CSIclipse: Presenting Crash Analysis Data to Developers

Peter Ohmann
University of Wisconsin–Madison
ohmann@cs.wisc.edu

Ben Liblit
University of Wisconsin–Madison
liblit@cs.wisc.edu

Abstract

Debugging is difficult and costly, especially for failures that occur after deployment. In prior work, we developed a suite of instrumentation and analysis tools, collectively titled the Crash Scene Investigation toolkit (CSI). These tools aid developers by providing additional information about failing program executions using latent data in post-failure memory dumps. While we showed that our technique is effective in reducing execution ambiguity, it lacked a proper user interface for developers.

In this paper, we present CSIclipse, a work-in-progress plugin for the Eclipse integrated development environment (IDE) that brings our analyses directly to the user. The goal of our plugin is to ease the burden of debugging production failures by conveniently presenting CSI trace and analysis data with intuitive source code overlays and powerful data exploration mechanisms. While designed for our CSI data, our plugin is likely general enough to support trace data from a variety of program analyses.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—User interfaces; D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids; D.2.6 [*Software Engineering*]: Programming Environments—Integrated environments

General Terms Algorithms, Languages, Human Factors

Keywords Debugging, postmortem program analysis, software development, integrated development environments

1. Introduction

Studies indicate that debugging, testing, and verification account for at least 50–75% of a software project's cost [6, 19]. Nevertheless, production-run failures are inevitable in complex software. Detailed crash reports can theoretically help developers, but present some important issues. First, gathering extremely detailed run information (at the level of an execution trace) is infeasible in a deployed scenario due to both performance and privacy concerns. Thus, we are forced to be selective about what crash information to record and report. Second, crash reports must be presented in such a way as to be comprehensible to developers.

In our prior work [15, 16], we enhanced failure data in core memory dumps with lightweight, customizable instrumentation, suitable for deployed applications. This work also demonstrated the effectiveness of the tracing with two post-mortem analyses focused on reducing the failure-relevant code a developer would need to consider. Collectively, our instrumentation and analysis framework comprises the Crash Scene Investigation toolkit (CSI)[1]. Relevant details of the CSI system are discussed in section 2.

Our prior work quantitatively demonstrated CSI's ability to substantially prune the suspect code a programmer must consider (relative to unenhanced core dumps). However, CSI lacked any visual interface to present results to developers. In addition, prior work by Parnin and Orso [17] indicates that mere counts of suspect lines are not always indicative of a tool's actual impact on debugging performance.

This paper presents CSIclipse[2], a plugin for the Eclipse IDE that is currently under active development. CSIclipse presents CSI analysis results to developers in a convenient but thorough manner. Specifically, CSIclipse provides support for (1) viewing trace data and analysis results, (2) annotating source code with execution coverage information based on analysis results, and (3) stepping through execution trace data. Our ultimate goal (see future work in section 5) is to extensively evaluate our techniques with a user study; IDE integration is a first step toward that goal.

The remainder of the paper is organized as follows. Section 2 provides necessary background information on our CSI framework. Section 3 discusses our design and describes each feature of CSIclipse in more detail. We consider related

[1] Source code for CSI is available at `http://pages.cs.wisc.edu/~liblit/ase-2013/code/`.

[2] Source code for CSIclipse is available at `http://pages.cs.wisc.edu/~liblit/etx-2015/code/`.

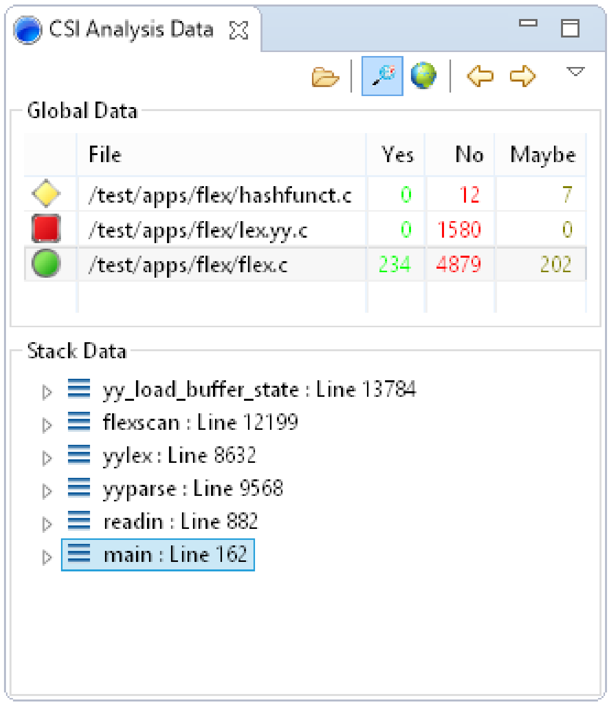
work in section 4. Section 5 details future work plans, and section 6 concludes.

2. Background

In this section, we describe background information on CSI, a lightweight instrumentation and analysis toolkit for C/C++ programs that we developed in prior work [15, 16]. In that work, we augment core dumps with lightweight, tunable run-time instrumentation. We then propose two postmortem analyses to take advantage of the enhanced data, in addition to information readily-available in core dumps. Here, we focus on the details relevant to analysis presentation. See our prior papers [15, 16] for full details.

CSI currently uses two instrumentation mechanisms to enhance readily-available core dump data. Both mechanisms store their trace data entirely in memory, and each can be enabled or disabled at function granularity by post-deployment binary configuration. The first mechanism, *path tracing*, is an extension of an efficient path profiling approach developed by Ball and Larus [3]. Path tracing records a partial execution suffix for each traced function. These traces reside in stack-local storage, and are discarded whenever a function returns; thus, at the time of a crash, we are able to extract path trace information only for traced functions in the active program stack. Each path trace may be *partial*, meaning that it represents a suffix of the complete execution path taken through the function; in other words, each path trace ends at the crash point (or final call site) in the frame, but may not begin at function entry. Our second mechanism, *program coverage*, complements these dense local path traces with coarse-grained global information. Coverage is more straightforward: we maintain one bit of global information for each *trace point* indicating whether or not that point ever executed during the current program run. The choice of trace points is customizable, and CSI provides support for a number of alternatives (e.g., functions, call sites, all statements, etc.). When used together, our path traces and program coverage bits provide dense information close to the failure point, and data that scales gracefully as failure exploration leaves the active stack.

We previously proposed two analyses to take advantage of our enhanced core dump data [15, 16], but here we focus on just the analysis relevant for CSIclipse: execution path restriction. The goal of our analysis is to extract execution information for a failure report from a single failing run of the program. Such a failing run leaves a core dump which contains some corresponding collection of trace and coverage data. However, this crash data is incomplete and therefore ambiguous: multiple failing runs could yield the same data. Therefore, the data in a core dump matches some subset of the possible ways that a program may have run before failing. Execution path restriction analysis partitions the nodes of a program's control-flow graph (CFG) into three classes: those that execute on *all* runs consistent with the analyzed core dump, those that execute on *no* such runs, and those that execute on *some subset* of consistent runs. Informally, we will often refer to these sets as "yes," "no," and "maybe" sets. Beginning with the conservative approximation that all nodes may have executed on the failing run, the goal of our analysis, then, is to reclassify nodes into the yes and no sets (making the maybe set as small as possible). Note that a complete program trace would always result in an empty maybe set, as we would directly observe all CFG nodes that did and did not execute. CSI, however, intentionally sacrifices complete information to reduce tracing overhead; thus, the optimal maybe set is often non-empty.


CSI Analysis Data

Global Data

	File	Yes	No	Maybe
◆	/test/apps/flex/hashfunct.c	0	12	7
🟥	/test/apps/flex/lex.yy.c	0	1580	0
●	/test/apps/flex/flex.c	234	4879	202

Stack Data

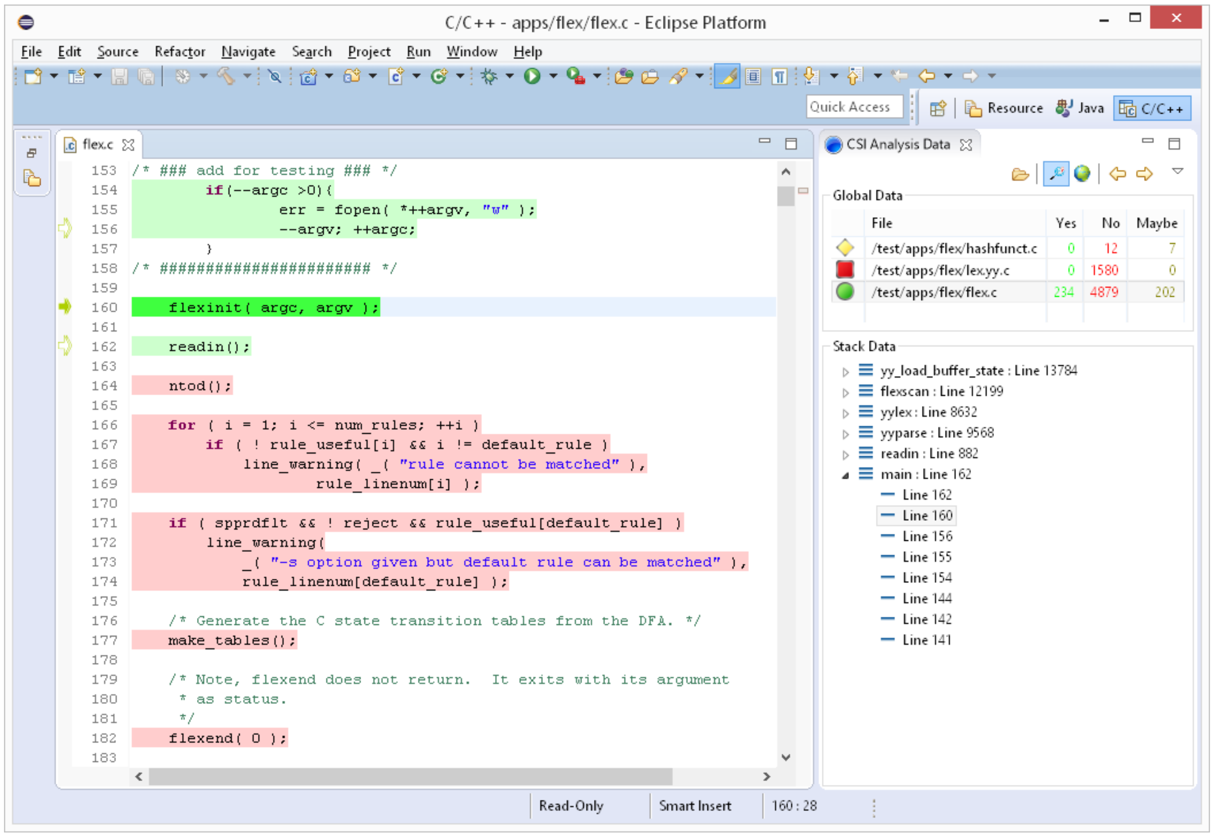
- yy_load_buffer_state : Line 13784
- flexscan : Line 12199
- yylex : Line 8632
- yyparse : Line 9568
- readin : Line 882
- main : Line 162

Figure 1. The CSIclipse data view. Screen shots in figures 1 to 3 show a debugging session for a crash in flex.

We perform this analysis both *intraprocedurally* (for each frame in the crashing program stack) and *interprocedurally* (across the entire program's execution). The result of running our analysis on a core dump is: one global record indicating the yes, no, and maybe sets for the entire program's execution, and one local record for each crashing stack frame indicating the yes, no, and maybe sets for the current call to that function. To better support consumers of analysis results (particularly CSIclipse), we output CFG nodes as line numbers, and split out data by source file; thus, each set contains a (possibly-overlapping) set of line numbers [15, 16].

3. Design

In this section, we describe the design of CSIclipse, a plugin for the Eclipse IDE. We have tested CSIclipse on Eclipse versions 4.3–4.5, on a variety of Linux, OS X, and Windows


C/C++ - apps/flex/flex.c - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Resource
Java
C/C++
flex.c
CSI Analysis Data
Read-Only
Smart Insert
160 : 28

```
153 /* ### add for testing ### */
154         if(--argc >0){
155                 err = fopen( *++argv, "w" );
156                 --argv; ++argc;
157         }
158 /* ####################### */
159
160     flexinit( argc, argv );
161
162     readin();
163
164     ntod();
165
166     for ( i = 1; i <= num_rules; ++i )
167         if ( ! rule_useful[i] && i != default_rule )
168             line_warning( _( "rule cannot be matched" ),
169                     rule_linenum[i] );
170
171     if ( spprdflt && ! reject && rule_useful[default_rule] )
172         line_warning(
173             _( "-s option given but default rule can be matched" ),
174             rule_linenum[default_rule] );
175
176     /* Generate the C state transition tables from the DFA. */
177     make_tables();
178
179     /* Note, flexend does not return.  It exits with its argument
180      * as status.
181      */
182     flexend( 0 );
183
```

Global Data

	File	Yes	No	Maybe
◆	/test/apps/flex/hashfunct.c	0	12	7
🟥	/test/apps/flex/lex.yy.c	0	1580	0
●	/test/apps/flex/flex.c	234	4879	202

Stack Data

```
▷ yy_load_buffer_state : Line 13784
▷ flexscan : Line 12199
▷ yylex : Line 8632
▷ yyparse : Line 9568
▷ readin : Line 882
◢ main : Line 162
      Line 162
      Line 160
      Line 156
      Line 155
      Line 154
      Line 144
      Line 142
      Line 141
```

Figure 2. Highlighting and stepping through frame-local analysis data

platforms. Screenshots from this section are taken with Eclipse version 4.5.0. Throughout this section, we will use an example crashing run from the program flex, which we obtained from the Software-artifact Infrastructure Repository [4, 18]. We analyzed the failure data from this run with our CSI toolkit, producing result data as detailed in section 2.

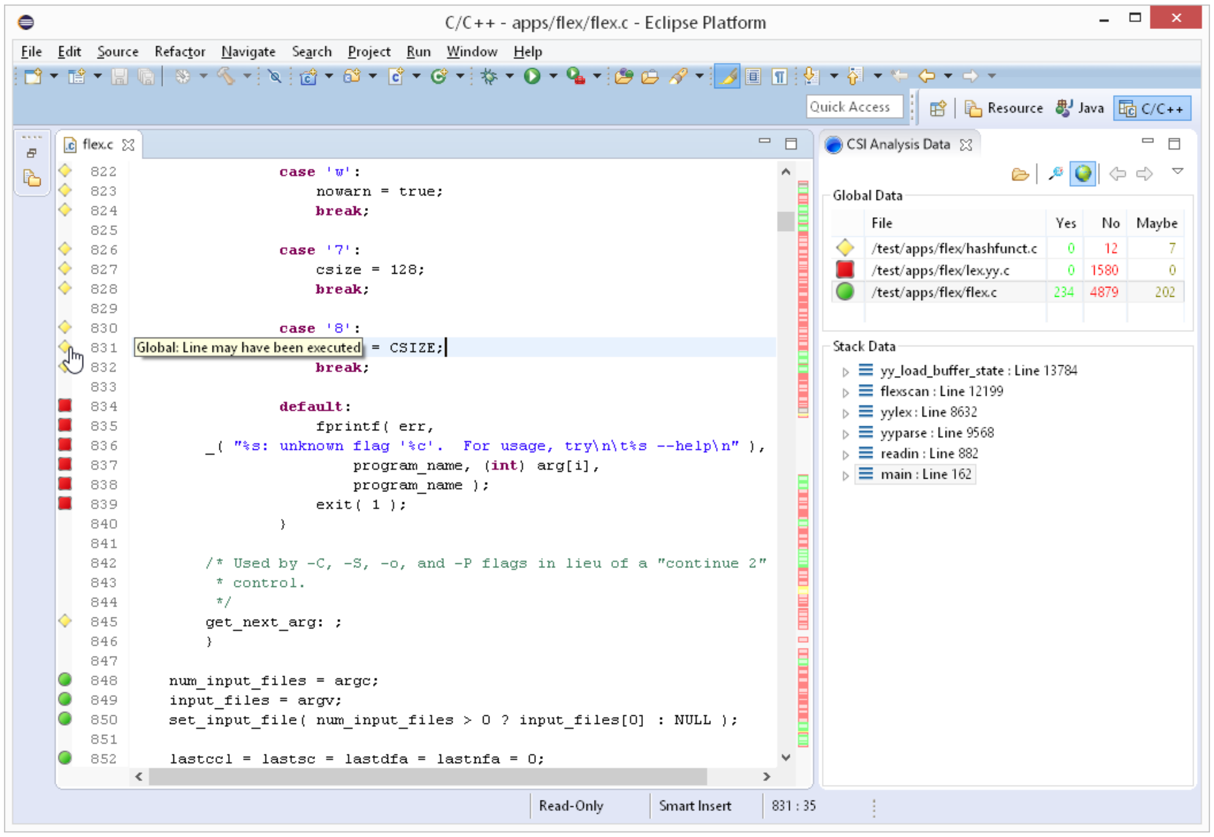
The high-level goal of CSIclipse is to encourage developers to use CSI by reducing the effort required to understand and utilize CSI trace and analysis data. CSI currently outputs sequences of line numbers for execution path restriction results and each stack frame's path trace. Developers must map line numbers to source lines manually: a slow, tedious process. CSIclipse automates this. First, CSIclipse annotates source lines based on their "yes," "no," or "maybe" classification, allowing developers to quickly scan source code to determine which lines did and did not execute. Second, CSIclipse allows developers to directly step through path trace data in the Eclipse source editor.

The heart of CSIclipse is the CSI Analysis Data view, shown in figure 1. This view allows the developer to load crash data, details all analysis data available for the provided crash, and provides navigation for viewing different aspects of CSI trace and analysis results. The toolbar at the top of the view has buttons for loading crash data, toggling local and global source code annotations, and stepping forward and backward through path trace data (respectively). We describe each of these functions in more detail throughout the remainder of this section.

CSIclipse loads analysis result data in a simple comma-separated value (CSV) format, with one record for each file in the program's source, and one record for each frame in the stack trace for the crash. Each record consists of:

- a source file, f
- a "yes" set containing lines of f (optional)
- a "no" set containing lines of f (optional)
- a "maybe" set containing lines of f (optional)
- a function name (frame only)
- a path trace containing lines of f (optional; frame only)

Note that CSIclipse only maintains data for a single crash report, corresponding to a single failing run of the program. Upon loading a new crash report, the old data is overwritten. After loading a crash report, the CSIclipse view displays global and stack-local analysis results as shown in figure 1. The Global Data section displays one entry for each source file in the project, indicating the number of lines in that file that definitely executed, did not execute, or may have executed for the provided crash data. In this example, we loaded our analysis result data from the crashing run of


C/C++ - apps/flex/flex.c - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Resource
Java
C/C++
flex.c
CSI Analysis Data
Read-Only
Smart Insert
831 : 35

```
                case 'w':
                    nowarn = true;
                    break;

                case '7':
                    csize = 128;
                    break;

                case '8':
Global: Line may have been executed = CSIZE;
                    break;

                default:
                    fprintf( err,
        _( "%s: unknown flag '%c'.  For usage, try\n\t%s --help\n" ),
                        program_name, (int) arg[i],
                        program_name );
                    exit( 1 );
                }

        /* Used by -C, -S, -o, and -P flags in lieu of a "continue 2"
         * control.
         */
        get_next_arg: ;
        }

    num_input_files = argc;
    input_files = argv;
    set_input_file( num_input_files > 0 ? input_files[0] : NULL );

    lastccl = lastsc = lastdfa = lastnfa = 0;
```

Global Data

	File	Yes	No	Maybe
◇	/test/apps/flex/hashfunct.c	0	12	7
🟥	/test/apps/flex/lex.yy.c	0	1580	0
🟢	/test/apps/flex/flex.c	234	4879	202

Stack Data

```
▷ yy_load_buffer_state : Line 13784
▷ flexscan : Line 12199
▷ yylex : Line 8632
▷ yyparse : Line 9568
▷ readin : Line 882
▷ main : Line 162
```

Figure 3. Annotations for global execution data

flex, which has three files in the project: lex.yy.c, which was completely unexecuted (and is therefore marked with a red square 🟥); flex.c, which had at least one line execute on the crashing run (and is marked with a green circle 🟢); and hashfunct.c, for which our analysis was unable to determine if any lines executed on the crashing run (and is marked with a yellow diamond ◇). Clicking on an entry brings the user to a scratch read-only version of the file for debugging. This behavior is configurable; we choose to create a read-only copy of the file by default to maintain correspondences among instrumentation metadata, analysis results, and source code line numbers. CSIclipse creates this copy the first time a user selects a particular file.

The Stack Data section displays the crashing program stack. For our example, the program crashed at line 13784 of flex.c, in function yy_load_buffer_state. Clicking on an entry here again opens a buffer (read-only by default), centered on the function corresponding to the selected frame. In addition, selecting a frame enables additional frame-local debugging functionality if the local annotations button (in the toolbar) is toggled to the "on" position. Figure 2 shows these features. First, CSIclipse highlights each source line in the relevant function to indicate whether that line executed during the execution of the selected stack frame: green for "yes," red for "no," and yellow for "maybe." Second, selecting a frame expands the frame's path trace record (if present). The current path entry is brightly highlighted (and indicated with an arrow ➧, as shown in figure 2), and the user can step through the trace using the forward and backward buttons in the toolbar. In a realistic debugging scenario, a developer is likely to start from the failure point, and step backward through the provided statement trace. CSIclipse also allows a developer to connect traces across stack frames (where path trace data is available) and examine calling context for the failure. In this example, a user is currently stepping through the path trace gathered for main's stack frame. Note that each other stack frame also displays the line number of the final call (or crash location) in that function, and has additional path trace information (as indicated by the twistie drop-down icon).

While looking through main's execution, the developer will notice that the function flexinit was called on line 160, but is no longer on the active stack; thus, our failure data will contain no path trace data for that call. However, flexinit is a complex initialization function, and, in this case, contains relevant context information for the failure. When the user activates the global annotations button (in the toolbar), CSIclipse adds a marker for each source line in all project files. Each marker indicates the line's execution status (yes, no, or maybe) for the entire program's execution. Figure 3 shows these markers for our example crash report, centered on a portion of flexinit. This global information is very nearly the standard notion of *line coverage* in program testing, with the small addition of the "maybe" possibility (for lines which may or may not have been covered in the execution). These markers are less invasive than their local counter-parts (as small icons in the side-bar of the source file). Being standard Eclipse markers, though, global annotations are also visible in the file overview to the right of the scroll bar. This has a number of important advantages: it allows a developer to quickly assess how much of a given file's code was used (visually complementing the information in the Global Data section), helps to identify large regions of executed or unexecuted code in a source file, and facilitates navigation to interesting code regions. These global markers also intentionally do not overlap with local annotations. In fact, global and local source annotations are entirely independent, and can provide useful information when used together. For example, a line annotated locally "no" and globally "yes" indicates that the containing function executed previously. A function whose final statement is marked as globally "no" never executed outside the crashing stack context.

4. Related Work

Others [8, 12, 13] have developed Eclipse plugins to evaluate and display program coverage information. Often, such tools operate in the context of running or automatically generating JUnit test cases for Java programs. We draw inspiration from these tools on how to present coverage information to developers in the Eclipse IDE. However, our analyses (1) work in the context of a single failure, (2) must also handle partial coverage data (i.e., cases where multiple executions could possibly lead to a provided core dump), (3) must recover this data from external failing executions (rather than Eclipse-local test executions), (4) segregate stack-local coverage data and global coverage data, and (5) also include path trace information. These concerns lead to different design choices. First, we use an explicit annotation for unknown execution data, as ambiguous traces are the normal case for our analysis. Second, we load execution data from external sources in a simple CSV format; this also improves the integration potential of our tool. Finally, we separate global and local data both by section and by different annotation types; this allows us to take advantage of the full range of CSI result data, and adapt to different debugging contexts.

Many prior tools present program traces to developers [1, 10, 14]. The Path Projection toolkit [10] visualizes program paths from static analysis reports in an IDE, and provides users with a variety of tools with which to explore, understand, and compare different error paths. Our traces differ in some important respects: they are incomplete, and are derived from incomplete data from deployed applications. Nevertheless, these are mature tools with support for visualizing "nested" stack frame traces. We could potentially improve trace visualization in CSIclipse by adopting similar approaches, or perhaps even integrate with existing tools.

Whyline [11] is a debugging tool that allows programmers to ask "why" and "why not" questions about a program's execution in the context of an interactive debugging session. The tool uses a combination of static and dynamic analysis to suggest and answer relevant questions. CSIclipse similarly aids developers in understanding failing program executions, but operates in a different context. For CSI, we assume that failure data is recovered from deployed software, and, thus, we intentionally reduce trace detail to improve instrumentation efficiency. Program slicing (in the context of debugging) extracts portions of a program's code that are relevant to a point of interest. Whyline and many other tools [2, 7, 9] allow a developer to perform static (generalizing all possible executions) or dynamic (restricted to a single failing execution) slicing within an IDE. CSI does support a separate analysis for partially-dynamic program slicing, but our present iteration of CSIclipse does not integrate this feature.

5. Limitations and Future Work

CSIclipse currently offers many features for viewing and navigating crash data, and we plan to build on our successes in a number of directions. In the near future, we plan to more-fully integrate CSIclipse with the existing Eclipse C/C++ Development Tooling (CDT). First, while the CSIclipse Analysis Data view complements standard Eclipse debugging features, the two are currently not closely integrated. Specifically, the CSIclipse Stack Data section presents stack trace information which largely overlaps with the Eclipse Debug view; our data should be accessible from this standard access point. Second, our plugin currently reads CSI analysis result data from an external file. When a developer loads a core dump into Eclipse for postmortem debugging, we could automatically run CSI analysis on that core dump in the background, and later present the results to the developer. Unfortunately, our analysis can run for a substantial amount of time, and relies on external static metadata (created during compilation). We believe that we can overcome these challenges, but the implementation remains future work.

The CSI toolkit itself has some limitations, most notably the fact that CSI currently only supports C/C++ programs. We are expanding our instrumentation and analysis frameworks to support other languages, including Java. CSIclipse, however, does not depend on a specific programming language; rather, it reads result data from an external analysis tool. Thus, new language developments will expand the applicability of CSIclipse, but are orthogonal to its development.

As noted in section 4, many analysis toolkits (including our own CSI toolkit) provide program slicing functionality to determine a smaller set of statements that are transitively responsible for crashing program state. Our analysis currently supports a static/dynamic hybrid variant of both intraprocedural and interprocedural backward slicing, and we plan to bring this functionality to CSIclipse in the future.

While we believe that our tool shows promise, we have yet to empirically evaluate its usefulness for real debugging tasks. We have designed a user study for this purpose, with two important goals: (1) to understand and model the debugging process using standard tools (such as Eclipse and gdb [5]), and (2) to determine the effect our tooling has on that process, and its effectiveness in reducing debugging effort and improving the quality of bug fixes. The development of CSIclipse is a key first step in this endeavor.

6. Conclusions

In this paper, we present CSIclipse, an Eclipse plugin for displaying and navigating analysis results from the CSI toolkit. CSIclipse is under active development, but already supports both local and global source code annotations, as well as bidirectional "stepping" functionality for execution traces. We have also identified several promising future directions for extension, as detailed in section 5. While CSIclipse was specifically designed to work with CSI, it could theoretically support many other analysis tools via basic CSV input (see section 3). Any tool that produces complete or partial coverage information for a program's execution may plug in to our framework. CSIclipse is designed to make the postmortem debugging experience easier, and we hope that other researchers can also benefit from our efforts.

7. Acknowledgments

This research was supported in part by NSF grants CCF-0904371, CCF-0953478, CCF-1217582, and CCF-1420866. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF or other institutions.

8. References

[1] B. Alsallakh, P. Bodesinsky, A. Gruber, and S. Miksch. Visual tracing for the Eclipse Java debugger. In T. Mens, A. Cleve, and R. Ferenc, editors, *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, pages 545–548. IEEE Computer Society, 2012.

[2] P. Anderson, T. Reps, and T. Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE Trans. Softw. Eng.*, 29(8):721–733, Aug. 2003.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, 1996.

[4] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[5] GDB developers. GDB: The GNU Project Debugger, July 2014.

[6] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, 41(1):4–12, Jan. 2002.

[7] C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling Java programs for parallelism. In *Proc. 2nd International Workshop on Multi-Core Software Engineering (IWMSE)*, pages 49–55, May 2009.

[8] J. Hofer. eCobertura. `http://ecobertura.johoop.de/`, Nov. 2010.

[9] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering the Indus Java program slicer to Eclipse. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 269–272. Springer, 2005.

[10] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal. Path projection for user-centered static analysis tools. In S. Krishnamurthi and M. Young, editors, *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'08, Atlanta, Georgia, November 9-10, 2008*, pages 57–63. ACM, 2008.

[11] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 301–310. ACM, 2008.

[12] G. C. Morrison, C. P. Inggs, and W. C. Visser. Automated coverage calculation and test case generation. In J. H. Kroeze and R. de Villiers, editors, *2012 South African Institute of Computer Scientists and Information Technologists Conference, SAICSIT '12, Pretoria, South Africa, October 1-3, 2012*, pages 84–93. ACM, 2012.

[13] Mountainminds GmbH & Co. KG and Contributors. EclEmma: Java code coverage for Eclipse. `http://www.eclemma.org/`, Sept. 2012.

[14] D. Myers and M. D. Storey. Using dynamic analysis to create trace-focused user interfaces for ides. In G. Roman and K. J. Sullivan, editors, *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 367–368. ACM, 2010.

[15] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *ASE*, 2013.

[16] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *Automated Software Engineering (Journal Special Edition)*, 2015. In submission. Available upon request.

[17] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[18] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software–artifact infrastructure repository. `http://sir.unl.edu/portal/`, Sept. 2006.

[19] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *NIST, RTI Project*, 7007(011), 2002.