High-Level Debugging Facilities and Interfaces. Design and Developement of a Debug-Oriented I.D.E.

Nick Papoylias

High-Level Debugging Facilities and Interfaces. Design and Developement of a Debug-Oriented I.D.E.

Nick Papoylias[1]

Technical University of Crete, Chania, Greece,
npapoylias@isc.tuc.gr,
WWW home page: http://www.softnet.tuc.gr/~whoneedselta/misha

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 free-software research 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, *Misha*. Misha, 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.

1 Introduction

1.1 Problem Statement

Today's advancement in IDEs although constantly offering new programming tools and levels of sophistication, has left debuggers where they were a decade or more 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 technologies in the debugging backend often offer 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.

Given the importance of software monitoring and debugging as it is expressed in scientific publications concerning *effort estimation*[**?**] and *project management* which on average assert that testing and debugging cover roughly 50 % percent of the development time [**?**], we propose - both theoretically and technologically - a possible route for the evolution of debuggers that would hopefully meet the current needs of software engineering.

2 Related Work

2.1 Published Work

As far as published work is concerned 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. That is the case with *Duel* [**?**] and *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. 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.

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.

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 [**?**]. In terms of published work some of these approaches can be found in [**?**], [**?**] and [**?**].

Last but certainly not least for reasons that we will discuss below when dealing with our syntax parser, our work is also related to the work of the Harmonia Project in Berkeley (see [**?**] and [**?**]) which deals with *high-level interactive software development*, *Language-Aware programming tools* and *programmer-computer interaction* although to our knowledge their work has yet to be expanded in debugging.

2.2 Technological Advancements

Besides expanding basic multi-threading support which appears in both major source-level debuggers [1], 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 [**?**], [**?**]. Our work relies heavily on some experimental work done for gdb [**?**] 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

[1] Gdb and MVSD

appears in *Visual Studio's data visualisers* [**?**] were data in html, xml or image form is according to semantics visualised for the programmer during debugging. Then there is the *high-level debugger* of Mathematica [**?**] which supports arbitrary computation at breakpoints in it's own language, including some visualisation of intermediate results, mainly mathematical formulae.

For the end an independent - but proprietary - project that we would like to mention is the high-level debugger JBixbe [**?**] which has some advanced capabilities in terms of *call-graph visualisation* and also a basic support for visualising data, like the popular front-end *DDD* [**?**] does.

3 Our Approach

3.1 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 concern the equivalency of source code with the running process. In 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 [**?**] 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 a semantically annotated parse tree of the source code. Our choice alters the classical debugging information flow as seen in Fig. 1 (grey area).

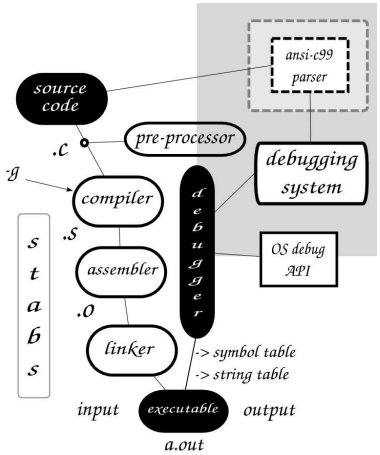

Fig. 1. Expanding the information available to the debugging system.

Now as seen in Fig. 1 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 provided us the means to develop and support features like *syntax-aware navigation* among other things. 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 technologies 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 [**?**]. Their architectural approach also includes a seperate parser to support these facilities 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. In the future we may be able to support these functions directly from the compiler itself, see [**?**] and [**?**].

3.2 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 afformentioned 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. As we can see in *Fig. 2* individual group statements, if, while and other syntax structures are blocked together in Misha to form logical units of execution that can accordingly be traversed.

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 [**?**], 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*

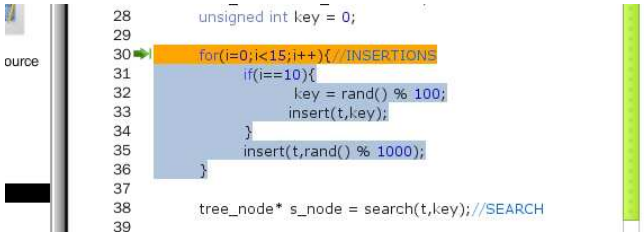

Fig. 2. Syntax Aware Navigation

and updating graph values, infinite graph expansion via menus and depth settings, layout capabilites and other features.

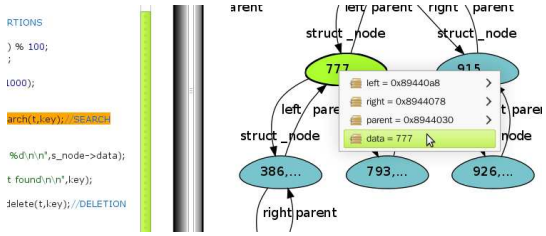

Fig. 3. Data Displaying, Positioning and Editing in graphs

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 related work. Part of what we have achieved here (controlling the symbolic debugger via python) is also a goal for gdb, which aims to use it's extention language as a separate platform for writting usefull tools [**?**].

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 as seen in Fig 4. 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*

Fig. 4. Python to C, seamless inter-language calls

Foundation. In responce to this interest and based upon the still exprerimental work done for *i386* native reverse execution [**?**], we integrated and enhanced the execution record facility of *gdb* with our syntax-aware navigation system 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.

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, for dealing with programming related issues.

4 Conclusion and Future Work

We will 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.

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.[2].

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