GDBTrace: A Tool for Tracing Program Execution at the Statement Level

François Bélanger and Timothy C. Lethbridge
School of Information Technology and Engineering (SITE)
University of Ottawa
Ottawa, Ontario, Canada
{fbelange, tcl}@site.uottawa.ca

Abstract

It is a well-recognized fact that debugging and tracking down the cause of a software failure are particularly difficult tasks in software development. The complexity of these processes increases with the size and age of the system. Often, it is possible to trace the cause of a failure to a certain routine or file; however, complex statements or convoluted logic may make finding the exact cause extremely difficult. This paper introduces GDBTrace, a statement-level tracing tool developed at the University of Ottawa. Although designed with statement level tracing in mind, this tool is also capable of keeping track of routine calls and variable modifications during program execution. GDBTrace has useful options that enable it to trace only selected parts of a system, thereby improving efficiency and reducing the amount of work required to analyse the trace.

Keywords

Program understanding and recovery, software engineering, software maintenance, tracing, software visualisation.

1. Introduction

Maintenance of large systems, and in particular, debugging and making correct modifications, are some of the most complex operations in the software development process.

Currently, it is estimated that over 50% of the effort employed by software engineers is not aimed at developing new systems, but rather at understanding and upgrading older ones [8]. Experience has shown that the maintenance and operation phases consume, on average, 60% percent of the effort of software engineers [5]. In many cases, it is necessary for the software engineer to understand a system developed over twenty years ago. Such systems, often called heritage or legacy systems are often vital to a corporation's survival.

These systems can prove to be very difficult to understand and correctly modify. The system may be poorly documented, the original developers may be unreachable, the system may be written in an obsolete language, etc. In such cases, it is necessary to obtain information about the system using reverse engineering tools.

Generally, reverse engineering is divided into three types of activities: data gathering, knowledge organisation and data exploration [11]. This paper deals with tools of the first category only – specifically, gathering information in the form of traces from running programs.

2. Alternatives Approaches to Gathering Information about Programs

When software engineers set out to make changes to a system, they need to understand the program sufficiently well so that they know what to change and have confidence their changes will be correct.

Programmers can use any of the following techniques to gather information about programs:

- Reading the program sequentially or opportunistically.
- Using static analysis tools to extract information from it such as static call hierarchies etc. [4]

- Using browsing or hypertext tools to follow relationships among static aspects of the code [7,9].
- Running the program under a debugger and following its execution.
- Running the program while gathering a trace using a tracing or profiling tool [12].

The first three techniques gather static information; however, the dynamic information obtained from the latter two techniques is often essential for understanding complex paths of execution. [10] In this paper, we will focus on the dynamic techniques,

2.1 Debuggers

Debuggers have been around since the earliest days of programming; among the most popular debuggers today is the GNU Debugger (GDB). Debuggers allow software engineers to not only execute a program step-by-step, but also to do such things as keeping track of variable modifications and stopping program execution at specified breakpoints.

To accomplish this last operation, GDB replaces program instructions with ones that cause exceptions (such as a divide by zero). When these exceptions are encountered, the GDB takes control and awaits user input. This technique is known as trapping a program [3].

2.2 Tracers and Traces

The tracing of program execution has also been performed since the earliest days of programming. In its simplest form, programmers instrument the code by inserting "print" statements at strategic points; these print statements can be activated when the program is run in a special tracing mode.

More sophisticated tracing tools are available that automatically instrument code. Some of them do this at the source-code level, but others work at the object code level. Tracing can also be implemented in interpreters or VMs, where programming languages use such technology.

Tracing facilities allow programmers to gather information about program execution at different levels of resolution.

- *Inter-process traces* record the messages that are passed between multiple running processes or threads. Instrumentation is placed around the primitives that send and receive such messages
- *Routine-call traces* record each routine call and return.
- *Statement-level traces* follow program execution one line of code (or one instruction) at a time.

Routine call traces are probably the most widely used. However, although they are useful for following the high-level paths of execution, they do not provide sufficient information for many tasks.

No matter what kind of tracing is used, most tracers provide options to output information such as routine arguments, or the values of certain variables.

2.3 Comparing tracing and debugging

When working with large programs, software engineers will often need to use both tracing and debugging because they have complementary advantages and disadvantages:

- Debugging is useful when the engineer has isolated the part of the code he or she needs to investigate or is working with a small program. On the other hand, trying to explore a massive program with a debugger can be like exploring the world on a bicycle.
- Tracing can give information about every line or routine executed, but the amount of information output can be truly massive, especially in the case of statement level tracing. With a debugger, the engineer has fine-grained control over what to look at and only limited information is output.
- When using a debugger, the engineer looks at information as the program executes. This allows the engineer to dynamically change his or her mind about where to place breakpoints, or what data to examine. Examining a trace, on the other hand, can only be done retrospectively.
- The output of a trace can be massaged and visualized so that the software engineer can better understand the dynamics of a program.
- Tracing can be used to run a program through multiple scenarios; the trace can then be examined to see, for example, what portion of lines have been covered.

Both techniques share a disadvantage: They slow the program down. In the case of debugging, this can be extreme slowing if the engineer is stepping a statement at a time; but even tracers can alter the real-time behaviour of a program.

To reduce the real-time impact of tracing, *profiling* is sometimes used instead. A profiler takes a snapshot of the execution at specified intervals. The overall behaviour of the program can then be reconstructed [2].

Table 1 summarizes the key disadvantages and advantages of debuggers and tracers. The final column describes hybrid technology to be discussed next

Issue	Debugger	Tracer	GDBTrace
• Easily used to get a big picture of a program (i.e. generating visualizations of overall patterns of execution)	No	Yes	Yes
• Can be used to rapidly try out multiple scenarios	Maybe	Yes	Yes
• Can be used with a large program, where the engineer has no idea of typical execution paths	No	Yes	Yes
• Easy control over the parts of the system to be examined	Yes	Only with manual instrumentation	Yes
• Produces a manageable amount of output	Yes	No. Especially not for statement level traces.	Possibly
• Permits normal real-time behaviour	No	Usually not	Depends

Table 1: Comparison of typical debuggers, tracers and GDBTrace

2.4 Combining the best features of tracing and debugging tools

The objective of the work discussed in this paper is to develop a solution that combines the advantages of both debugging and tracing, without the disadvantages of either.

The approach taken is to create a statement-level tracer that uses a debugger as its means to generate the trace. We have chosen to implement the tracer on top of GDB. This makes our tool, called GDBTrace, very portable and generic.

The resulting tool *is* a tracer, so it has all three of the advantages of automatic tracing listed in the first three rows of Table 1.

The challenges are therefore as follows:

1. Achieving easy control over the parts of the system to be examined (as in a debugger or with manual tracing instrumentation). This is accomplished by providing the tracer with a variety of powerful commands; these will be discussed in section 3.3.

2. Ensuring the tool produces a manageable amount of output. This is also done by options that give the user control over what is traced, as well as by piping the trace through a facility that "compresses" it.

3. Avoiding system slowdowns that would cause behaviour problems. This is the biggest challenge, and is the main focus of the latter part of the paper.

3. Design of GDBTrace

In this section we discuss the overall design of GDBTrace.

By itself, a debugger is a powerful tool. Its major drawback resides in the fact that a large amount of user input is usually required. GDBTrace requires no input once it has begun tracing. In order to achieve this, GDBTrace was designed as an Expect script driving GDB, as shown in Figure 1.

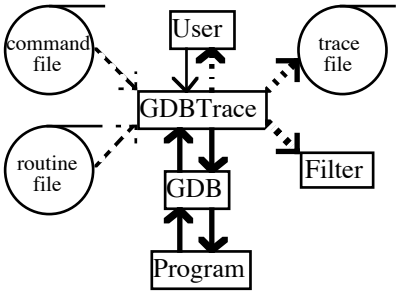

Figure 1: The architecture of GDBTrace. Bold lines represent continuous communication between two components. Thin lines represent a single communication that occurs between components when the program starts. Dashed lines represent optional communication determined by the options the user sets.

Expect is a powerful automation tool designed by Don Libes, at NIST. This tool is capable of recognizing patterns, spawning processes and then controlling them. Thus, scripts written for Expect have already been used to automate many UNIX tools such as *rlogin*, *ftp*, etc [1]. Because Expect is a Tcl extension, scripts written for Expect can also include Tcl program flow instructions [6]. It is by the use of the automation capacities of Expect and the program-flow commands of Tcl that we can obtain traces.

3.1 Debugger Automation

Major advantages of using the GDB as the base for GDBTrace are that it makes GDBTrace language independent and portable. It is capable of working with programs written in any language as long as the GDB "knows" how to deal with them. Likewise it is capable of working on many different platforms.

GDB has the capacity of stepping through a program and outputting the instructions executed. To generate the trace, Expect looks for certain patterns in the GDB output, analyses their contents, and takes actions based on what it sees.

Since the GDB's output always follows a certain format depending on its state, it is easy to recognize specific patterns and isolate the important information contained within it. This matching operation continues until GDB's output indicates program execution has terminated.

Each time a specific pattern is matched, different instructions are executed in order to deal with the specific event. This may include end all tracing, start tracing, trace only the next N lines, etc. This type of flexibility is achieved by the use of Tcl program-flow instructions.

3.2 Tcl Commands in GDBTrace

Tcl statements allowed us to program a great level of flexibility into GDBTrace. The user may select any specific region of the program to trace by setting certain options before execution. This effectively reduces the loss in execution speed by selectively tracing parts of the program where errors are known (or are thought) to occur.

It is also possible for the user to limit the depth of the trace, or to avoid tracing common library routines (such as the C "`printf`" routine). These last options accelerate the speed of execution and help overcome challenge 3 discussed above.

Finally, because of the use of Tcl, it is possible to create command files in order to customize the tracer. Such files may contain specific instructions such as a list of routines to ignore while tracing. Tcl also allows us to write the output of the trace to a pipe.

3.3 Commands Available with GDBTrace

As mentioned earlier, GDBTrace implements many commands to allow for flexible use, faster execution and shorter traces. These commands can be specified as command-line options or in a file.

The commands are summarized in Tables 2 thru 5, and are discussed in the next few sections.

3.3.1 Commands to Start and End Tracing

As mentioned previously, statement-level traces often end up being quite voluminous. Therefore, it is essential to limit the length of such traces. GDBTrace implements the -b, -B, -c and -C commands, which are used to control the start and end of the tracing process.

The -b command is used to indicate to GDBTrace to start tracing indefinitely from this point. It has the lowest priority of the start/end tracing commands, in the sense that the other commands can force tracing to stop. This command is implemented by having the debugger set a breakpoint at the specified location. Once this breakpoint is hit, tracing begins. It can be terminated if execution reaches parts of the code marked with the -c or -C options, discussed below. The -b option allows the software engineer

to avoid tracing until some interesting lines or routines are executed.

The -B option is like -b, except that it indicates to GDBTrace that it should only trace until the function returns. This is useful in cases where only a certain part of the call hierarchy is to be traced. Upon returning from a function marked with a -B option, GDBTrace ceases tracing and resumes full-speed execution of the program.

The -c option is used to instruct GDBTrace to cease tracing and to resume full-speed execution whenever a matching line is encountered. Full-speed execution continues until another line or routine matching a -b or -B command is reached.

The -C option instructs GDBTrace to delete all the -b or -B commands and resume full-speed execution.

The -c and -C options are useful when program efficiency is a concern or upon reaching a point in the program that is known to be functioning correctly.

Command	Effect
-b breakpoint	Start tracing when breakpoint is encountered (while not already tracing)
-B breakpoint	Same as -b, but stop tracing when the routine matching the breakpoint returns.
-c breakpoint	Stop tracing when breakpoint is encountered (while tracing).
-C breakpoint	Same as -c, but prevents the tracer from subsequently stopping in response to any -b or -b command.
-e breakpoint	Used to detect a (or many) occurrence(s) of a particular statement or routine.
-E breakpoint	Same as -e, but does not detect multiple occurrences of a statement or routine.

Table 2: Commands to start and end tracing

Multiple use of the -b, -B, -c, -C, -e and -E commands is permitted, giving the software engineer fine-tuned control over what is to be traced. For example:

- The program can be made to record only the visits to a specific set of lines or routines by specifying -b and -c for each of them.
- It can be used to detect if a particular routine (or line) is ever hit, without noticeably affecting performance, by specifying a -e or -E command for that routine. The output should be just one line, or zero lines if the routine is never hit. This can be done in a debugger too, but when the routine is hit, the engineer would then have to manually remove all breakpoints and resume execution. Doing so might perturb the timing of the system.

3.3.2 Controlling Trace Depth and Length

GDBTrace implements options to limit the depth of the trace. They are -d, -x, -y, -w, -W and -n. All of these options limit the length of the trace in order to achieve faster execution.

Command	Effect
-d depth	Trace *depth* levels of the call hierarchy only.
-x routine	Do not step into anything called by *routine* while tracing, but trace routine itself.
-w file	Do not trace any of the routines listed in *file*
-W routine	Do not trace *routine*, or anything below it.
-n lines	Stop tracing after n lines have been traced

Table 3: Commands to control trace depth and length

The -d command limits the depth of the trace to a specific number of levels of the call hierarchy below the level at which tracing is started (i.e. by -b or -B). This can be used to prevent the tracing of low-level facilities such as utilities. It also permits the software engineer to get a skeletal overview of the higher levels of the hierarchy, drastically shortening the trace. Specifying `-b routine1 -d 1`, for example, can ensure that only the statements in or below `routine1` are traced. The -d option is implemented by passing the `next` command to the debugger instead of the `step` command, once the required tracing depth has been reached.

The -x command is used to instruct GDBTrace to trace the specified routine (if encountered while tracing), but not its subroutines. This command is useful in cases where the software engineer knows that a routine is functional in nature, so

there would be no need to trace the routines it calls.

The -w command is used to prevent GDBTrace from tracing common library routines. The argument to the -w option is a filename containing the names of these common routines. Such routines include C's `printf` command. This command was added to deal with GDB 5.0 and higher. Starting with GDB 5.0, all routines, including standard library routines are stepped into. It is usually assumed that standard library routines return correct results and it is therefore not necessary to trace them. Also, many standard library routines are quite long and, if included, greatly augment the length of the trace.

The -W option is like -w, except that it specifies a particular routine that should not be traced. The difference between -W and -x is that with -x, the routine is traced, but not its subroutines; with -w, even the routine itself is not traced.

The -n option is used to instruct GDBTrace to trace only for a specified number of steps whenever tracing is started. This command is useful to limit the length of the trace in cases where the depth limiting commands are inadequate (e.g. in the case of a loop that executes many times).

Multiple uses of the -d, -x, -w and -W commands are permitted.

3.3.3 The Variable Tracing Command: -u

A very powerful aspect of GDBTrace is its ability to trace up to the point where a variable changes in value.

The -u command instructs GDBTrace to resume full-speed execution when hitting a line that sets the specified variable to any, or a specific, value.

Command	Effect
-u var	Stop tracing and resume full-speed execution whenever *var* changes in value.
-u var=value	Stop tracing and resume full-speed execution whenever *var* is set to *value*.

Table 4: Two forms of the -u command

There are two ways of using this command, illustrated in Table 4. The first involves only specifying the variable name; in this case if an instruction changes the value of that variable, full-speed execution resumes. The other way of using this command is specifying a variable name and a value. GDBTrace will resume full-speed execution of the program when an instruction sets the specified variable to the specified value.

This command is useful because one of the questions frequently asked by software engineers is, "How did that variable get set to that value?"

Multiple use of the -u command is permitted. The major drawback of this feature is the slowdown associated with its use.

3.3.4 Commands that Control Input and Output

GDBTrace incorporates many commands that allow the user to customize input and output to his or her current needs. They are -r, -v, -o, -i, -s, -t and -l.

Command	Effect
-r	Output only lines that represent routine calls (or variable value changes if v is used as well)
-v *var*	Whenever *var* changes in value, output the change.
-o *file*	Direct output to *file*
-i *file*	Obtain extra commands from *file*
-s	Output a stack trace whenever tracing is resumed (by -b, -B, -e or -E) so the context is clear,
-t	Output a timestamp on each line, at the highest resolution available
-l	Indent the trace according to the depth in the routine call hierarchy to make the trace easier to read.

Table 5: Commands to control trace depth and length

The -r command instructs GDBTrace to only output lines that contain routine call traces or variable change traces. This permits the user to obtain a higher-level view of the system – a routine-call trace or the trace of the use of certain variables (as specified using the -v command below). Only a single use of the -r command is permitted. Note the debugger still has to step through every line when the -r command is used; the command merely hides un-needed output.

The -v command instructs GDBTrace to display value changes of a certain variable, specified by the argument to the command. This command's major drawback is the slowdown associated with its use. Multiple usage of the -v command is permitted.

The -o command instructs GDBTrace to send its output to a file rather than to `stdout`. Under DOS or Unix, the effect of the -o command can also be obtained by using the ">" redirection operator.

The -i option instructs GDBTrace to obtain additional commands from a file. This is useful to avoid having to type all commands on the command line. Multiple use of the -i command is permitted.

The -s option instructs GDBTrace to obtain a stack trace from the debugger after encountering a start tracing instruction. This option is useful in order to allow the user to understand the path the program followed before arriving at this point if it was not being traced. Only a single use of the -s command is permitted.

The -t command instructs GDBTrace to output a timestamp on each line. This allows the user to determine the execution speed of the program while being traced. Only a single use of this command is permitted.

The -l command instructs GDBTrace to indent the trace depending on the level at which it is located. This allows for easier visualization of the trace. Only a single use of this command is permitted.

4. Making GDBTrace Useful by Allowing it to Gather Data Quickly

As discussed earlier, because GDBTrace makes use of GDB there will be a performance penalty in using it to gather a trace. In order to make GDBTrace useful, we must therefore quantify this penalty, and determine modes of use in which the penalty has little impact.

GDBTrace works sufficiently fast for programs that only execute a few thousands of lines, so in those cases the performance penalty will not be a problem. We therefore performed our experiments on tasks that consume considerable CPU time. We chose an open-source program: Xpaint, as well as a simple program we wrote, to exercise the features of GDBTrace. This test program is composed of three files. These files mostly consist of loops executing mathematical functions and `printf` statements in order to allow us to control the length of execution. This program also contains a recursive call that is fifteen levels deep. Table 6 summarizes the tasks we utilised to test the functionality of GDBTrace.

Task	**Statements executed in complete trace**
Xpaint – Normalize Contrast	5555745
Xpaint – Spray	3354
Xpaint – Greyscale	2458587
Xpaint – Solarize	1228821
Test program	600 to 10M

Table 6: Program runs used to experiment with GDBTrace

4.1 Measuring slowdown

Slowdown can be measured both in terms of CPU time used and elapsed time.

Overall CPU slowdown for full tracing ranges from about 60 to over 175, whereas elapsed time slowdown ranges from about 600 to over 1100. The differences between these the two types of slowdown, and the presence of ranges can be accounted for by the fact that a lot of time is consumed in operating system facilities that are not actually traced.

The most useful figure is the elapsed time slowdown; this is the figure that has practical relevance to the software engineer, and it is what we will try to reduce.

Table 7 shows the slowdown when particular features of GDBTrace are switched on, while still obtaining a full trace.

Feature	Average slowdown (From basic GDBTrace)
-u *var=value*	1.281-Varies depending on number of lines traced and number of variables to check.
-v *var* (trace a variable)	1.281-Varies depending on the number of variables to be traced.
-t (output timestamp)	1.009
-l (indent output)	1.235

Table 7: Slowdowns associated with various GDBTrace commands that make performance worse.

4.2 Approaches to reducing the slowdown

A software engineer will often want to use GDBTrace on a large program, either when the slowdown of a full trace becomes prohibitive or when timing is being interfered with so behaviour changes. The following are some of the strategies he or she can use:

The first set of strategies can be used when the engineer knows nothing about the program

Strategy 1. Generate a top-level skeletal trace using the -d command. Clearly the effectiveness of this will depend on the depth of the dynamic call hierarchy of an individual program. If a program does everything in its main program, there can be no benefit at all. Table 8 shows how slowdown changes as the argument to -d is increased for each of the Xpaint and test program full-trace runs.

Task	-d 2	-d 4	-d 6	-d 8
Xpaint - Normalize Contrast	843	843	843	843
Xpaint- Spray	1296	2475	19619	51071
Xpaint- Greyscale	1158	1158	1158	1158
Xpaint- Solarize	667	667	667	667
Test program (1134 lines)	91	137	153	157
Test program (41 453 lines)	112	149	156	157

Table 8: Slowdown when tracing to different levels of depth.

As the results in Table 8 suggest, the more we augment the argument to the -d option, the longer the trace takes. This is normal as the deeper into the system we trace, the longer the execution time will be. Therefore, in the case of CPU intensive operations, it is better to begin tracing the higher-level routines first and then gradually drop levels, while at the same time using commands to limit the depth of the trace, than obtaining a full trace immediately.

Using the results from Table 8, we have been able to determine that Xpaint's Greyscale operation is confined to two levels of depth. This conclusion is confirmed by an examination of Xpaint's source code. The same conclusion can be applied to the Solarize and Normalize Contrast functions.

The Spray function results led us to believe that this function dives deeply into subroutines. An analysis of the Xpaint source code confirmed our hypothesis.

The test program results are exactly what we expected. However, this program was specifically written to test GDBTrace's functionality and therefore contains up to 15 levels of depth. However, all levels do not contain the same number of statements to execute, which explains the difference between the version with 1134 lines and the one with 41 453 lines.

Strategy 2: Generate a true routine-call trace, setting breakpoints for each routine. This can be accomplished by first gathering a trace of the routines called (using -r). This can be put in a file, and then run through a simple Unix script call 'gtlistroutines' that we would provide to extract

the unique routines and turn them into -b commands. The file can then be imported using the -i option; additionally -n 1 can be specified to ensure that tracing occurs only whenever a routine is entered.

Of course, in order to generate the file of routines, the software engineer must first run the trace slowly using -r; however once this is done, he or she can then speed up subsequent experiments.

This strategy would only provide a high-level view of the system. In the case where specific details are needed, it would be better to use other commands provided by GDBTrace.

Strategy 3: Same as strategy 2, except using -n 100 and -n 1000 instead of -n 1. This strategy would generate a larger amount of information than the previous one, but would also provide more details as to the working of the system. The volume of information should not be overwhelming.

The next set of strategies can be used when the software engineer begins to understand the behaviour of the program, and can therefore be selective about what he or she will execute.

The key to these strategies is to pick a set of routines to include, exclude or exclusively trace. The software engineer can explore the trace to determine this set, but we will also provide a set of options on gtlistroutines that will help him or her do so.

Running 'gtlistroutines -c' will count the total number of statements executed while in any routine in the system (based on a full trace, or a trace that was truncated using -n).

The output of 'gtlistroutines -c' has three columns: The routine name; the number of statements executed while in that routine, and the number of statements executed while in that routine or (recursively) any routines it calls.

Running 'gtlistroutines -f' will take a trace and give a list of routines sorted in descending order by fan-in. This list can be used to eliminate utilities.

The following strategies are other strategies that could be used by software engineers in order to avoid severe performance penalties while running GDBTrace.

Strategy 4: Only tracing the five most CPU-intensive routines using -B -d1.

Strategy 5: Only tracing the routine that takes the median number of statements, using -d3. This is designed to illustrate a typical use of the system.

Strategy 6: Excluding from the trace the top five routines using -w.

Strategy 7: Excluding from the trace the set of utilities using -w.

5. Future work

The following are some of the ideas we have for improving GDBTrace:

- Creating a GUI that will allow the software engineer to rapidly repeat traces, specifying different options.
- Automatically display the results of gtlistroutines so the user can select some routines to include or exclude.
- Allowing the user to nest -i commands.

6. Conclusions

In this paper, we have discussed the design of GDBTrace, a trace generation tool designed as an Expect script running on top of the GDB debugger.

The main objective was to obtain a trace generator that has the level of control found in debuggers, with the power of a tracer to let the program run at full speed.

The actual instrumentation provided by GDB and the expect script take a very large toll on performance of the program being traced; the software engineer therefore has to judiciously use options that limit the amount of information being gathered. We presented experiments showing the kinds of slowdowns the user of GDBTrace can expect when various options are employed.

It can be seen from the experiments that there are ways to use GDBTrace without suffering a significant performance penalty.

Source code for GDBTrace can be obtained at http://www.site.uottawa.ca/~tcl/kbre/GDBTrace.

Acknowledgements

This work is supported by Mitel Corporation and NSERC and sponsored by the Consortium for Software Engineering Research (CSER).

The authors would like to thank (in no particular order) the following people who gave

us references or feedback for this paper: Dr. Robert Laganière, Abdelwahab Hamou-Lhadj, Abdelouahab Zaghrouti, Huixiang Liu and Sonia Vohra.

Expect is a public-domain tool written by Don Libes of NIST.

About the Authors

François Bélanger is a second-year undergraduate student in Computer Engineering at the University of Ottawa's School of Information Technology and Engineering (SITE). He can be contacted via email at fbelange@site.uottawa.ca.

Timothy C. Lethbridge is an associate professor at SITE. His research interests include reverse engineering and software visualization. He can be contacted via email at tcl@site.uottawa.ca

References

[1] See http://expect.nist.gov for a library of these scripts.

[2] Ball, T. and Larus, J. "Optimally Profiling and Tracing Programs", Proceedings of the 19th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1992, pp. 59-70.

[3] Gilmore, J. and Shebs S. "GDB Internals", http://sources.redhat.com/gdb/onlinedocs/gdbint_3.html#SEC8

[4] Gouranton, V. and Le Metayer D. "Formal Development of Program Analysers", Proceeding of the 8th Israeli Conference on Computer-Based Systems and Software Engineering, 1997, pp. 101-110

[5] Lethbridge, T. C. and Laganière R. "Object-Oriented Software Engineering: Practical Software Development Using UML and Java" McGraw-Hill, 2001

[6] Libes, D. "Exploring Expect", O'Reilly, 1996

[7] Sim, S.E.; Clarke, C.L.A; Holt R.C. and Cox, A.M. "Browsing and Searching Software Architectures", Proceedings of the IEEE International Conference on Software Maintenance, 1998, pp. 381-390.

[8] Standish, T.A. "An Essay on Software Reuse", IEEE Transactions on Software Engineering vol. SE-10, no. 5, Sept 1984, pp.494-497.

[9] Storey, M.-A.; Wong, K. and Müller, H.A. "How Do Program Understanding Tools Affect How Programmers Understand Programs?", Proceedings of the 4th Working Conference on Reverse Engineering, 1997, pp. 12-23.

[10] Systä, T. "Understanding the Behavior of Java Programs", Proceedings of the 7th Working Conference on Reverse Engineering, 2000, pp. 214-223.

[11] Tilley, S.R.; Smith, D.B. "Coming Attractions in Program Understanding", Carnegie Mellon University, 1996

[12] Walker, R.J.; Murphy, G.C.; Steinbok, J. and Robillard, M.P. "Efficient Mapping of Software System Traces to Architectural Views", Proceedings of CASCON 2000, pp. 31-40.