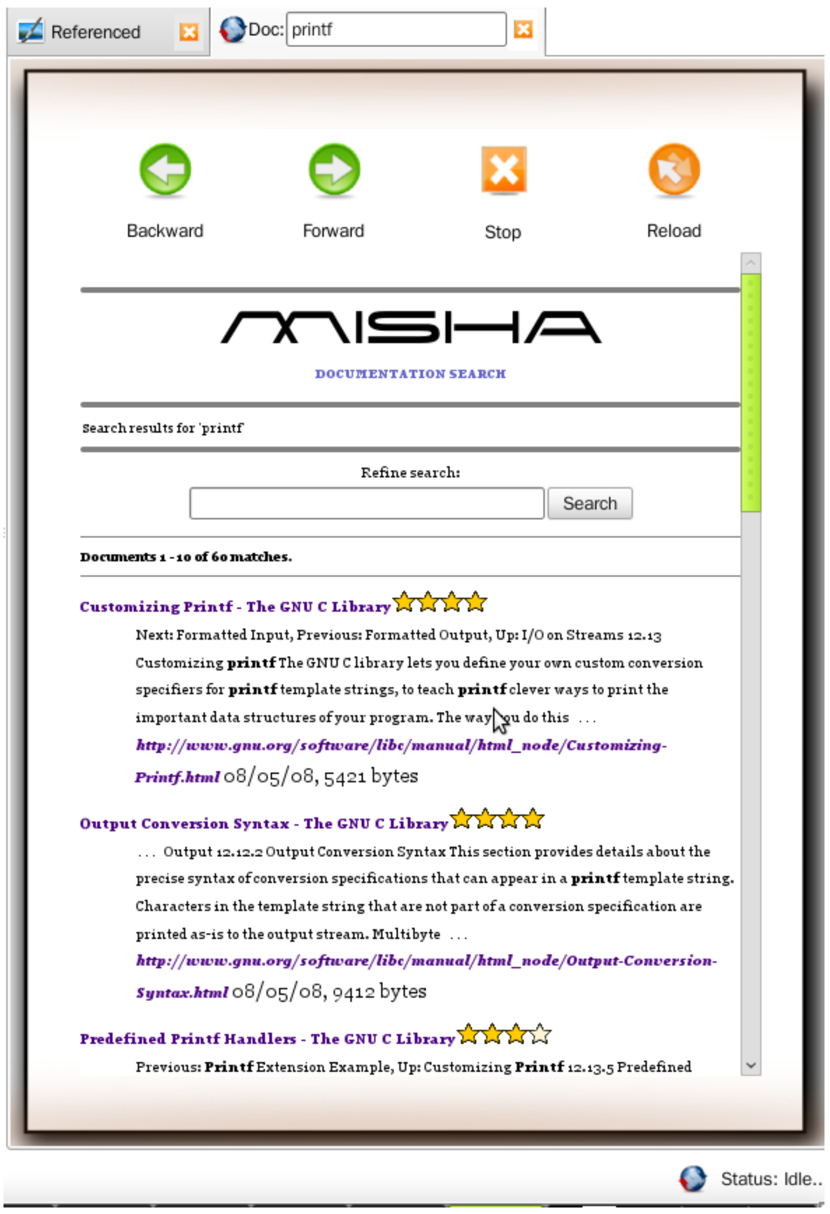


Figure 29. Our bottom bar with our voice commands controls



Search results for 'printf'

Documents 1 - 10 of 60 matches.

Customizing Printf - The GNU C Library

Next: Formatted Input, Previous: Formatted Output, Up: I/O on Streams 12.13 Customizing **printf** The GNU C library lets you define your own custom conversion specifiers for **printf** template strings, to teach **printf** clever ways to print the important data structures of your program. The way you do this ...

http://www.gnu.org/software/libc/manual/html_node/Customizing-Printf.html 08/05/08, 5421 bytes

Output Conversion Syntax - The GNU C Library

... Output 12.12.2 Output Conversion Syntax This section provides details about the precise syntax of conversion specifications that can appear in a **printf** template string. Characters in the template string that are not part of a conversion specification are printed as-is to the output stream. Multibyte ...

http://www.gnu.org/software/libc/manual/html_node/Output-Conversion-Syntax.html 08/05/08, 9412 bytes

Predefined Printf Handlers - The GNU C Library

Previous: **Printf** Extension Example, Up: Customizing **Printf** 12.13.5 Predefined

Figure 30. The mozilla client plugin with our online documentation searching system.

8 FUTURE WORK - DISCUSSION

Introducing new facilities and interfaces for debugging systems, was only the first step of our journey. As we have shown so far *Misha R.ide* is a source-code base that can both be expanded to meet the needs of a full-blown IDE *and* provide a framework for future research proposals on development tools, interfaces and programming languages.

In this last chapter we examine more closely the future potential of our work.

8.0.1 *Supporting legacy debugging and programming facilities*

We would like to see our system integrate some additional legacy debugging techniques that are not currently supported by this first version of *Misha*. These include *cathpoints*, *watchpoints* and *tracing* that although are covered by our python bindings of the *gdb machine interface* have not yet been integrated with the rest of the system.

In terms of common programming facilities that appear in most mainstream IDEs, Misha should certainly support, *syntax completion*, *profiling*, *refactoring*, *unit-testing*, *code revisioning* and a *user-interface design* solution. With these features added we believe that our IDE will not only be an innovative tool but rather a complete state of the art development solution.

8.0.2 *High-level debugging enhancements*

Now for our high-level debugging system we would like to focus on ideas that *further intergrate our syntax-parser* with the rest of the environment.

One such intergration for example can result in a *memory optimization* solution for our reverse debugging system. Currently the recording mechanism keeps track of data changes occuring on a instruction-level basis. This is a rather redundant approach since the changes that are really relevant to the programmer occur on the statement level (that can span many machine instructions wide) and only for the part of the execution stack that resides inside the programmers's source code, excluding all external library calls. This kind of optimization that can easily be developed with the augmented debugging information of our parsing tree can greatly reduce the memory footprint of the replay debugging system.

We will also like to see our system expanding to the thrieving field of *multi-threaded debugging*. As mentioned earlier the basic operations are already implemented for such an expantion, but there are other posibilities as well. Static code analysis for example that uses our versetile parser can be implement to automatically deduce various *race conditions* between different threads.

In the same line of thinking, our data display system can be expanded to incorporate *call-graph representations* from which a more intuitive interface for setting breakpoints can emerge.

Finally the core implementation of our parser can be enhanced to read source code *incrementally*, giving the possibility among other things to graphically monitor source code changes as they happen.

8.1 PROBING FURTHER..

Apart from the experience and knowledge gained in the course of this work, a lot of new ideas that transent debugging systems have emerged. Especially the multi-language testing and development facilities that we have developed, made us think of the possibility of integrating more than two languages that seamlessly interconnect (without the programmer's intervention through glue-code) in a single and unified environment. Without of course the need of a common intermediate representation.[1].

We have already begun working towards this direction, that does not only aim to unify language environments but also different execution platforms such as the linux desktop and the web. We believe that *Misha R.ide* will proove to be a creative source-code base for developing such a *Multi-language, multi-purpose and free environment*... So we urge you to stay tuned ;)

1 as in .net or jython environements for example where there is a common byte-code backend

BIBLIOGRAPHY

[1] Richard M. Stallman. Emacs: The extensible, customizable display editor. *ACM Conference on Text Processing*, 1981. (Cited on pages ix and 3.)

[2] Imperial College Department of Computing. Free on-line dictionary of computing. http://foldoc.org/Grace+Hopper. (Cited on page 1.)

[3] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, pages 1–10, 2002. (Cited on page 1.)

[4] Narasimhaiah Gorla, Alan C. Benander, and Barbara A. Benander. Debugging effort estimation using software metrics. *IEEE Trans. Software Eng.*, 16(2):223–231, 1990. (Cited on page 2.)

[5] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-83595-9. (Cited on page 2.)

[6] V.A. Vazioulin. *The Logic of History (in greek)*. Ellhnika Grammata, 2004. Translation-Diligence-Comments: Dimitris Patelhs. (Cited on page 2.)

[7] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference, 1983. ISBN 0139376992. (Cited on page 3.)

[8] David MacKenzie Julia Menapace, Jim Kingdon. *The "stabs" debug format*. Cygnus Support, 2004. (Cited on pages 7 and 21.)

[9] Stan Shebs John Gilmore. *GDB Internals*. Cygnus Solutions, 2004. (Cited on page 7.)

[10] Stan Shebs Richard Stallman, Roland Pesch. *Debugging with GDB*. Gnu Press, 2003. (Cited on pages 7, 8, 12, and 29.)

[11] Andreas Zeller. *Debugging with DDD*. Gnu Press, 2004. (Cited on pages 8 and 12.)

[12] AdaCore. Using the gnat programming studio. `http://libre.adacore.com/wp-content/files/auto_update/gps-docs/The-Data-Window.html`. (Cited on page 8.)

[13] Michael Golan and David R. Hanson. Duel - a very high-level debugging language. 1993. (Cited on pages 12 and 16.)

[14] Mireille Ducassé and Anna-Maria Emde. Opium: a debugging environment for prolog development and debugging research. *SIGSOFT Softw. Eng. Notes*, 16(1):54–59, 1991. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/126496.126500. (Cited on pages 12 and 16.)

[15] Ziya Aral and Ilya Gertner. High-level debugging in parasight. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 151–162, New York, NY, USA, 1988. ACM. ISBN 0-89791-296-9. doi: http://doi.acm.org/10.1145/68210.69230. (Cited on page 15.)

[16] Jose C. Cunha, Joao Lourenco, and Vitor Duarte. Using ddbg to support testing and high-level debugging interfaces. 1995. (Cited on page 15.)

[17] Joel Huselius. Debugging parallel systems: A state of the art report. Technical report, 2002. (Cited on page 15.)

[18] D. Y. Cheng. Hdb-a high level debugging. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 568–573, New York, NY, USA, 1989. ACM. ISBN 0-89791-341-8. (Cited on page 16.)

[19] Cristina Cifuentes, Trent Waddington, and Mike Van Emmerik. Computer security analysis through decompilation and high-level debugging. In *In Proceedings of the Workshop on Decompilation Techniques*, pages 375–380. IEEE Press, 2001. (Cited on page 16.)

[20] Gdb Development Team. Pythongdb. `http://sourceware.org/gdb/wiki/PythonGdb`, 2009. (Cited on pages 17, 18, 24, and 29.)

[21] Free Software Foundation. High-priority projects. http://www.fsf.org/campaigns/priority.html, 2009. (Cited on page 17.)

[22] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *In ISCA*, pages 284–295, 2005. (Cited on page 17.)

[23] Steven Allen Lewis. Techniques for efficiently recording state changes of a computer environment to support reversible debugging, 2001. (Cited on page 17.)

[24] Tankut Akgul and Vincent J. Mooney. Instruction-level reverse execution for debugging, 2002. (Cited on page 17.)

[25] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 567–576. ACM, 2007. (Cited on page 17.)

[26] Andrew Begel and Susan L. Graham. An assessment of a speech-based programming environment. In *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, pages 116–120. IEEE Computer Society, 2006. (Cited on page 17.)

[27] Gdb Development Team. Gdbreversible. http://sourceware.org/gdb/wiki/ReversibleDebugging, 2009. (Cited on pages 18 and 24.)

[28] Teawater. Gdb record patch. http://sourceforge.net/project/shownotes.php?release_id=604719, 2009. (Cited on page 18.)

[29] Visual Studio Developer Center. Visualisers. http://msdn.microsoft.com/en-us/library/zayyhzts.aspx, 2009. (Cited on page 18.)

[30] Wolfram Research. Instant high-level debugging. http://www.wolfram.com/technology/guide/InstantHighLevelDebugging/, 2009. (Cited on page 18.)

[31] Dan Burque. Time travel made possible with eclipse. http://www.eclipsecon.org/2006/Sub.do?id=84, 2006. (Cited on page 18.)

[32] Bill Lewis. Debugging backwards in time. http://video.google.com/videoplay?docid=3897010229726822034&ei=t_17SpSUAaac2wKx9_XXAw&q=odb+bill+debugging, 2006. (Cited on page 19.)

[33] Ds-emedia. Jbixle, high-level debugger. http://www.jbixbe.com/, 2006. (Cited on page 19.)

[34] Berkeley. Harmonia project. http://harmonia.cs.berkeley.edu/harmonia/index.html, 2009. (Cited on page 22.)

[35] Stefan Seefeld. Synopsis: A source-code introspection tool. http://synopsis.fresco.org/index.html, 2009. (Cited on page 22.)

[36] Tom Tromey. Interview: Gcc as an incremental compile server. http://harmonia.cs.berkeley.edu/harmonia/index.html, 2007. (Cited on page 22.)

[37] Brad King Kitware. Gcc xml, extention. http://www.gccxml.org/HTML/Index.html, 2009. (Cited on page 22.)

[38] John Ellson. Graphviz, graph visualization software. http://www.graphviz.org/, 2009. (Cited on page 23.)

[39] Dennis M. Ritchie Brian W. Kernighan. *The C Programming Language, Second Edition*. Prentice Hall, 1998. (Cited on pages 27 and 28.)

[40] Guido van Rossum. Python programming language. http://python.org/, 2009. (Cited on pages 27 and 30.)

[41] Jutta Degener Jeff Lee, Tom Stockfisch. Ansi c yacc grammar. http://www.quut.com/c/ANSI-C-grammar-y.html, 1998. (Cited on page 28.)

[42] Will Estes Vern Paxson and John Millaway. *Lexical Analysis With Flex*. University of California, 2007. (Cited on page 28.)

[43] Richard Stallman Charles Donnelly. *Bison – The Yacc-compatible Parser Generator*. Gnu Press, 2009. (Cited on page 28.)

[44] Roland McGrath Sandra Loosemore, Richard Stallman. *The GNU C Library Reference Manual*. Gnu Press, 2007. (Cited on page 28.)

[45] Gnome documentation library. http://library.gnome.org/devel/glib/, 2009. (Cited on page 28.)

[46] David Beazley. Ply (python lex-yacc). http://www.dabeaz.com/ply/, 2008. (Cited on page 29.)

[47] Eleftherios Koutsoos, Eleftherios Koutso Os, Stephen C. North, Intsparcd Compsparc, Sparcascode Sparccm, Sparcmcemit Sparcmcode Sparcasemit, and Intsparc Intnull Intnulld. Drawing graphs with dot, 1993. (Cited on page 29.)

[48] Paul Smith Richard Stallman, Roland McGrath. *GNU Make – A Program for Directing Recompilation*. Gnu Press, 2007. (Cited on page 30.)

[49] Tollef Fog Heen. Pkgconfig. http://pkg-config.freedesktop.org/wiki/, 2009. (Cited on page 31.)

[50] The GNOME Project and PyGTK Team. Pygtk: Gtk+ for python. http://www.pygtk.org/, 2009. (Cited on page 32.)

[51] Gtkmozembed: Gtk mozilla embedding widget. http://www.mozilla.org/unix/gtk-embedding.html, 2009. (Cited on page 32.)

[52] ht://Dig Group. Htdig www search engine software. http://www.htdig.org/, 2009. (Cited on page 32.)

[53] David Huggins-Daines. Pocketsphinx - sphinx for handhelds. http://www.speech.cs.cmu.edu/pocketsphinx/, 2009. (Cited on page 32.)

[54] Alexandre Moore. nuovext icon theme for gnome. http://nuovext.pwsp.net/, 2009. (Cited on page 32.)

[55] Jeffrey C. Lagarias. The 3x+1 problem and its generalizations. *American Mathematical Monthly*, 92:3–23, 1985. (Cited on page 33.)

[56] Papoylias Nikos. Misha research ide. https://launchpad.net/misha, 2009. (Cited on page 33.)

A

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with

generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified

Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this

License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.