Assembly Visualization and Analysis:

An Old Dog CAN Learn New Tricks!

Jennifer Baldwin, Del Myers, Margaret-Anne Storey, Yvonne Coady

University of Victoria

Abstract

Software engineering practices and tools have had a significant impact on productivity, time to market, comprehension, maintenance and evolution of software in general. The fact is that low-level systems have been largely overlooked in this arena, partially due to the complexities they offer and partially due to the inherent ”bare bones” nature of the domain. This lends an opportunity to explore the application of state-of-the-art high-level theories to low-level practice. Our initial investigations indicate that there are real issues that even experienced developers face. We believe modern tools can help in this domain, and this paper explores the ways in which we believe visualization will be of particular importance.

1. Introduction

Program comprehension is complex and time-consuming, particularly in manually tuned, low-level system codebases such as those written in assembly language. The current lack of adequate tool support for these kinds of systems further exacerbates this problem. Whereas engineers of higher-level systems quite often rely on tools for effectively navigating codebases and analyzing design, corresponding support for lower-level systems is severely lacking. The ultimate target of this research is to address open issues in the area of program comprehension of assembly code. We will do so by building a framework of tools that are designed to allow software architects, developers, support engineers, and testers to comprehend large, monolithic, complex system infrastructures (in excess of 100 KLOC and 10,000 branches). The resulting prototypes should allow developers to more easily maintain code or implement new features, even in legacy code, thereby enabling shorter turnaround times. Most importantly, as the assembly developer workforce ages, these tools should lessen issues linked to situations when the expert leaves the team.

1.1 Motivation

Though program understanding has received much attention from the research community, these approaches and corresponding tools only have limited application to large scale low-level systems. Even fundamental characteristics such as control flow and data flow are exceedingly hard to track at scale in the systems we propose to target. Though metamodels often serve to aid comprehension of higher-level systems, no corresponding metamodels currently exist for assembly, and there are still no widely accepted metamodels for C.

Irreducibility of Control Flow Graphs A number of software code analysis methods assume reducible control flow graphs. Unfortunately, assembly source code development leads to heavily optimized control flow with multiple entries and multiple exits. An example of such a control flow graph is depicted in Figure 1 [1]. This image was built with GraphViz [2] using IBM HLASM language, which contains no concept of subroutines, only conventions. The hexagon node is the entry point, and the trapezoid node is the exit point of the routine. Two shaded oval nodes represent calls to other subroutines and the other oval nodes represent linear blocks. There are several loops with multiple entries and multiple exits, a characteristic common to highly tuned low-level systems and malware programs.

Difficult Decomposition of Complex Control Flow Graphs The intertwining of multiple computation threads in a single source code module often results in complex control flow graphs for which decompositions are not easy, as shown in Figure 2 [1]. One of the key aspects of any software comprehension activity is the amount of information required to be understood. New methods and techniques targeting legacy code need to take this complexity into account by potentially grouping related control flow abstractions accordingly.

1.2 User Requirements

In March 2009, we visited the CA Labs location in Prague to perform an informal user study. We met with four developers

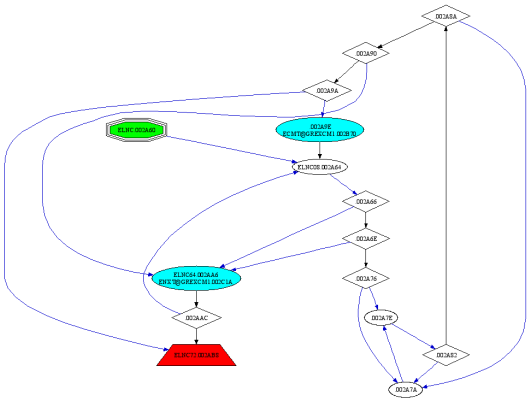

Figure 1. Irreducible Control Flow Graph.

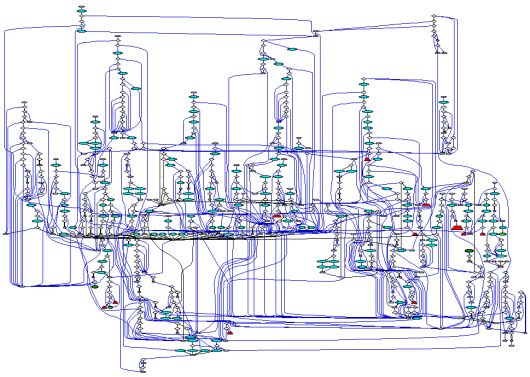

Figure 2. Complex Control Flow Graph.

from various backgrounds. They were able to freely discuss the issues they felt they had as well as what they believed was fundamental for their understanding. There seemed to be three separate areas of concern:

1. Development Tools
2. Debugging/Runtime Tools
3. Visualization/Comprehension Tools

The development tools would include syntax highlighting/checking, code completion and being able to search for references. Basically the common IDE support that Eclipse [3] provides for Java developers. Debugging tools would allow viewing values of registers at runtime, or from a log file, and stepping through the system. Finally, visualization and comprehension tools would include tools such as control flow through sequence diagrams, data flow tools and architectural diagrams.

Table 1 shows a summary of the results from the interviews with developers. DSECT refers to a dummy control section, similar to a struct in C, and CSECT refers to a control section or block of executable statements. To quickly

Developer	Issues
Assembly	Connections between modules, identification of subroutines, DSECT and CSECT support, register support.
Database	DSECT modification, debugging tools, data flow.
Eclipse Plugin	Syntax highlighting/checking, integration with mainframe, code completion, reference lists and graphs.
Assembly	Separating listings into source code and modules using each label/variable name.

Table 1. Summary of Developer Issues at CA Labs.

summarize this table, the first and last developer were both experienced with assembly but still had difficulties. These included the connections between modules, identifying a subroutine (non-existent in HLASM), understanding and locating the use of DSECTs and CSECTs, seeing how register usage is propagated throughout the system and understanding where variables and labels are used throughout modules in the system. The second team of developers were maintaining a mainframe database and had many issues debugging their system, since most errors were caused by a runtime modification to a dummy section. The third developer had already created an Eclipse plugin for his own use that included syntax highlighting and integration for syncing files the mainframe, but still lacked many features such as syntax checking, code completion and reference lists and graphs.

1.3 Current State-of-the-Art

The tools, that we are aware of, which are currently employed in industry include IDAPro [4], a disassembler and debugger, PaiMei [5] which is a reverse engineering framework, a runtime and memory analysis tool called Responder [6], and Visual Studio's debugger [7]. Other research efforts include GSPIM [8], which is used for visualization of low-level MIPS Assembly programming and simulation.

Other tools, including those not related to assembly, may hold promise in this domain. Some of these are types of visualizations such as distribution maps and terrain maps, as well as graph-based tools such as [9, 2, 10]. There are also other reverse engineering frameworks and exploration tools that could be useful [11, 12, 13, 14, 15]. Some other tools that exist for runtime analysis are the Visualization Execution Tool or VET [16], which helps programmers manage the complexity of execution traces, and also other tools for debugging [17, 18] and memory analysis [19]. Runtime tools are particularly important in helping developers identify memory leaks, buffer overflow, causes of segmentation faults, as well as understanding how registers and their values are used.

Concern mining might also be of interest in order to locate feature implementations, or concerns, within the code. Some of these tools include FINT [20], the Aspect Mining Tool (AMT)[21] and others [10, 22, 23, 24].

We believe the combination of features from these tools are needed to effectively assist developers in understanding

```
<sourcecode filename="MyDoom.ose">
    <functionEntryPoint address="4A462E" index="10"
          module="Email-Worm.Win32.Mydoom-d-dumped.ex_" name="start">
        <function address="4A7355" index="74" name="sub_4A7355">
            <call calladdress="4A736D" functionaddress="4A471C" index="13"
                  name="sub_4A471C"/>
            <call calladdress="4A737F" functionaddress="4A4CEE" index="24"
                  name="sub_4A4CEE"/>
            <call calladdress="4A738F" externalFile="Unknown"
                  functionaddress="4A1178" index="external"
                  name="gethostbyname"/>
            <call calladdress="4A739C" externalFile="KERNEL32.dll"
                  functionaddress="4A109C" index="external" name="Sleep"/>
            <call calladdress="4A73A8" functionaddress="4A8AB8" index="96"
                  name="memset"/>
            <call calladdress="4A73C2" externalFile="Unknown"
                  functionaddress="4A114C" index="external" name="htons"/>
            <call calladdress="4A73E1" externalFile="KERNEL32.dll"
                  functionaddress="4A10AC" index="external"
                  name="CreateThread"/>
            <call calladdress="4A73EE" functionaddress="4A73F8" index="75"
                  name="sub_4A73F8"/>
        </function>
    </functionEntryPoint>
</sourcecode>
```

Figure 3. XML used by the Sequence Diagram.

and maintaining low-level software. They can do so by providing visual assistance as well as customized views of a system. Our goal is to extract features or expand tools that could pertain to assembly code. Software exploration tools are abundant although none of them are currently geared towards assembly. Interestingly enough, many debuggers exist for assembly, yet developers are still asking for different features such as register usage and propagation, as well as being able to step back and forth through program execution. Finally, concern mining has not been applied to low-level languages such as assembly and could be an interesting avenue of research. Figure 2 may benefit from separation of control flows into chunks based on concerns, and it may help maintainers with no previous knowledge of the system to understand what a system does, apart from LOADs and GOTOs.

2. Proposed Solution

Our main goals in this work are to develop theories of comprehension in low-level systems, establish associated metrics capturing achievement in comprehension, incorporate these findings into an appropriate framework for tool support, and finally to measure impact on program comprehension tasks. Here we focus on our two prototype tools for high-level visualizations within this domain. First we consider static control flow through a sequence diagram viewer, followed by perspectives for memory and construct mapping.

2.1 Sequence Explorer

One of the more difficult challenges in understanding assembly code is following control flow. This is due to the inherent, unstructured nature of the code, as discussed earlier. Therefore, the first step in visualizing assembly was to create a sequence diagram, or basically a static call graph. This data was visualized by extending the Sequence Explorer from the CHISEL Lab at the University of Victoria [25].

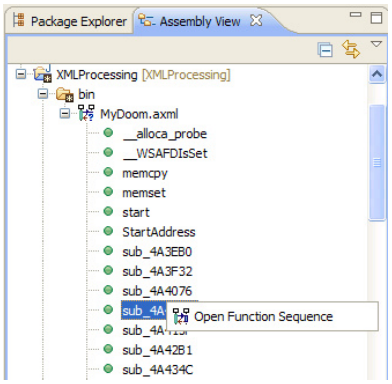

Figure 4. Tree View of XML Format.

First we will discuss the design of the sequence diagram tool and second, how it was extended for the purposes of this project.

2.1.1 Sequence Explorer Design

Much work in industry and in research has been spent implementing multiple instantiations of very similar visualizations for program control flow. Therefore, a need was seen for a reusable, interactive sequence diagram viewer in order to eliminate duplicate work. The Sequence Explorer view was built to this end.

The design of the view has two primary goals. First is model-independence, second is interactivity/navigability. Model-independence means that the viewer is not tied to any particular model or data format in its back-end. The viewer has been employed to visualize program control flow from various sources. Such sources include control flow of assembly language instructions (in this research), dynamic traces from instrumented Java programs [26], and call structures of static Java source code. This been accomplished by utilizing a framework compatible with the Eclipse JFace [27] viewer framework. This means that implementors must write some Java code in order to realize their application, but they are also abstracted far away from the details about how to draw the lines, boxes, and labels necessary for displaying the view.

The second goal of interactivity and navigability was inspired by the fact that sequence diagrams can quickly become very large and extremely complex. The viewer has integrated features to help overcome this problem. Some of the features are illustrated in section 2.1.2. A short listing of the features includes: animated layout, highlighting of selected elements and related sub-calls, grouping of related calls (such as loops), hiding/collapsing of call trees and package or module structures, customizable colors and labels for visual elements such as activation boxes and messages, keyboard navigation through components, and the ability to reset (focus) the sequence diagram on different parts of the call structure. These features were studied and evaluated in [26, 28].

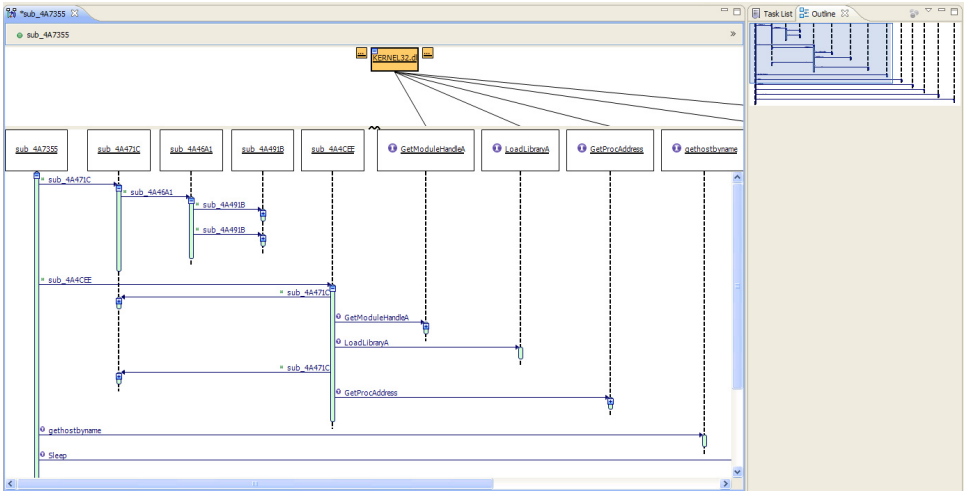

Figure 5. Sequence Diagram Viewer.

```
<softwarePackage name="CBT019">
  <sourceModule name="BLKSCAN" language="ASM" lastAddress="000D20"
        isIgnored="false" description="">
    <controlSection name="BLKSCAN" startAddress="00000" length="00D24"
          stmt="80"/>
    <dummySection name="IHADCB" startAddress="00000" length="00058"/>
  </sourceModule>
  <sourceModule name="CHECKPVT" language="ASM" lastAddress="0001C8"
        isIgnored="false" description="">
    <controlSection name="CHECKPVT" startAddress="00000" length="001D2"
          stmt="65"/>
    <dummySection name="CVT" startAddress="00000" length="00500"/>
    <dummySection name="CVTXTNT1" startAddress="00000" length="0000C"/>
    <dummySection name="CVTVSTGX" startAddress="00000" length="00050"/>
    <dummySection name="CVTXTNT2" startAddress="00000" length="00084"/>
  </sourceModule>
</softwarePackage>
```

Figure 6. XML used by the Visualiser.

```
Group:CBT019 Member:BLKSCAN Size:000D20 Tip:CBT019.BLKSCAN
Group:CBT019 Member:CA Size:0001C8 Tip:CBT019.CA Group:CBT019
Member:CHECKPVT Size:0001C8 Tip:CBT019.CHECKPVT Group:CBT019


Stripe:CBT019.BLKSCAN Kind:BLKSCAN Offset:00000 Depth:00D24
Stripe:CBT019.BLKSCAN Kind:IHADCB Offset:00000 Depth:00058
Stripe:CBT019.CHECKPVT Kind:CHECKPVT Offset:00000 Depth:001D2
Stripe:CBT019.CHECKPVT Kind:CVT Offset:00000 Depth:00500
```

Figure 7. Visualiser Files (content.vis and markup.vis)

2.1.2 Sequence for ASM

Figure 3 shows the XML format, translated from an IDAPro plugin's output used for the static call graph, or sequence diagram. This XML data contains the file we are looking at, which functions are defined within it, and what calls that function makes, as well as their locations and names. This particular XML represents MyDoom, a computer virus for Microsoft Windows.

The tree viewer in Figure 4 shows all of the defined functions that a user can select. Once the user has selected a function, a sequence diagram such as Figure 5 will be seen.

This screenshot shows that the user has selected the function *sub_4A7355*, and can then expand the function calls they are interested in to see what calls that function makes and so on. Any of the functions that have an *I* icon next to them are imported functions, meaning they are located in another module. At the top of the figure, there is a package diagram which shows which module this function is defined in. Here we can see that many of the imported functions come from the *KERNEL32.dll* file. When an imported function is selected, the XML file corresponding to it, if it exists, is parsed and the information added to the diagram. We can also see the thumbnail view in the outline pane. The viewer also allows users to set any of the calls as the new root of the diagram and reset the root to the caller of that function. These are available as right click options on the subroutine's lifeline. Additionally, there is a breadcrumb view at the top of the diagram so the user can select any function along the path to navigate through the calls.

Future work for the assembly extension of this tool will include being able to filter the control flow so that it only shows areas of interest. PaiMei is able to filter control flow information so if we can combine this filtering with the output from IDAPro, then we can apply this filtering ability to the assembly extension since it has already been built into the Sequence Explorer framework.

2.2 Visualiser

We will now discuss the background of the Visualiser tool and how it was used in the context of an assembly system.

2.2.1 Visualiser Background

The Visualiser [29] is an extensible Eclipse plugin, originally part of AJDT, that can be used to visualize anything

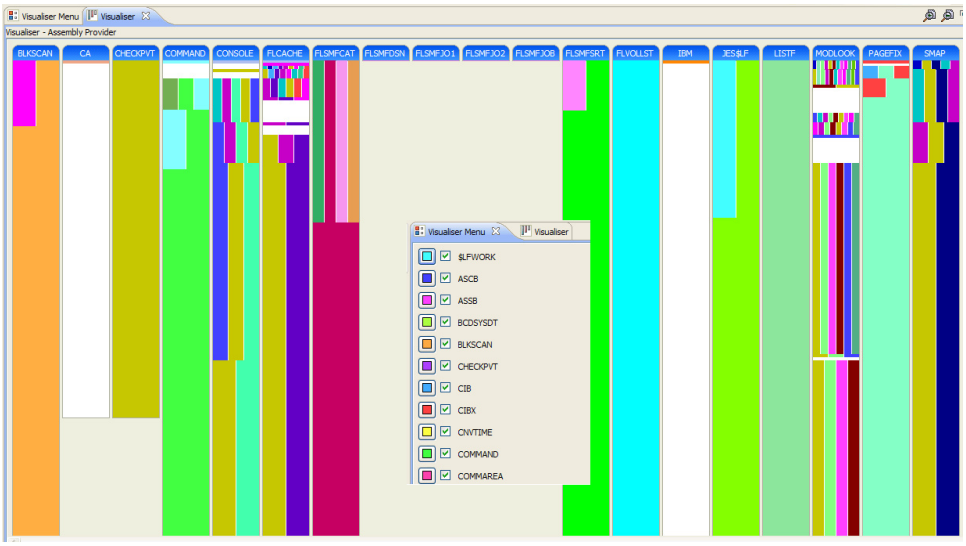

Figure 8. Visualization of CBT019.

that lends itself to a 'bars and stripes' style representation. It began as the Aspect Visualiser, which was used to visualize how aspects were affecting classes in a project. It did so by showing each class as a bar, with its length corresponding to the classes' length in lines of code. Each aspect was then color-coded and their locations drawn within the bar based on their location and number of affected lines of code (or lines of code in the aspect itself). The Visualiser provides extension points and there are a few publicly available providers, including those for Google searches and CVS history. We have also used it before in the context of tool support for systems in [30].

2.2.2 Visualiser for ASM

We extended the Visualiser by creating another provider to show a high-level view of certain constructs in assembly language, as well as navigation to those constructs in the code. For example, trying to see where DSECTs are defined and used at a high level was quite difficult for a CA developer, who was using ctrl-F to find all of its uses. Since most assembly programs are one large module, it will be important to split the bars up first by module, then by subroutine, then into the CSECTs, DSECTs etc. We envision building an interactive treemap combined with bars and stripes to provide this interactivity to move from large to small granularity.

For this particular example, we used an open source MVS program called CBT019 [31], or FOOD LION Utilities from John Hooper. We have selected this program since it is comparable in size to many of those at CA labs. The XML that was used in this example is shown in Figure 6. This XML file is transformed to two files that the Visualiser is able to read, these are shown in Figure 7. The top file, *content.vis*, draws all of the bars and the second file, *markup.vis*, places the stripes throughout those bars. The visual output for this XML file is shown in Figure 8. The menu, which shows each of the CSECTs and DSECTs, is shown in the center of the screenshot. Each bar represents a module with its length equal to its size in memory. Each CSECT and DSECT are also color-coded and their size and location correspond to that in memory. This example represents a memory view of the system, not a source view, which would mean that lengths and locations are based on lines of code. Since neither a source code view, or memory address view is a complete solution, either a combined view or two separate views will be required. The usage of these constructs, in addition to their definition, is important and will also need to be considered.

3. Future Work and Conclusions

Tool support, as we know it today, was not available to developers who worked primarily with assembly and therefore it is not surprising that it is not widely available. Furthermore, it is unclear if high level program comprehension theories and tools map directly into this domain. This lends an opportunity to explore the application of state-of-the-art high-level theories to low-level practice. Our initial investigations indicate that our approach is feasible, and exposes many possible avenues for future research. Such avenues include exploring further tools and metrics, other languages and codebases, and ties to higher-level theories.

We believe that assembly software comprehension tools will aid developers in many areas. Increased comprehension will enable shorter turnaround times for maintenance and modifications of software. This coupled with navigational and development tools can also support easier, faster, and more reliable implementation of new features in legacy software. Another important factor is avoiding issues when an expert leaves the team. The fact is that new generations of developers are accustomed to a certain level of tool support,

and by accepting and using it, they will reap the benefits we believe exist.

Acknowledgments

We would like to thank Radek Marik at CA Labs, for providing data and feedback necessary for the visualiser, and David Ouellet and Martin Salois from DND for their involvement in creating the Sequence Explorer extension.

References

[1] R. Marik, "GREX Architecture - Package Comprehension Report," CA Labs, Tech. Rep., 2009.

[2] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Softw. Pract. Exper.*, vol. 30, no. 11, pp. 1203–1233, 2000.

[3] Eclipse.org home. http://www.eclipse.org/

[4] IDA Pro Disassembler. http://www.hex-rays.com/idapro/

[5] P. Amini, "PaiMei - Reverse Engineering Framework," in *RECON '06: Reverse Engineering Conference*, Montreal, Canada, 2006.

[6] ResponderPRO. https://www.hbgary.com

[7] Visual Studio. http://msdn.microsoft.com/en-gb/vstudio/default.aspx

[8] P. Borunda, C. Brewer, and C. Erten, "GSPIM: graphical visualization tool for MIPS assembly programming and simulation," in *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2006, pp. 244–248.

[9] N. Synytskyy, R. C. Holt, and I. Davis, "Browsing software architectures with LSEdit," in *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 176–178.

[10] J. Bohnet and J. Döllner, "Visual exploration of function call graphs for feature location in complex software systems," in *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*. New York, NY, USA: ACM, 2006, pp. 95–104.

[11] M.-A. D. Storey, K. Wong, and H. A. Müller, "Rigi: a visualization environment for reverse engineering," in *ICSE '97: Proceedings of the 19th international conference on Software engineering*. New York, NY, USA: ACM, 1997, pp. 606–607.

[12] A. Desnos, S. Roy, and J. Vanegue, "ERESI: a kernel-level binary analysis framework," in *SSTIC '08: Symposium sur la Securite des Technologies de l'Information et Communications*, Rennes, France, 2008.

[13] Q. Wang, W. Wang, R. Brown, K. Driesen, B. Dufour, L. Hendren, and C. Verbrugge, "EVolve: an open extensible software visualization framework," in *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*. New York, NY, USA: ACM, 2003, pp. 37–ff.

[14] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen, "SHriMP views: an interactive environment for information visualization and navigation," in *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*. New York, NY, USA: ACM, 2002, pp. 520–521.

[15] M. Eichberg, M. Haupt, and M. Mezini, "The SEXTANT Software Exploration Tool," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 753–768, 2006.

[16] M. McGavin, T. Wright, and S. Marshall, "Visualisations of execution traces (VET): an interactive plugin-based visualisation tool," in *AUIC '06: Proceedings of the 7th Australasian User interface conference*, 2006, pp. 153–160.

[17] Sysersoft. http://www.sysersoft.com/

[18] SoftICE. http://en.wikipedia.org/wiki/SoftICE

[19] Corelabs site. http://corelabs.coresecurity.com/

[20] M. Marin, L. Moonen, and A. van Deursen, "FINT: Tool Support for Aspect Mining," in *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 299–300.

[21] J. Hannemann and G. Kiczales, "Overcoming the Prevalent Decomposition in Legacy Code," in *Workshop on Advanced Separation of Concerns, Int'l Conf. Software Engineering (ICSE)*, 2001.

[22] P. Tonella and M. Ceccato, "Aspect Mining through the Formal Concept Analysis of Execution Traces," in *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 112–121.

[23] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 301.

[24] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 210–224, 2003.

[25] C. Bennet, D. Myers, M.-A. Storey, and D. German, "Working with 'monster' traces: Building a scalable, usable, sequence viewer." in *In Proceedings of the 3rd International Workshop on Program Comprehension Through Dynamic Analysis (PCODA)*, Vancouver, Canada, 2007, pp. 1–5.

[26] C. Bennett, D. Myers, M. Storey, D. German, D. Ouellet, M. Salois, and P. Charland, "A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 4, 2008.

[27] JFace - Eclipsepedia. http://wiki.eclipse.org/index.php/JFace

[28] C. Bennett and M.-A. Storey, *Tool features for understanding large reverse engineered sequence diagrams*. University of Victoria, 2008.

[29] The Visualiser. http://www.eclipse.org/ajdt/visualiser/

[30] J. Baldwin and Y. Coady, "Adaptive Systems Require Adaptive Support - When Tools Attack!" in *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*, 2007, p. 10.

[31] CBT Tape - MVS Freeware. http://www.cbttape.org