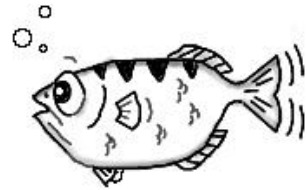


NICK PAPOYLIAS

Technical University of Crete
Chania, Greece
Email: npapoylias@isc.tuc.gr



Nick Papoylias · Technical University of Crete · Chania, Greece

GDB development team,
gdb@sourceware.org

cc: *DDD development team:* zeller@gnu.org, apg@users.sf.net
The GNU Project: gvc@gnu.org

Chania, May 26, 2007

Subject: High-Level debuggers, facilities and interfaces

Dear developers of the GNU debugger,

My name is Nick Papoylias, and I am an undergraduate student in the Computer Engineering Department at the Technical University of Crete (Chania, Greece www.ece.tuc.gr). I am interested in initiating a free software project involving debugging systems - which I'll soon describe to you in brief - that would either be part of my graduating thesis, or an independent project altogether (this depends on the response I will have from the part of my faculty). My resources including bibliography, articles, related work e.t.c are relatively low which is why I decided to contact you, after all "re-inventing the wheel" is not part of anyone's intentions. Also I would like to be able to contact people doing similar work in other faculties, so I would greatly appreciate your help in this area too, if it happens that such information is available to you. It is maybe redundant to say that my interest in debugging, is greatly inspired by your work, as well as that of the DDD people, and that recent initiatives including "reverse debugging" in GDB's and

GNU's "high-priority" list, convinced me that there is scientific and community interest for related work in this field.

My research work, by the time of this writing, can be best described by the general title "*High-Level Debuggers - Facilities and Interfaces*". The "**High-Level**" part of the title is a loan from the programming-language world, the usage of which can be substantiated by the fact that Gdb and similar "Low-Level" or "Source-Level" debuggers have an input-language of their own - Gdb/MI in your case - which one can use to get access to the "facilities" of the "Low-Level" debugger. The main idea here is that someone can use Gdb (or another source-level debugger) as a back-end, not just to provide access in the "low-level" facilities, but as a means to built "higher-level" structures using gdb's input language as well as other systems, the combined effort of which can provide the "higher-level" functionality. User-defined commands in gdb's features is an excellent macro-equivalent (in terms of programming languages) for the above line of thinking.

The "**Interfaces**" part mentioned in the title refers to the new input/output and overall graphical user interface features that can be built on-top of these "higher-level" facilities, giving the programmer new ways to form his development cycle. A thorough investigation of the role of debugging, in engineering as a whole and in software engineering in particular, is needed for the final suggestions and implementations in this part of the project.

Before going any further to describe the specific, facilities and interfaces, that by the time of this writing, I intent to include in the implementation, I have to mention that the final goal of this project is both the theoretical investigation of the field, as well as *a simple demonstrative IDE targeting the C programming language using Gdb as it's back-end and intermediate systems*. The choice of language, although not very relevant for this brief description, was not a tough one, since the target language had to be clear syntax-ed, widely used and understood, fully supported by gdb, and one that would clearly help demonstrate the new debugging features and their implications in the development cycle. If the project turns to be both creative and fun for the community, I am sure that implementations for other languages will follow. But I don't want to get ahead of myself. A list of specific facilities/interfaces and their description in general terms follows:

- ***Syntax-Aware Debugging:*** This feature is intended to be the workhorse for the overall project. The implication here is that by using a parser to analyze the source code, debugging can take place in terms of specific syntactic structures (expressions, statements, conditionals, loops, functions, data-structures, modules/files) in the syntax-level and not in the source-level, having different "template" information readily available to the user according to the nature of each structure of the target language. The user will be able to pinpoint syntax-structures of interest as a whole, and not just as source-lines, and debugging can take place both as stepping through a "logical-unit" of evolution and as watching the evolution of the syntax structure over time, freezing execution when needed. Of course single stepping through source lines, will be available to the programmer at all times, in addition to stepping through the structures. Conditional debugging, as well as specific user-defined in-structure information, will be supported. The evolution of data will be monitored by a "generalization" of watchpoints.
- ***Data and Syntax Displaying:*** Greatly inspired by the work on DDD, Data Displaying will be an essential part of the "interface" for the "high-level" debugger, incorporating new features as that of syntax displaying and new interaction schemes between the data/syntax view and the source code of a project, "Debugging-In-Human-Mode" will be supported under the programmer's request, that is to say that execution time will be slowed down between breakpoints, so that the evolution of data and syntax can be easily monitored in the display, giving the user enough time to "interrupt" the program precisely when needed. Such a feature, can have a great impact on source-level documentation and education, since pre-defined debugging schemes (presenting the evolution of algorithms and data) running under "Human-Mode" can be easily used to describe and demonstrate functionality with a click of a button, text and speech annotations between "evolution-steps" can be used to document execution, having a more-clear context than source-level comments.
- ***Using a Debugging Extension Language:*** Controlling the Debugger and Benchmarking Code through Guile. Now, this part is tricky. Gener-

ating benchmarks and data-analysis charts, for debugging as well as demonstration purposes is usually handled by a dummy-procedure in the source language. When the source language is relatively "low-level" and when all the different functionality of a program needs to be checked and double-checked this can cause a lot of frustration. Command line debuggers such as gdb incorporate syntax for conditionals and other program structures such as debugging statements, which by their very nature and intention-of-usage do not constitute a full-blown language that can handle all such cases, this is where Guile comes into play. Having Guile being the equivalent of "Gdb/MI" for the high-level debugger, as well as being able to automatically generate bindings of the user's program for itself, can make programming and debugging a lot more fun! The added implication here is that "high-level" debuggers can encompass "higher-levels" of abstraction, through an extension language, giving rise to even more sophisticated systems. Programmers will even be encouraged to use extension-languages and especially Guile, not just for debugging, but for structuring their programs as well, taking advantage of the merits of an "emacs-like" architecture.

- **Graphical Aids and Input facilities:** 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 from the very nature - when used in IDEs - of "monitoring" and "handling" a running process tend to demand their share of the desktop. Having a second monitor while programming, for debugging and documentation purposes, seems to address the problem, but not a lot of people fancy (or want to spent money on) a second monitor. Taking advantage of the newly presented graphic capabilities of beryl and/or compiz for usability inside-programs and not just outside them (in window-managers) can be a great help in this area (especially for positioning widgets in a 3d-plane, and for using transparencies), I don't know though if any work is being done towards these goals (maybe things like that technically need to be propagated downwards - in gtk+ - in order to be implemented). For the options part presented to the programmer though, there is an eas-

ily implementable solution, since from one hand, the command line of the extension language will be able to handle a lot of functionality through typing (automatic completion and shortcuts), from the other hand debugging systems in general can greatly benefit from a "voice-command" interface, since a number of distinct frequently used commands like "run", "stop", "pause", "watch <variable-name>" , "human mode" e.t.c can be readily recognized by today's free speech-recognition technology. (the usefulness and/or irritation - apart from accessibility issues - of a debugger that talks "back" to you¹, have to be investigated)

- **Related features from community proposals:** Now, it would prove wrong not to include (and also try to improve) in this project, related community proposals that exhibit "higher-level" functionality, some of which are of significant importance for "higher-level" debuggers, such as the fore-mentioned "reverse-debugging" facility, with the add-on capability - in the context of syntax aware debugging - to step-backwards a whole syntax "evolution step". Bookmarks in general and the usage of forking for debugging purposes, can be used as a building block for many interesting feats, including the parallel evaluation of two instances of a project, one of which has been recently patched at runtime with a simple change (with the added ability to discard or keep the changes after evaluating)². In addition "Tracing" can be used for "mainstream" debugging (not just for embedded and/or execution-time critical applications), since it is a very common practice during the development cycle and in-between bug-fixes to execute the program outside the debugger concentrating in the real-time logical (or not) execution of the program. Having the debugger operate in "stealth mode" during these runs (effectively transforming breakpoint operations to tracepoint operations as well as automatically collecting user specified bookmarks to return later) can prove extremely useful in this

¹Actually, speech annotations in debugging sessions, is a form of a "talking-back" debugger, although in this case it is much more like a "talking-back" to yourself scenario, with all the psychiatric implications involved.

²Keeping track of source level changes, as well as the debugger's state even between development sessions is of great importance for the development cycle - the interaction of the debugger with a revision system has to be investigated.

context. Not to mention here that the combined usage of bookmarks and tracing, can prove valuable in addressing memory-related issues of "reverse-debugging" where memory-consuming bookmarks can become "sparser" with tracing filling the missing pieces in between³. To sum up, while a decade ago it would prove inefficient to watch a program so much "closely", storing data and state every once in a while providing "higher-level" facilities, today with the advancement of horsepower and storage capabilities, it would not only prove efficient but also quite useful.

I wouldn't want to go any further with features and details (such as the potential interaction of high-level debuggers with profilers and similar systems), mostly out of respect for your time, but these are in general terms the driving thoughts behind this effort. I would greatly appreciate - besides providing me with related bibliography on the subject - on your commenting on the overall effort both technically and in respect with it's potential usefulness for the community. I thank you in advance.

Yours sincerely,

Nick Papoylias

*P.S.: "Who cares (or even doubts) about the "world domination" of free software, when we can feed with programs and data the whole galaxy and beyond ... ? Us, taking advantage of ALL of today's technical capabilities of computer networks and producing in abundance, is merely a technicality... which nevertheless suffices to send "proprietary software" in the museum of history. What really distinguishes us though from our proprietary past is that we treat our neighbor as a creative human-being, and not as a "dear" costumer. It is beginning to look clear now that Free Software NEEDS Free Societies."*⁴

³Of course execution will be able to rewind only in bookmark locations.

⁴Abstract from the unpublished manuscript of *pars(e)val*, one of the many young knights of the λ -calculus.