Evaluation and usability of programming languages and tools (PLATEAU)

Conference Paper · January 2010

DOI: 10.1145/1869542.1869605 · Source: DBLP

3 authors, including:

Craig Anslow
Victoria University of Wellington
92 PUBLICATIONS **598** CITATIONS

Evaluation and Usability of Programming Languages and Tools (PLATEAU) PLATEAU 2009

Technical Report
ECSTR10-12, July 2010
ISSN 1179-4259

School of Engineering and Computer Science
Victoria University of Wellington, New Zealand

Craig Anslow, Shane Markstrum, Emerson Murphy-Hill (Editors)
Email: craig@ecs.vuw.ac.nz, shane.markstrum@bucknell.edu, emhill@cs.ubc.ca

1 Introduction

The Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) was co-located with the Onward and OOPSLA conferences in Orlando, Florida, USA from 25-29 October 2009. This technical report is a summary of the workshop including the proceedings.

2 Scope

Programming languages exist to enable programmers to develop software effectively. But how *efficiently* programmers can write software depends on the usability of the languages and tools that they develop with. The aim of this workshop is to discuss methods, metrics and techniques for evaluating the usability of languages and language tools. The supposed benefits of such languages and tools cover a large space, including making programs easier to read, write, and maintain; allowing programmers to write more flexible and powerful programs; and restricting programs to make them more safe and secure. We plan to gather the intersection of researchers in the programming language, programming tool, and human-computer interaction communities to share their research and discuss the future of evaluation and usability of programming languages and tools. We are also interested in the input of other members of the programming research community working on related areas, such as refactoring, design patterns, program analysis, program comprehension, software visualization, end-user programming, and other programming language paradigms.

At the Programming Languages Grand Challenges panel at POPL 2009, Greg Morrisett claimed that one of the great neglected areas in programming languages research is the bridge between programming languages and human-computer interaction: the evaluation of the usability of programming languages and tools. This is evident by the recent research programs of major languages conferences such as POPL, PLDI, OOPSLA, and ECOOP. The object-oriented conferences tend to have at least one or two papers in the areas of corpus analysis or evaluation methodologies, but the authors of the papers seem to avoid using the results of their studies to make conclusions about the languages or tools themselves. Software engineering and human-computer interaction conferences tend to have better support of language usability analysis (CHI 2009 had three tracks that showcase research in this direction), but have limited visibility to the programming languages community.

This workshop aims to begin filling that void by developing and stimulating discussion of usability and evaluation of programming languages and tools with respect to language design and related areas. We will consider: empirical studies of programming languages; methodologies and philosophies behind language and tool evaluation; software design metrics and their relations to the underlying language; user studies of language features and software engineering tools; visual techniques for understanding programming languages; critical comparisons of programming paradigms,

such as object-oriented vs. functional; and tools to support evaluating programming languages. We have two goals:

1. Develop a research community that shares ideas and collaborates on research related to the evaluation and usability of languages and tools.

2. Encourage the languages and tools communities to think more critically about how usability affects the design and adoption of languages and tools.

3 Committees

The workshop was organised by members working on the New Zealand Software Process and Product Improvement (SPPI) Project. The program committee comprised of international researchers from the areas of programming languages, software engineering, and human computer interaction.

3.1 Organising Committee

- Craig Anslow - is a PhD student in the School of Engineering and Computer Science at Victoria University of Wellington, New Zealand. His PhD research focuses on *Multi-touch Table User Interfaces for Collaborative Software Visualization* and is supervised by James Noble and Stuart Marshall. Craig has experience in building applications to support the evaluation of programming languages using information visualization and multi-touch techniques.

- Shane Markstrum - recently defended his PhD in the Department of Computer Science, University of California, Los Angeles, USA. His PhD dissertation is titled *Enforcing and Validating User-Extensible Type Systems* and was supervised by Todd Millstein. Shane has extensive experience in building domain-specific languages for type systems and building plugins for Eclipse that focus on language-oriented features. Shane has joined the faculty of Bucknell University as an assistant professor of Computer Science.

- Emerson Murphy-Hill - is currently a postdoctoral fellow at the University of British Columbia in the Software Practices Lab with Gail Murphy, researching how software developers find and adopt software tools. He recieved is Ph.D. from Portland State University in 2009. His research interests include human-computer interaction and software tools.

3.2 Program Committee

We thank out program committee for reviewing the papers and giving valuable feedback on the design of our workshop.

- Craig Anslow - Victoria University of Wellington, New Zealand
- Andrew Black - Portland State University, USA
- Larry Constantine - University of Madeira, Portugal
- Jeff Foster - University of Maryland, College Park, USA
- Bob Fuhrer - IBM Research, USA
- Donna Malayeri - Carnegie Mellon University, USA
- Shane Markstrum - University of California, Los Angeles / Bucknell University, USA
- Stuart Marshall - Victoria University of Wellington, New Zealand
- Todd Millstein - University of California, Los Angeles, USA
- Emerson Murphy-Hill - University of British Columbia, Canada
- James Noble - Victoria University of Wellington, New Zealand
- Ewan Tempero - University of Auckland, New Zealand

4 Workshop Program

The programme was divided into a keynote, paper sessions, discussions of the papers, and a panel. The workshop began with a Welcome and Introduction by the organising committee.

Larry Constantine then gave an interesting keynote that covered the usability of programming tools and inspired participants to start thinking more about designing tools to increase usability. A couple of exercises were conducted to illustrate ways in which participants could increase the usability of tools.

11 papers were submitted and 8 papers were accepted for presentation. The papers covered visualization and analysis and why they affect the usability of language, types and APIs and why we might not want them, and non-traditional programming models and the problem with user studies with them. Discussion sessions were followed at the end of each paper session which allowed the audience to engage more about the topic.

The workshop concluded with an exciting panel with industry programming languages and HCI veterans as panelists and the topic of discussion was "Bridging the gap between Programming Languages and HCI". The workshop was moderated by Andrew Black.

Breaks and lunch provided a good opportunity for discussion at the workshop. Lunch was had at a nearby restaurant. Concluding the workshop attendees mixed with each other at the OOPSLA Reception.

Table 1: PLATEAU Workshop Program.

0830-0900	Welcome and Introductions
0900-1000	Keynote - Larry Constantine "User Experience Design for Programming Languages and Tools"
1000-1030	Morning Break
Session 1. Visualization and Analysis (why they affect the usability of language)	
1030-1050	Jennifer Baldwin, Del Myers, Margaret-Anne Storey, and Yvonne Coady. Assembly Visualization and Analysis: An Old Dog CAN Learn New Tricks!
1050-1110	Yit Phang Khoo, Jeff Foster, Michael Hicks and Vibha Sazawal. Triaging Checklists: a Substitute for a PhD in Static Analysis
1110-1130	Christine A. Halverson and Jeffrey Carver. Climbing the Plateau: Getting from Study Design to Data that Means Something
1130-1200	Discussion on the role and value of data presentation/visualization in evaluating languages and tools
1200-1330	Lunch Break
Session 2. Types and APIs (why we might not want them)	
1330-1350	Akira Tanaka. Language and Library API Design for Usability of Ruby
1350-1410	Mark Daly, Vibha Sazawal and Jeffrey Foster. Work In Progress: an Empirical Study of Static Typing in Ruby
1410-1430	Stefan Hanenberg. What is the Impact of Type Systems on Programming Time? - First Empirical Results
1430-1500	Discussion on the role and value of user studies vs. code surveys and how to evaluate the effectiveness of types and APIs
1500-1530	Afternoon Break
Session 3. Non-traditional Programming Models (the problem with user studies)	
1530-1550	Meredydd Luff. Empirically Investigating Parallel Programming Paradigms: A Null Result
1550-1610	Nan Zang. End User Programming Opportunities and Challenges on the Web
1610-1625	Discussion on evalution strategies for fringe or non-traditional programming models
1625-1630	Organizers report and participant feedback
1630-1730	Panel - "Bridging the Gap between Programming Languages and Human-Computer Interaction"

5 Keynote

Speaker Larry Constantine. Professor in the Department of Mathematics and Engineering at the University of Madeira, Portugal and Director of the Laboratory for Usage-centered Software Engineering.

Title User Experience Design for Programming Languages and Tools.

Abstract Users are often at best an afterthought among software developerseven when they themselves are the users! Usability of programming languages and tools is all too often equated with raw functionality or casually dismissed as a matter of mere syntactic sugar. In one sense the usability issues for programming languages and tools are nothing special. Whether its a UML modeling tool or a Web 2.0 ERP system, on the other side of the screen is a human eye and hand coordinated by a human brain. The same broad principles and specific techniques of sound interaction design apply.

This keynote by an award-winning interaction designer and design methodologist attempts to frame the issues in the usability of the tools we use, exploring the dimensions of user experience in programming languages and tools, as well as examining what might be unique or special about our experience as users. Specific recommendations and proposals for improving the usability and user experience of languages and tools are presented.

Bio Larry Constantine, IDSA, is a Professor in the Department of Mathematics and Engineering at the University of Madeira where he teaches in the dual-degree program that he helped organize with Carnegie-Mellon University in the United States. He heads the Laboratory for Usage-centered Software Engineering, a research and development group dedicated to making technology more useful and usable. One of the pioneers of modern software engineering, he is an award-winning designer and author, recipient of the 2009 Stevens Award for his contributions to design and design methods, and a Fellow of the Association for Computing Machinery.

6 Panel

Are today's programming languages as easy to read and write as they could be? Are they good at communicating the programmer's intent to others? Can programing language designers learn anything from the Human-Computer Interaction community that will help us design better languages?

We held a workshop on this area entitled: *"Bridging the Gap Between Programming Languages and Human-Computer Interaction"*. The workshop was moderated by Andrew Black and had three distinguished panelists who discussed their ideas around this issue, and opened the discussion to the workshop participants.

- Larry Constantine - Professor, University of Madeira, Portugal
- Dan Ingalls - Distinguished Engineer, Sun Microsystems Laboratories, USA
- Robert Biddle - Professor of Human-Computer Interaction, Carleton University, Canada

Assembly Code Visualization and Analysis:

An Old Dog CAN Learn New Tricks!

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

University of Victoria

{jbaldwin, delmyers, mstorey, ycoady}@cs.uvic.ca

Abstract

Software engineering practices and tools have had a significant impact on productivity, time to market, comprehension, maintenance and evolution of software in general. 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. The fact is that anyone can understand a few lines of assembly, even hundreds, but when you move into the tens of thousands or more, most people will require additional cognitive support. 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, such as the overwhelming size but also obfuscation of system function in malware. 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 assembly code comprehension. This goal will require a theory of how assembly programmers comprehend code and the challenges which they face. Second, we will build prototype tools that will be evaluated at a later stage of our work. This framework of tools will be designed to allow software architects, developers, support engineers, testers and security analysts to better 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, and should also speed up the analysis of malware, thereby enabling shorter turnaround times. Most importantly, as the assembly developer workforce ages, these tools aim to 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 widely accepted metamodels currently exist for assembly or even 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 a program written in the IBM HLASM code [3], 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, which means that this graph cannot be reduced to a single node and is irreducible.

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

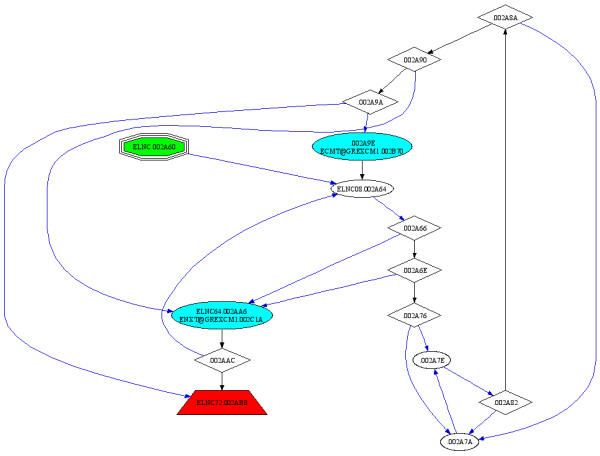


Figure 1. Irreducible Control Flow Graph.

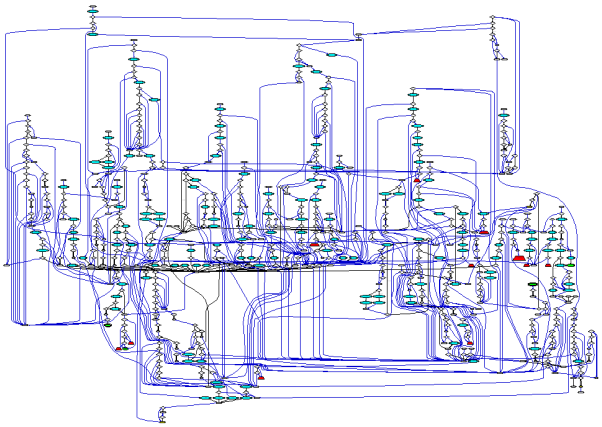


Figure 2. Complex Control Flow Graph.

understood. New methods and techniques targeting legacy code need to take this complexity into account by potentially grouping related control flow abstractions accordingly.

These images are not meant to show bad programming structure, but instead a structure that is almost unavaoidable in assembly programming. While these particular images are provided by CA for HLASM, this same principle holds for applications written in different assembly dialects. For example, developers at the Department of National Defense (DND) have issues with the intentional obfuscation in malware, so that it is exceedlingly difficult to pinpoint the security threat it contains.

This paper is organized as follows: Section 2 describes an initial user study. Section 3 discusses the related tools we have studied while Section 4 introduces the prototype extensions of these tools in regards to assembly language. Section 5 ends with future work.

2. Initial Findings: Assembly Programmer Tool Needs

In March 2009, we visited the CA PTC (Prague Technology Center). CA PTC is known primarily for its mainframe computer and distributed computing applications and solutions used by businesses. Here we met with five developers 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. Their stories are summarized below.

First Engineer Rob[1] was an experienced assembly developer who was working with an extremely large module. We saw how he worked with the code and he also told us what the most important tools he wanted were. These include connections between modules, analyzing subroutines (HLASM does not have functions, only coding conventions), support for dummy sections (DSECTs) and control sections (CSECTs), as well as register usage. 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.

In order to find DSECTs, Rob was using text search. There was no way to see where they were defined or used at a high level, and no navigation to them.

Second Engineers The second interview we had was with two engineers, David and Joe, that both worked on a database written in assembly. When bugs occured in the system, usually it was due to an instruction modifying a DSECT. When this occured, they had to use a crossreference tool that finds everything that modifies the DSECT, and it has some shortcomings.

David and Joe wanted something similar to a debugging tool, which would allow them to jump through modules to follow data flow by using a log file. Navigation would allow them to move back and forth with their selections and also back and forth with how the program ran, with the addition of breadcrumbs to show this. They also needed to know the values in each register. Additionally, they wanted to have architectural diagrams that developers could collaborate on. Figure 3 shows a mockup of this tool.

[1] The names used in this study are fictitious.

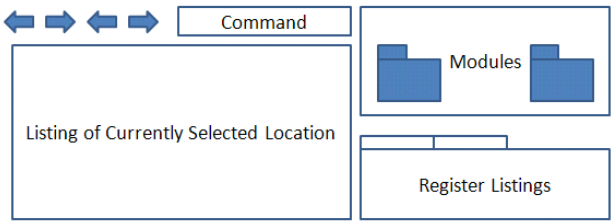


Figure 3. Mockup UI Design for Debugging Tool

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	Separate listings into: - source code - modules using each label/variable name

Table 1. Summary of Developer Issues at CA PTC.

Third Engineer Alex had created an Eclipse plugin that provided syntax highlighting for assembly, as well as being able to upload/download files from the mainframe and execute them. It also included the ability to view the log file as well as the outline of the assembly code (macros, DSECT, CSECT).

Future work that Alex wanted for this tool included code completion and syntax support in the editor, so the code would not have to run on the mainframe first. Also being able to search and graph references for where the symbols are used (or defined).

Fourth Engineer Bill had been given the task of trying to understand a huge assembly system called `reportbroker`, which has 493 modules. He had written a tool that separates data from the listing into two different files, one for the source code and one for the usage of each label, varname etc by module. This was because it was important for him to know which modules to look at when he was dealing with a specific label or variable name.

Summary There seemed to be three separate areas of concern:

- Development Tools
- Debugging/Runtime Tools
- 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 [4] 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 those for control flow, data flow tools and architectural diagrams.

Table 1 shows a summary of the results from the interviews with developers.

3. Related Work

This section will discuss some of the existing tools that we have investigated during this work. This is only a subset of those tools. Many others, including those for dynamic visualizations, were reviewed in [5]. First we discuss at a broad level, many of the tools we have investigated. We then focus on two tools that appear to match the needs of the users as identified in the previous section of the paper. The first is the Sequence Explorer tool, and the second is the Visualiser.

3.1 A Subset of Tools

Some tools that are currently employed in industry for assembly include IDAPro [6], a disassembler and debugger, PaiMei [7], which is a reverse engineering framework, Responder [8], a runtime and memory analysis tool, and Visual Studio's debugger [9]. Other research efforts include GSPIM [10], which is used for visualization of low-level MIPS assembly programming and simulation, and BitBlaze [11], a binary analysis platform to analyze, understand, and develop defenses against malicious code.

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 [2, 12, 13]. There are other reverse engineering frameworks and exploration tools that could be useful [14, 15, 16, 17, 18]. Other tools that exist for runtime analysis are the Visualization Execution Tool or VET [19], which helps programmers manage the complexity of execution traces, and other tools for debugging [20, 21] and memory analysis [22]. 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 [23], the Aspect Mining Tool (AMT) [24] and others [13, 25, 26, 27].

We believe a combination of features from these tools are needed to effectively assist developers in understanding 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, but only a handful are 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 and could be an interesting avenue of research. Figure 2 may benefit from separation of control flows into chunks based on concerns, providing a better way for maintainers to understand a system than trying to figure out intertwined `LOAD`s and `GOTO`s.

3.2 Sequence Explorer

One of the more difficult challenges in understanding assembly code is following control and data 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 static call graph. This data was visualized by extending the Sequence Explorer [28] from the CHISEL Lab at the University of Victoria. We selected this tool because it is available as open source and thus open to extension, which is the only one as far as we know.

3.2.1 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. The first is model-independence, and the 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 [5], and call structures of static Java source code. This has been accomplished by using a framework compatible with the Eclipse JFace [29] 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 4.1. 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 [5, 30].

3.3 Visualiser

In order to visualize certain aspects of assembly code at a high-level, we needed some sort of tree map. The visualiser is not a tree map, but somewhat similar with its use of colored stripes and blocks. It is also freely available and easily extended, which is why it was selected as a first step towards a scalable tool for this purpose.

3.3.1 Visualiser Background

The Visualiser [31] is an extensible Eclipse plugin, originally part of AJDT, that can be used to visualize anything 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 its 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 [32].

4. Towards More Effective Assembly Tools

Our eventual goals of 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 the impact on program comprehension tasks. Since this work is in its early stages, here we simply focus on our two prototype tools for high-level visualizations within this domain: static control flow through the extension of a sequence diagram viewer, and perspectives for memory and construct mapping.

4.1 Control Flow for ASM

To create a control flow tool for assembly, we needed the data for the control flow as well as the tool itself. The call data was retrieved for X86 using an IDAPro plugin built by DND. For the tooling, we built an extension of the Sequence Explorer discussed in the previous section. This tool presents the user with all of the functions defined in a module within a tree in Eclipse, once the user has selected a function, a control flow diagram such as Figure 4 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. 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 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 on the right hand side. The viewer 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 are breadcrumbs at the top of the diagram so the user can select any function along the path to navigate through the calls. When the user is finished navigating, they have the option to save the state of the diagram.

Future work for the assembly extension of this tool will include being able to filter the control flow so that it only

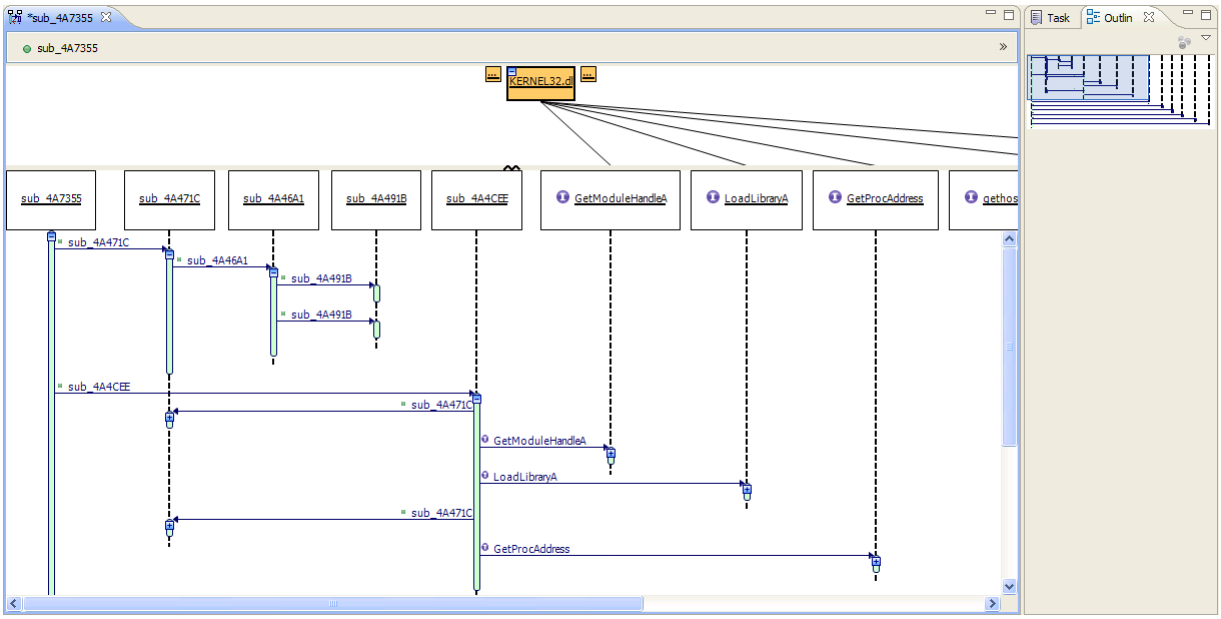


Figure 4. Control Flow Viewer.

shows areas of interest and to be able to load control flow information on demand from IDAPro, so that it is no longer static. PaiMei [7] is able to filter control flow information and the filtering ability has already been built into the Sequence Explorer framework. Therefore, we need to combine this filtering with the output from the IDAPro [6] plugin. As for dynamic control flow, IDAPro provides the API to easily build extensions, and DND has already built an example plugin to provide interaction with Java programs.

4.2 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 text search to find all of its uses.

For this particular example, we are visualizing an open source MVS program called CBT019 [33], otherwise known as FOOD LION Utilities from John Hooper. We have selected this program since it is comparable in size to many of those at CA. This example represents a memory view of the application using listing files created during system compilation. The data gathered from the listing files is first transformed into an XML file, which is then transformed to the format the Visualiser understands. The visual output for this XML file is shown in Figure 5. The menu, which shows each of the CSECTs and DSECTs, is shown in the center of the screenshot. Each bar (or column) represents a module with its length equal to its size in memory. Each CSECT and DSECT is also color-coded and its size and location correspond to that in memory. This view allows developers to see at a high-level, where all of the DSECTs and CSECTs are located and also how much memory of the entire system, they consume. Developers can also interact with the diagram by double clicking on each colored segment to navigate to that DSECT or CSECT. There are also additional options provided by the Visualiser itself, such as zooming in and out, fit to view, limit view to affected bars, and group and member views (when packages are present).

This example is 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. Additionally, since most assembly programs are one large module, with the current tool, this will appear as one long bar with many colored blocks. Therefore, it will be important to provide some ease of exploration by splitting 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.

5. Future Work

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 still nearly non-existent. 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. We intend to do so by using these new tools we develop to elicit require-

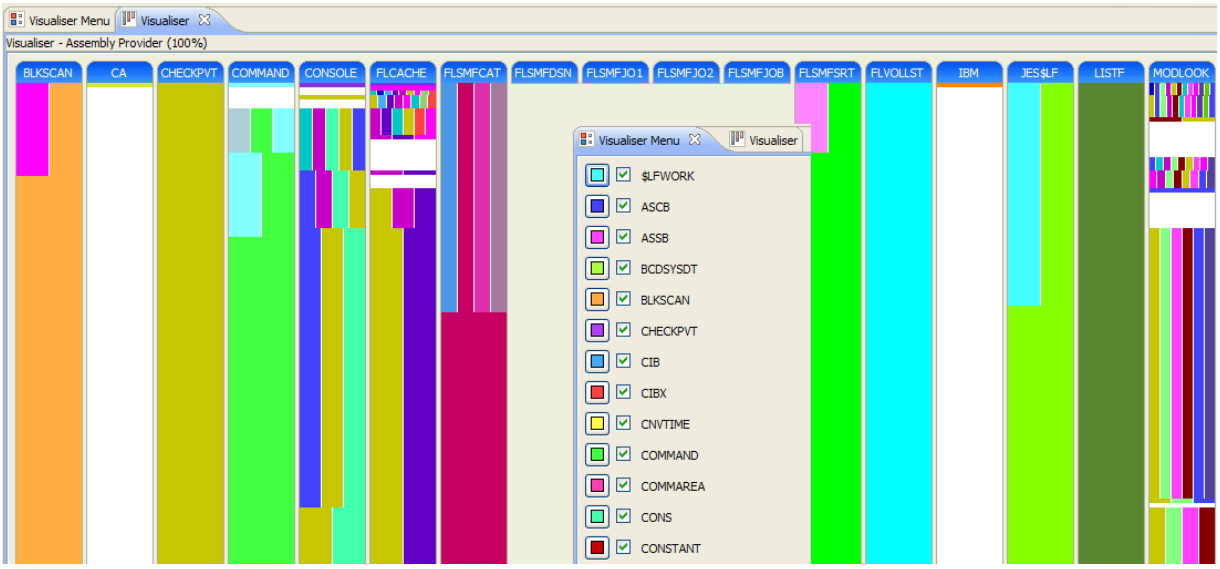


Figure 5. Visualization of CBT019.

ments. Since the programmers may not ask for such tools, having never used them before, we can continue to adapt them based on the challenges we observe. Finally, we can see if our tools address these challenges by performing case studies with subject software systems followed by user studies with developers. 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. These theories will include refining program comprehension theories such as those found in [34, 35, 36, 37].

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 David Ouellet and Martin Salois from Defence Research and Development Canada (DRDC) for their involvement in creating the Sequence Explorer extension; and Radek Marik at CA Labs, for providing data and feedback necessary for the visualiser. This work was funded by DRDC contract W7701-82702/001/QCA and by CA Labs.

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] High Level Assembler and Toolkit Feature. http://www-01.ibm.com/software/awdtools/hlasm/

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

[5] C. Bennett, D. Myers, M.-A. 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.

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

[7] P. Amini, “PaiMei - Reverse Engineering Framework,” in *RECON ’06: Reverse Engineering Conference*, Montreal, Canada, 2006.

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

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

[10] 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.

[11] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “Bitblaze: A new approach to computer security via binary analysis,” in *ICISS ’08: Proceedings of the 4th International Conference on Information Systems Security*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 1–25.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

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

[30] C. Bennet, "Tool features for understanding large reverse engineered sequence diagrams," Master's thesis, University of Victoria, 2008.

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

[32] 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.

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

[34] A. von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[35] P. Kruchten, "The 4+1 view model of architecture," *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, 1995.

[36] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," *Commun. ACM*, vol. 37, no. 5, pp. 72–82, 1994.

[37] M.-A. D. Storey, F. D. Fracchia, and H. A. Mueller, "Cognitive design elements to support the construction of a mental model during software visualization," in *WPC '97: Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 17.

Triaging Checklists: a Substitute for a PhD in Static Analysis

Khoo Yit Phang Jeffrey S. Foster Michael Hicks Vibha Sazawal
University of Maryland, College Park
{khooyp,jfoster,mwh,vibha}@cs.umd.edu

Abstract

Static analysis tools have achieved great success in recent years in automating the process of detecting defects in software. However, these sophisticated tools have yet to gain widespread adoption, since many of these tools remain too difficult to understand and use. In previous work, we discovered that even with an effective code visualization tool, users still found it hard to determine if warnings reported by these tools were true errors or false warnings. The fundamental problem users face is to understand enough of the underlying algorithm to determine if a warning is caused by imprecision in the algorithm, a challenge that even experts with PhDs may take a while to achieve. In our current work, we propose to use *triaging checklists* to provide users with systematic guidance to identify false warnings by taking into account specific sources of imprecision in the particular tool. Additionally, we plan to provide *checklist assistants*, which is a library of simple analyses designed to aid users in answering checklist questions.

1. Introduction

In recent years, the research and industrial communities have made great strides in developing sophisticated software defect detection tools based on *static analysis*. Such tools analyze program source code with respect to some explicit or implicit specification, and report potential errors in the program. Static analysis tools show great promise in automating defect detection: new analysis techniques and tools are now regularly reported in the research literature as having found bugs in significant open-source software [Ayewah et al. 2007; Engler and Ashcraft 2003; Engler et al. 2000; Foster et al. 2002; Hovemeyer and Pugh 2004; Naik et al. 2006; Shankar et al. 2001]. Microsoft routinely uses tools to find bugs in production software [CSE; Das 2006] and other large software houses, such as Google [Ayewah et al. 2007; Ruthruff et al. 2008] and EBay [Jaspan et al. 2007], are beginning to follow suit.

Despite these successes, most static analysis tools remain limited in formal adoption, particularly tools that use sophisticated algorithms [Ayewah et al. 2008; Ayewah and Pugh 2008]. In our opinion, one of the key reasons is that many tool designers today fail to appreciate that the human user is an essential component of the defect detection process. A tool can output a list of possible errors, but the user has to determine if a reported warning is actually an error and, if so, how to fix it. In fact, we consider a tool to be effective only if it can successfully collaborate with the user to locate actual errors and fix them. To do so, we believe that tools must be able to convey their results to the user efficiently and with sufficient information for the user to correctly and quickly arrive at a conclusion.

In our previous work (Section 2), we developed a code visualization tool that is designed to efficiently explain the often long and complicated errors reported by static analysis tools. While this reduces the users' effort to understand error reports, we discovered that users face a more fundamental difficulty in understanding how false warnings may arise from specific sources of imprecision in static analysis algorithms. Training is an impractical solution to this problem; even static analysis experts such as ourselves find that it can often take some time to study a particular static analysis tool to truly appreciate and internalize all the intricacies in the underlying algorithm.

Instead, we believe that we can provide systematic guidance to the user in the form of *triaging checklists* (Section 3). Checklists are very practical devices to guide users in triaging, since users simply follow the instructions on the checklist to answer each question and to determine the conclusions. Checklists can also be very specific, since they can be designed by tool developers to point out known sources of imprecision in their tools and instruct users how to look for them. To be most effective, we want checklists to be customized to individual warnings such that users will only need to answer exactly the minimum number of questions to triage each warning. In this paper, we describe our ongoing efforts to explore how sources of imprecision may be traced through various static analysis algorithms, and how to con-

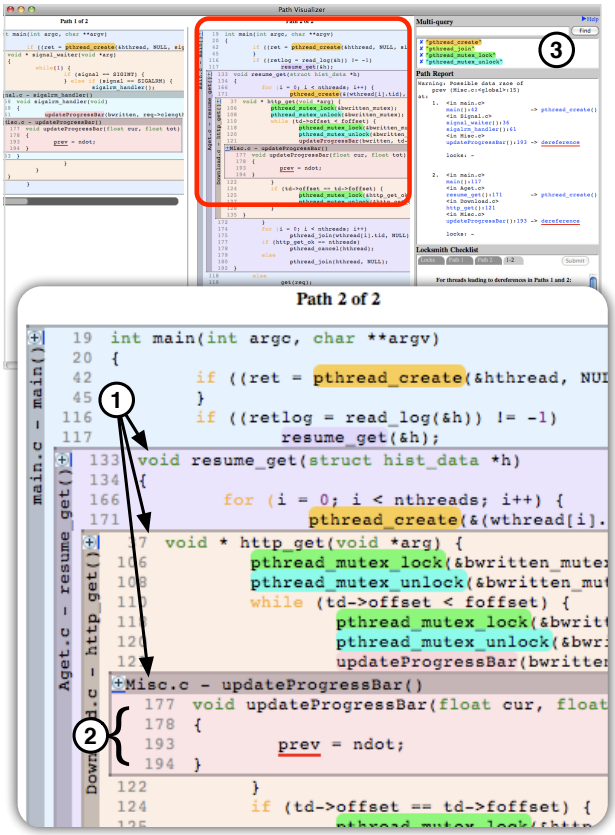


Figure 1: Path Projection (top) and a close-up which shows (1) function call inlining, (2) code folding, and (3) multiple keyword searches.

struct clear, easy-to-understand checklists with this information.

2. Previous Work: Visualizing Program Paths with *Path Projection*

In previous work, we developed *Path Projection*, a general user interface toolkit for visualizing and navigating *program paths* [Khoo et al. 2008a]. Program paths are lists of program statements, and are used by many static analysis tools to report potential errors. For example, CQual [Greenfieldboyce and Foster 2004] and MrSpidey [Flanagan et al. 1996] report paths corresponding to data flow; BLAST [Beyer et al. 2004] and SDV [Ball and Rajamani 2002] provide counterexample traces; Code Sonar [GrammaTech, Inc. 2007] provides a failing path with pre- and post-conditions; and Fortify SCA [Fortify Software Inc. 2007] provides control flow paths that could induce a failure.

However, manually tracing program paths to understand a warning is a tedious task: the user has to jump between different functions in different files and sift through many lines of source code, while trying to work out how relevant statements along the path may or may not lead to the error. Most source code editors are inefficient for this task as they are designed to view one section of a file at a time, whereas a program path may span multiple functions across several files. Many editors provide hyperlinks for users to quickly navigate between different sections of the path, but even with hyperlinks, there is still a significant cognitive burden on the user to remember fragments of code for each path section, or alternatively, to manually arrange multiple editor windows to see all of the path at once.

With these issues in mind, we designed Path Projection to visualize program paths in a way that helps users see the entire path at once. Path Projection uses two main techniques—*function call inlining* and *code folding*—to "project" the source code onto the error path. An example is shown in Figure 1. In the shown path, `main` calls `resume_get`, and the body of `resume_get` is inlined directly below the call site (1). Then `resume_get` calls `http_get` (via `pthread_create`), so the latter's body is inlined, and so on. Inlining ensures the code is visually arranged in path order, which removes the need to jump around the program to trace a path. Code folding is used to hide away irrelevant code, indicated by discontinuous line numbers, so that the user is initially shown as much of the path as will fit in one screen (2). We show only lines that are implicated in the error report, and the function names or conditional guards of enclosing lexical blocks (including matching braces). The highlighted keywords `pthread_create`, `pthread_mutex_lock`, etc. would normally be folded, but are here revealed through multiple keyword searches (3).

We evaluated Path Projection's utility with a controlled experiment in which users triaged reports produced by Locksmith [Pratikakis et al. 2006], a static data race detection tool for C. Locksmith reports data races by listing the call stacks that access the shared variable for each conflicting thread, so, we use Path Projection to visualize the call stacks side-by-side. We measured users' completion time and accuracy in triaging Locksmith reports, comparing Path Projection to a "standard" viewer that we designed to include the textual error report along with commonly used IDE features. We found that Path Projection improved the time it takes to triage a bug by roughly one minute, an 18% improvement, and that accuracy remained the same. Moreover, participants reported they preferred Path Projection over the standard viewer. Users spent little time looking at the error report in the Path Projection interface, which suggests that Path Projection succeeds in making paths easy to see and understand in the source code view.

3. Current Work: Checklists for Triaging Static Analysis

We find from our experiments that, although a good visualization such as Path Projection reduces the effort to triage the result of static analysis tools, many users still find the triaging task to be very difficult. To triage a reported error,

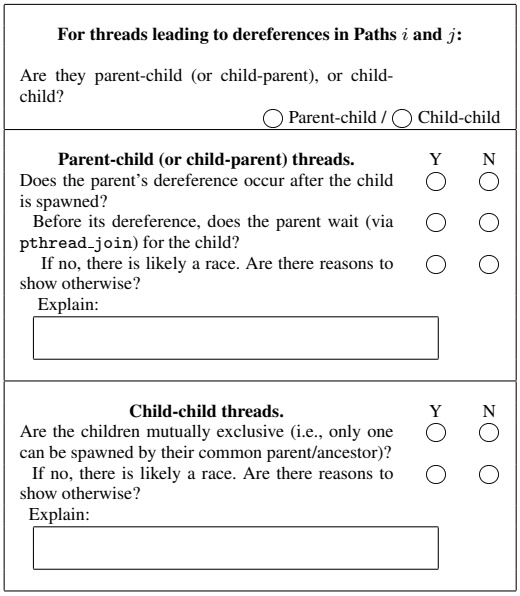


Figure 2: Checklist for triaging Locksmith reports

the user has to know enough about the analysis performed—in particular, its sources of imprecision—to determine if the error is a false positive. For example, a static analysis may be *path insensitive*, meaning it assumes that all conditional branches could be taken both ways. Thus the tool may falsely report errors on unrealizable paths. As another example, a static analysis tool may be *flow insensitive*, meaning it does not pay attention to statement ordering. Thus the tool may decide that some source data might reach a target location even if an intermediate assignment statement kills the flow of data, making this impossible at run time.

In our experience, reasoning about imprecision to detect false positives is out of reach for most users. We found that even with extensive tutorials, participants had trouble triaging the results from static analysis tools [Khoo et al. 2008b]. Their triaging procedures were usually ad hoc and inconsistent, often neglecting some sources of imprecision (and thus sometimes wrongly concluding a report to be a true bug) or assuming non-existent sources of imprecision (and therefore wasting time verifying conditions certain to hold).

3.1 Triaging checklists

We believe we can greatly improve the effectiveness of static analysis tools by providing users with checklists to guide them through the triaging process. A *triaging checklist* enumerates a series of questions that the user has to answer in order to triage a particular error. In our Path Projection study, we developed a partial checklist to help users triage error reports from Locksmith. One common source of false positives from Locksmith is its path insensitivity, so this checklist focuses on verifying the realizability of paths implicated in a data race.

A section of the checklist is shown in Figure 2. This section is shown when Locksmith reports that two threads, i and j, access a shared variable without holding a common lock, which would lead to a data race. The user has to examine the call stacks of the conflicting threads in the Locksmith report to determine if the variables may be accessed simultaneously from threads i and j.

The user first has to decide whether threads i and j are in a parent-child or other (child-child) relationship. If the user selects parent-child, the user then needs to determine if the dereference in the parent occurs after the child thread is created (otherwise there is no race) and if there are no blocking thread joins preventing the parent from dereferencing the shared variable until the child has joined (Locksmith ignores calls to pthread_join). The child-child case is analogous—the user must check whether the two children are mutually exclusive (e.g., spawned in disjoint branches of an **if** statement), which would preclude a race. For both parent-child and child-child races there is a catch-all checklist question for other reasons that could preclude the race, e.g., due to branching logic along the given path.

In our final Path Projection study (described in Section 2), we provided our users with the same checklists for both user interface conditions. Prior to that study, we ran a pilot study without checklists, and found that users took much longer to complete the given tasks. Although the two studies are not strictly comparable, we observed that users triaged error reports roughly 40% (or four minutes) faster with checklists than without them, and their conclusions were more reliable. A key reason for improved accuracy is that checklists make clear exactly what users need to look for, so they can be systematic and not miss important indicators. The reason for improved efficiency is that the checklist enumerates exactly what must be done, and no more. For example, the Locksmith checklist has no mention of verifying whether a listed lock is actually held—in this case Locksmith's algorithm is perfectly precise, so its conclusions are trustworthy. But in our earlier pilot study, many users would get distracted examining if locks were or were not held.

We think there is much promise in checklists, so we plan to study their use more clearly and systematically. First, we are working to generalize the use of checklists to other tools and other types of imprecision in static analysis. For example, Locksmith is also imprecise because it uses a flow-insensitive alias analysis, which means that while paths to dereferences in different threads may be simultaneously realizable, the dereferences may actually be to different memory locations (and thus not a race), contrary to what the alias analysis thinks.

Second, we would like to build static analyses that efficiently track sources of imprecision, and use this information to construct checklists that are specific to each reported error.

For example, for the statement **if** (x) p = q, we know p and q are aliases only if x is non-zero, but a *path-insensitive* alias analysis would conservatively ignore the conditional and simply assume, unconditionally, that p and q are aliases. If this assumption is used to generate an error report, the report will be a false positive if x is always zero. Thus, the analysis should keep track of when it takes this imprecise step. If the assumption leads to an error, a checklist item can be constructed to ask the user to check whether x may indeed be non-zero. This basic idea is similar to client-driven pointer analysis [Guyer and Lin 2003], which attempts to selectively remedy the imprecision of its pointer analysis based on feedback from subsequent client static analyses. While useful, automated remedies are not always possible, nor can they always be identified cheaply or reliably. Checklists take advantage of the human's expertise and computational ability to verify well-defined problems that may have no satisfactory automated solution.

Finally, we plan to measure the efficacy of checklists and checklist assistants through controlled user studies for bug triage, of the flavor of the one used to evaluate Locksmith's existing checklist [Khoo et al. 2008b]. As we gain further insight and experience in developing checklists, we will move to automate the generation of tool-specific checklists as well, drawing on the basic theory of abstract interpretation (which expresses the ways in which an analysis domain is conservative) [Cousot and Cousot 1977]. We will also consider means to allow users to construct their own checklists that can take advantage of accumulated analysis information.

3.2 Checklist assistants

We are also investigating the use of *checklist assistants*, which are simple analyses to help answer specific questions in triaging checklists. Unlike the core analyses of tools, these simple analyses need not be sound; they will simply point the user in the right direction, ultimately relying on his/her judgment. For example, our Path Projection interface contains a rudimentary assistant in the form of a multi-keyword search that highlights and reveals matching text even if they had been hidden by code folding. Consider the Locksmith checklist again: the user is asked in one case to check if a parent joins a child thread before an access to a common shared variable; if so, there is no race, since at that point the child thread has exited. To assist in this task, the user may enter " pthread_join " into the multi-keyword search to highlight all matching occurrences of that text in the displayed path. The user can then visually scan for a matching occurrence of pthread_join between the accesses in the parent and child threads. If there is a such occurrence, the user can quickly determine that there is no race. A more sophisticated assistant may recursively search all functions called between the parent and child threads for occurrences of pthread_join , further simplifying the user's effort to answer the checklist.

Rather than "baking in" these sorts of analyses into a given tool, we are looking into providing a generic library of checklist assistants that can be reused across different types of static analysis. These may be developed in the style of ASTLog (later, PREFast) [Crew 1997], for simple syntactic queries, or a more general data flow analysis framework parameterizable by the lattice, transfer functions, and so on [Chambers et al. 1996; Duesterwald et al. 1997; Dwyer and Clarke 1996; Hall et al. 1993]. Ideally, these checklist assistants should have access to the internal results of the tools' core analyses (e.g., the control flow graph, points-to graph, etc.); however, we are also exploring the possibility of working with just the information available from the tools' error reports, to make checklist assistants applicable to any tool. We also imagine allowing users to indicate that a heuristic analysis be used automatically, once it becomes sufficiently trusted.

4. Related Work

Checklists have attained widespread adoption in a variety of fields [Hales and Pronovost 2006], including emergency room triage [Berman et al. 1989], aviation [Degani and Wiener 1990], and ergonomics [Brodie and Wells 1997]. In software engineering, checklists play an important role in software inspection tasks. Anderson et al. [2003] demonstrate how CodeSurfer can be used to answer questions in NASA's Code Inspection Checklist. Ayewah and Pugh [2009] developed a checklist for Findbugs to help users rate the severity of reported warnings. The successful adoption of checklists in many fields gives us confidence that we can greatly improve the usability of static analysis tools by giving users checklists.

Several tools exist to query code facts, such as Ciao [Chen et al. 1995], JQuery [Janzen and Volder 2003], and SemmleCode [Semmle Limited]. Lencevicius et al. [2003] propose using querying for interactive debugging, and Ko and Myers [2008] built a debugger called Whyline that allows programmers to ask "why" and "why not" questions about a program trace. Partiqle lets users express relational queries over program traces [Goldsmith et al. 2005]. In contrast to these approaches, our checklist assistants are specifically intended to tackle imprecision in static analysis tools. Martin et al. [2005] propose PQL (Program Query Language), a simple language for writing static analyses that implemented via compilation to datalog programs that work with bddbddb [Whaley and Lam 2004]. We may be able to use ideas from PQL in developing our checklist assistants, but we hope to provide a more flexible system that employs a range of static analysis techniques rather than one approach.

5. Conclusion

In this paper, we propose to use *triaging checklists* as one key tool to make static analysis tools easier to use. While a good visualization is useful to explain a warning efficiently, a good triaging checklist provides users with clear and complete instructions to decide if a warning is truly an error or

false warning. We are investigating how checklists can be applied to a variety of static analyses, as well as how to trace sources of imprecision in static analysis to construct checklists that are highly tool- and error-specific. Additionally, we are also exploring *checklist assistants*, which are lightweight analyses designed to help users answer checklist questions.

Acknowledgments

This research was supported in part by National Science Foundation grants CCF-0541036 and CCF-0915978.

References

Paul Anderson, Thomas Reps, Tim Teitelbaum, and Mark Zarins. Tool support for fine-grained software inspection. *IEEE Software*, 20(4):42–50, July/August 2003.

Nat Ayewah, Hovemeyer David, J.D. Morgenthaler, J. Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, September 2008.

Nathaniel Ayewah and William Pugh. A report on a survey and study of static analysis users. In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-051-7. doi: http://doi.acm.org/10.1145/1390817.1390819.

Nathaniel Ayewah and William Pugh. Using checklists to review static analysis warnings. In *DEFECTS '09: Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, pages 11–15, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-654-0. doi: http://doi.acm.org/10.1145/1555860.1555864.

Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2007.

Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, January 2002.

D. A. Berman, S. T. Coleridge, and T. A. McMurry. Computerized algorithm-directed triage in the emergency department. *Annals of Emergency Medicine*, 18(2), February 1989.

Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Blast query language for software verification. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 2–18, Verona, Italy, August 2004. Springer-Verlag.

David Brodie and Richard Wells. An evaluation of the utility of three ergonomics checklists for predicting health outcomes in a car manufacturing environment. In *Proc. of the 29th Annual Conference of the Human Factors Association of Canada*, 1997.

Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for Intra- and Interprocedural Dataflow Analysis. Technical Report 96-11-02, Department of Computer Science and Engineering, University of Washington, November 1996.

Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proceedings of the International Conference on Software Maintenance*, pages 66–75, 1995.

Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

Roger F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

CSE. Microsoft center for software excellence. `http://www.microsoft.com/windows/cse/default.mspx`.

Manuvir Das. Formal specifications on industrial-strength code: From myth to reality. In *Proceedings of the 18th International Conference on Computer Aided Verification*, page 1, August 2006.

Asaf Degani and Earl L. Wiener. Human factors of flight-deck checklists: The normal checklist, 1990. NASA Contractor Report 177549.

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

Matthew B. Dwyer and Lori A. Clarke. A Flexible Architecture for Building Data Flow Analyzers. In *Proceedings of the 18th International Conference on Software Engineering*, pages 554–564, Berlin, Germany, March 1996.

Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, Bolton Landing, New York, October 2003.

Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Fourth symposium on Operating System Design and Implementation*, San Diego, California, October 2000.

Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching Bugs in the Web of Program Invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, Philadelphia, Pennsylvania, May 1996.

Fortify Software Inc. Fortify Source Code Analysis, 2007. `http://www.fortifysoftware.com/products/sca/`.

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.

Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 385–402, New York,

NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094841.

GrammaTech, Inc. CodeSonar, 2007. `http://www.grammatech.com/products/codesonar/overview.html`.

David Greenfieldboyce and Jeffrey S. Foster. Visualizing Type Qualifier Inference with Eclipse. In *Workshop on Eclipse Technology eXchange*, Vancouver, British Columbia, Canada, October 2004.

Samuel Z. Guyer and Calvin Lin. Client-Driven Pointer Analysis. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 214–236, San Diego, CA, USA, June 2003. Springer-Verlag.

B. M. Hales and P. J. Pronovost. The checklist – a tool for error management and performance improvement. *Journal of Critical Care*, 21(3), 2006.

Mary Hall, John M. Mellor-Crummey, Alan Carle, and René Rodrıguez. FIAT: A Framework for Interprocedural Analysis and Transformation. In *Proceedings of the 6th Annual Workshop on Parallel Languages and Compilers*, August 1993.

David Hovemeyer and William Pugh. Finding bugs is easy. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA Companion*, pages 132–136. ACM, 2004.

Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM. ISBN 1-58113-660-9. doi: http://doi.acm.org/10.1145/643603.643622.

Ciera Christopher Jaspan, I-Chin Chen, and Anoop Sharma. Understanding the Value of Program Analysis Tools. In *OOPSLA'07 Practitioner Reports*, 2007.

Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. Path projection for user-centered static analysis tools. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, November 2008a.

Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. Path Projection for User-Centered Static Analysis Tools. Technical Report CS-TR-4919, Department of Computer Science, University of Maryland, College Park, August 2008b.

Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 301–310, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368130.

Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engg.*, 10(1):39–74, 2003. ISSN 0928-8910. doi: http://dx.doi.org/10.1023/A:1021816917888.

Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094840.

Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI '06*, pages 320–331, 2006.

Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings. In *International Conference on Software Engineering*, 2008. To appear.

Semmle Limited. Semmle — Query Technologies. `http://semmle.com`.

Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, Washington, D.C., June 2004.

Climbing the Plateau

Getting from Study Design to Data that Means Something

Christine A. Halverson
Social Computing Group
IBM T.J. Watson Research Center
krys@us.ibm.com

Jeffrey Carver
Department of Computer Science
University of Alabama
carver@cs.ua.edu

Abstract

Evaluating the usability of a programming language or tool requires a number of pieces to fall into place. We raise issues along the path from study design to implementation and analysis drawn from the experience of running several studies concerned with a new parallel programming language – X10. We summarize several analyses that can be drawn from different aspects of the same data.

Categories and Subject Descriptors D.2.5 [**Software Engineering**]: Testing and Debugging; H.1.2 **[Information Systems]**: User/Machine Systems—Human Factors

General Terms Measurement, Experimentation, Human Factors, Languages.

Keywords parallel programming; programmer productivity; study design; data collection; integrated methodology; analysis of programmer behavior.

1. Introduction

While programming massively parallel computer systems has been going on for at least two decades, few details are known about its practice. The information that is available is a combination of anecdotal evidence and a few studies of parallel programming students. Neither of these sources is bad, just limited.

In contrast, scientific computing "in the wild" using high performance computing systems covers a wide range of problems within a variety of organizational structures: industry, academic, public and clandestine government.

While scientific software has a lot of issues in common with "traditional" software, it also has many unique issues due to its use of massively parallel machines and the fact that the primary job and skill set of many of the programmers is science (or other work) not computer science.

This work was motivated by our participation in the DARPA High Productivity Computing Systems (HPCS) program [10]. The program is in the last of three phases of an eight-year effort aimed at developing peta-scale machines that significantly improve the productivity of its users, i.e. scientists, programmers, data managers and system administrators.

As a member of the PERCS Productivity team at IBM [15] the first author is working to demonstrate an increase in programmer productivity with IBM's new machine and tools from a 2002 baseline. While definitions of productivity are contentious, two things are clear. The ability of programmers to use the machine and its software stack (including programming language and assorted tools) – that is, the *usability* of all of these things – is important. We also must compare usability between 2002 and what will be available around 2010.

We need to understand how parallel programming is occurring and how its practice is impacted by differences in machines, tools and languages. In this paper we outline the path from study design through study implementation and analysis for just one of the many cases we need to evaluate. In particular we focus on how choices made early in study design impact the kinds of analysis that can be done.

2. Related Work

There is literature from a number of disciplines that influenced our approach to study design. Previous efforts to understand programmer behavior have encompassed three main methodologies: self-report via survey or interview, automated measurement of machine-human

interaction during programming, and empirical studies – whether in the laboratory or in the field. (For example. See Perry et al. [16,17] using field based approaches; Hofer and Tichy [9] for a review of empirical approaches in the last decade; and Basili [2] for lab based empirical approaches.)

There have been studies in HPCS since the early 80's. Some studies centered on specific machines, while others focused on programming languages or parallel environments [3]. The manner of data collection and the available subject pool have largely constrained the method and measurements. As most studies occurred in universities, the subjects were students, usually in their first parallel programming class [7]. The programming task was generally a class assignment, such as Sharks and Fishes [8]. Course requirements also dictated the programming language. Tools and machines were determined respectively by personal preference and availability. As in earlier studies on sequential programming, data collection is one of three types: manual, automated or *hybrid*, a combination of the two. Hochstein et al [8] present a *hybrid* method that combined automated data gathering with manual data provided by programmer self report.

We have used a different combination for a hybrid–the integrated methodology [4]. This method combines automatic data collection (SUMS, [13,14]) with concurrent manual observations. These observations are taken by a trained independent observer, eliminating two of the problems noted with self-report data, namely the self-interruption requiring context switching by the programmer and the potential bias expressed in the content of the programmer's self reports.

Methodological difference aside, one of the pervasive problems in empirical studies of programmers is the ecological validity of the study. Ecological validity refers to how close the method, setting, and materials mimic the real world situation. Controlled experiments by nature limit the variance in what is being studied. The recognition of the necessity for ecological validity in programmer studies is not new. Schneiderman and Carroll [18] focused on the need for studies of professional programmers in their native environments, and Perry et al picked up this call again in the mid 1990s [16,17]. In both cases researchers were figuring out ways to study programmers in the wild, which resulted in qualitative and quantitative data.

3. Study Design: Building the Trail

Designing the study is laying out the trail to be traversed. The point is to consider all the problems and issues in advance so that during the study the right type of high quality data can be collected.

Our overall goals were to provide a baseline (circa 2002) for single programmer behavior and show progress in productivity improvement delivered by the X10 programming language [21] and the X10 development toolkit (X10DT). We made the following choices in study design.

SUBJECTS: Prior studies were mostly done with students who were taking their first parallel programming classes [7,8]. For PERCS, we recruited students with programming experience and taught them one of 3 parallel languages [4,5]. (See also [19] re designing studies for HPCS). However, feedback from the HPCS program noted that the results of these studies didn't necessarily apply to actual experience levels.

In response, we chose to look at two levels of experience. Prior trouble recruiting HPC professionals led us to define experienced subjects as those having 10 or more years of experience in parallel programming without constraining where they worked currently. To avoid some student issues novices were considered to be those that have had at least one parallel programming class and at least 3 years of experience programming. In this way we hoped to bracket the problem space and avoid effects caused by having just learned parallel programming. Subjects were also required to be familiar with the most commonly used editors and tools. For the baseline these tools were vim, emacs, and gnu debugging and TotalView. For the newer condition the tools were Java and Eclipse (as the X10DT is built into Eclipse as a plug in).

BASIC DESIGN: To cover both goals and experience levels we ran 4 groups of 10 subjects in a standard 2 x 2 design. This would give us a group of novices and experts for each language condition: MPI and X10. We hoped to target the majority of experienced subjects through our association with National Energy Research Scientific Computing Center (NERSC) and Lawrence Berkeley Laboratories (LBL) while picking up our novices either through postdocs at the same institutions or advanced graduate students associated with them.

PROBLEM: We needed to use a programming "problem" that would be realistic but actually solvable in the time available. In our previous study we used a Synthetic Scalable Compact Application (SSCA). (Developed for HPCS [1]). SSCA1 presents a pattern matching problem, such as gene sequencing. Although it's a problem from genetics, it does not require deep domain knowledge to solve.

From prior experience we knew that the problem was solvable by most within a 2-day time frame. We provide subjects with working serial code and ask them to parallelize a portion. Making it parallel can be done in two ways: a more difficult wave-front algorithm or a straightforward embarrassingly parallel solution. Again, to

reduce time, we attempted to focus the subjects on the easier approach.

ENVIRONMENT: There are multiple issues regarding hardware and software setup. We must duplicate 2002 circumstances for the baseline while also accommodating the newer tools. Finally both setups must allow data collection.

For the baseline, our model was an existing machine: NERSC's IBM SP RS/6000 Power3—Seaborg. It was brought online in 2001 and still had the (updated) software and tools that fit our needs. Our pilot subject used Seaborg until it was decommissioned in January 2008. At that time we switched our setup to Bassi—an IBM p575 Power5 system. While the computational capabilities are somewhat different, we were able to provide the same software stack (operating system, editors, mpi library and compile commands) as we had on Seaborg to approximate 2002 conditions. All study subjects all used Bassi.

Most programmers in 2002 programmed directly on the interactive portion of a machine's nodes via secure shell connection from a desktop. However, others developed code on a local machine and then uploaded and ran the code. We wanted to accommodate both styles. For our purposes, laptops are as powerful as a desktop and more portable. We set up five identical ThinkPad T61p laptops for this study. The table below shows the machine configuration and data collection software used.

	Laptop	Bassi
OS	Fedora Core 6 Linux	IBM POE, AIX
Editors	Vim, emacs	Vim, emacs
Shell	Bash	Bash
Languages	Fortran 77 & 90, C	Fortran 77 & 90, C
Message Passing	MPI	MPI
Web Browser	Firefox (NERSC and language sources)	none
Automated Data Collection	Hackystat Slogger (web) pFig (Eclipse) Istanbul (screen cap)	Hackystat

DATA COLLECTION: We strove for unobtrusive and automated data collection. The first goal is to minimize any effect on programmer behavior while the second goal is to minimize experimenter effort. This means using the computer to automatically collect interaction data. However programmers engage in activities besides typing on a keyboard, so we also needed to collect behavioral data.

For this project we used the Hackystat v7 framework for automated data collection. Hackystat [6] is an open source framework for automated collection and analysis of software engineering process and production metrics. Hackystat users attach software "sensors" to their development tools, which unobtrusively collect and send raw data about development to a Hackystat web server for display and analysis. On both the laptops and Bassi we used sensors attached to the command line, the bash shell, and the vim and emacs editors. On the laptop we also used another piece of software – slogger [20] – to record web browser activity. (Note: subjects were restricted to references for language and machine operation. No googling was allowed.)

The addition of X10 and the X10DT required using Eclipse. The Hackystat Eclipse sensor, like its other editor sensors, captures whether or not a file is being edited and whether the file size is changing. With vim and emacs this information allows us to infer that the programmer has edited and saved the file. Then the user needs to go to the command line to build and run the file. However, as Eclipse is configured to automatically save and build we cannot necessarily make the same inference of programmer intent. Instead we used a separate data gathering Eclipse plug-in written in Java, called PFIG (Programmer Flow Information Gatherer [12]). PFIG instruments Eclipse to monitor navigation activity in the IDE. It collects data such as the location of the text cursor within files, usage of the package explorer, the locations and contents of variables and methods within classes, program launches, and changes to source code. This was more data than we needed, but we did feel that seeing the patterns of use exposed within Eclipse would provide us valuable information about the use of X10 and the X10DT.

Finally, observations were used to fill in gaps where no automated data was collected and to infer programmer intent in some cases. In our previous study [4,5] of 27 subjects, 3 observers could not cover the subjects continuously. In that case, we chose a sampling method where one observer was assigned to a cohort of 9 (one language condition). Observations were taken for 5 minutes on one subject, after which the observer moved to the next subject in the cohort. Unfortunately this meant that each subject was only observed for a 5 minute period out of 45 minutes, which was not always sufficient to cover gaps in the automated data collection. For this study we knew we would have similar limitations on observers. We designed the study so that at any one time each observer only had 3-4 subjects to observe. Each subject was observed at least once per minute insuring coverage with automated data.

We knew there would still be situations where we would have coverage difficulties. Video data would provide similar detail, but while easy to record, it requires fairly obtrusive equipment and faces a scaling problem – 10 subjects require 10 cameras. Our solution was to move to screen capture. While it requires significant time to analyze, this type of data does have the benefit of being relatively unobtrusive and automatically collected. We used an open source project called Istanbul [11] for screen

capture. Requiring some additional user setup work it is more intrusive than Hackystat, but the payoff seemed worth it.

Overall we had three kinds of data collected on three different time scales. The Hackystat sensors poll on a 5 -15 second interval. The human observations are within a minute resolution and for the X10 portion the PFIG log was time stamped at millisecond resolution.

4. Running the Study: Climbing Up

Before the study could be run, two additional things needed to happen. One was to get human subjects approval from the appropriate institutional review board (IRB), and the other is to recruit subjects. IRB approval varies with institution and each situation has its own challenges so we do not deal with it here—other than the caveat to allow plenty of time to deal with this step.

The other issue is actually recruiting, qualifying and scheduling subjects for the study. We changed several aspects of the study to facilitate recruiting We had shortened the study from nearly a week to two days. We also paid subjects and got NERSC management to allow people to participate during work time. We still had trouble getting experienced subjects. Only 4 qualified and participated.

We were more successful with the students at Rice where we recruited subjects for the two novice language conditions. We qualified 9 subjects for the X10 condition and 7 for C+MPI. Due to 2 dropouts we ended up with 7 in each condition.

PROTOCOL: At both NERSC and Rice the set up was the same. The study procedure was close with a few variations. The protocol consisted of:

- Physical setup of machines
- Welcome subjects and obtain consent
- (X10 condition – 1 day language tutorial)
- Cover basics of machine and problem organization
- Introduction to problem
- Coding
- Daily breaks for lunch and snack
- Complete problem
- Take post survey

All subjects worked in the same room, facilitating the process for the experimenters.

Subjects were provided with electronic and hard copies of the problem statement and associated materials about the details of the machine set up. They also had pen and paper for note-taking. Subjects were cautioned not to confer about the problem, but were able to share information about working on the laptop or Bassi's environment.

One distinction between NERSC and Rice is that at Rice the problem was verbally presented in addition to the written presentation. At Rice we added a series of test cases that explored edge conditions to ensure the problem was solved. Passing all test cases and having the parallel code running on 8 processors faster than the serial version defined task completion. Some finished the task within a day and moved onto variations while others needed two days. Some subjects never completed the task. At the end of each day data was collected and backed up on a separate hard drive.

5. Analysis: The Boulder at the Top

All of the work thus far only pays off if you can analyze the data for the results you intended. Analysis happens for many reasons and on many levels. For illustration we summarize two analysis types: quantitative and qualitative.

Before beginning analysis we first unified and time aligned the sensor traces. For the Rice data we merged the Hackystat traces from the command line, shell and editors, along with web activity, Eclipse activity, and the per-minute human observation logs. The millisecond resolution, time ordered traces, were output to comma separated (csv) files for subsequent analysis.

Pre and post study survey data provided the information about the subject demographics and preparation as well as their perception of the study and the X10 language. These responses were summarized and used to provide context for the other analyses.

COMPLETION: Interleaving the time ordered data makes some analyses quite easy. Unambiguous completion criteria using the test scripts at Rice meant we were able to determine the time at which the problem was first solved successfully on each subject's laptop and on the 8-processor run on Bassi. This time, minus the sum of subject time away from the laptop, defined *time to completion* and served as the quantitative basis for comparing productivity between the two language conditions.

PATTERNS: Understanding programmer practice is important for understanding programmer use and perception of programming languages and tools. This requires a more qualitative approach. We began analyzing the event traces to determine the proportion of time spent in particular aspects of development (analysis, coding, debugging, etc.).

For the NERSC data we had only Hackystat and observation data. Analysis was done by hand by progressively segmenting the record into appropriate categories, based on what was in the data and our knowledge of programmer behavior and software engineering. For example a series of commands such as

edit, make, edit iterated several times suggest cleaning up code (such as compiler errors). The sequence *edit, make, run, edit* suggests debugging. These sorts of manually derived categories formed the basis for our analysis of the Rice data.

VISUALIZATION: An important goal of examining the log of developer activities was to infer whether developers were following any type of consistent workflow. There are two reasons why such an investigation is interesting. First, many of the developers who write code for HPC machines are not trained software engineers or computer scientists, as was the case with some of our subject sample. Second, there are some additional activities required in the development of HPC code that are not present in the development of more traditional types of code, e.g. parallelization of code, debugging code running on multiple processors, and tuning the code/algorithm to increase performance on the parallel machine. Therefore, we expected to see some development patterns emerge from the visual analysis of the data.

In order to identify the workflow that a particular developer followed, we first needed to isolate the programming events. Then we could analyze the order and frequency of these events. The programming events we were interested in were: *Writing New Code*, *Debugging Serial Code*, *Debugging Parallel Code*, and *Tuning Code Performance*.

To conduct this analysis we used an IBM proprietary tool, Zinsight, to view the various types of data described earlier in one screen. To identify which programming event occurred, we examined the commands typed at the command line or executed through the Eclipse interface. Specifically, we were interested in identifying when the subjects edited code, compiled code and executed code. When the subjects executed code, we were also able to capture the number of processors used (ranging from one to eight). While we could do this by reading the csv file in Excel, Zinsight made patterns easier to discern.

This analysis was conducted under the assumption that the sequence of commands would suggest the programming event that occurred. We began our analysis with the following hypothesized relationships:

- Edit → Make = *Writing New Code*
- Run (1 processor) → Edit → Make → Run (1 processor) = *Debugging Serial Code*
- Run (n processors) → Edit → Make → Run (n processors) = *Debugging Parallel Code*
- Run (n processors) → Make → Run (m processors) = *Tuning Code Performance*

As many qualitative hypotheses go, these hypotheses did not survive contact with the actual data fully intact. For this paper, we analyzed the development log of three subjects in detail. These three subjects exhibited different workflows, some of which coincided with our hypotheses and some did not.

Subject 1

We observed two patterns with subject one. First, most of his runs were done on two processors. It is not clear whether the lack of runs on one processor indicate that serial coding was not done in isolation, i.e. he went straight to parallel coding, or something else. The pattern we observed was that after a series of runs on two processors, he began systematically working up to more processors (i.e. three, four, five, etc…). Each time he added processors, he also edited the code. We assume the edits were to correct issues that became evident as more processors were added. At some point, this subject returned to two processors and started the process over again. We infer that the pattern this subject was following was: 1) Add new functionality; 2) Debug on two processors; 3) Add processors and debug; 4) Return to Step 1 on two processors.

The second pattern we observed in subject one was regarding his edits of source code. We were interested in the longer periods of editing, rather than quick fixes. These longer periods of editing fell into two groups: 1) the subject was running on more than two processors before and after the edit (the same number both times); 2) the subject was running on more than two processors before the edit, but only two after the edit. In addition, during many of the long editing sequences there are multiple 'make' commands executed before a 'run' command is executed.

Subject 2

The patterns we observed for Subject one were not as evident for Subject 2. We did not observe the same pattern of systematically running on more processors. Furthermore, for Subject 2, when he did a long editing session, it was not always followed by a run command. Sometimes it was followed by a make command and then more editing.

Subject 3

This subject again evidenced another pattern that did not exactly match those of Subjects one and two. Subject three performed most of his runs on two processors. There are only a few runs that used four processors and they are all at the end of the development cycle. It appears that this subject was not systematic about adding new functionality and then debugging that functional on multiple processors before moving on to new functionality. We also did not observe the same editing patterns for Subject three that we observed for Subject one.

This analysis is very preliminary and based only on three of the subjects who participated in the study. The next step of this work is to go further with this analysis to see whether these patterns are common across subjects or whether each subjects used their own style. The result of

this analysis will allow us to identify HPC development patterns.

6. Discussion & Conclusion

The analyses we present here are only a few of those done; many more are possible. At Rice, only one of the MPI subjects finished the task, while five X10 subjects completed. Median time to completion for X10 subjects was a little more than half the time it took the MPI subject. In contrast, at NERSC all of the subjects using MPI finished, most of them in a day. Given those facts what do we really know from this study?

In part our knowledge is incomplete because the study itself is incomplete. We still need to run more experienced subjects using both languages. With each study we run we learn more about what it takes for a successful study. We cut the length of the study in order to appeal to more experienced (and time crunched) subjects. However, even with that and the backing of NERSC management we were still unable to recruit the number we needed.

Based on our earlier experience we added specific ending criteria and test cases in order to ensure that we knew when the problem had been successfully completed. We also changed how we observed programmer behavior to ensure coverage when programmers were not interacting with the computer. Both of these changes were successful, providing us with more accurate data that we can now use confidently for more in depth analyses exploring programmer behavior on the way to completion.

There are things we still need to know. Students had trouble with MPI, even though they had parallel programming experience. This fits what we know from scientific computing where a programmer's first job is not computer science. While some of our subjects were engineers, some were also computer scientists, suggesting that other issues are involved. We need to figure out what those issues are.

In short, doing studies like this are difficult but rewarding. The more we do them, the closer we get to understanding what elements are important to help evaluate new programming languages and tools. Our hope is that the more we understand about the individual variability of programmers, as well as the similarities between approaches to parallel programming, then we will be able to evaluate and design tools to accommodate those issues.

Acknowledgments

We thank all our subjects and the others working with us on the PERCS project: Rachel Bellamy, Jonathan Brezin, Catalina Danis, Peter Malkin, John Richards, Cal Swart and John Thomas. Thanks also to Wim de Pauw who developed Zinsight and made it possible for us to look at our data visually. This work was supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] Bader, D. A., Madduri, K., Gilbert, J. R., Shah, V., Kepner, J., Meuse, T., and Krishnamurthy, A. http://www.ctwatch.org/quarterly/articles/2006/11/designing-scalable-synthetic-compact-applications-for-benchmarking-high-productivity-computing-systems/ Retrieved 8/31/09

[2] Basili, V. http://www.cs.umd.edu/~basili/papers.html. Retrieved 8/31/09

[3] J. C. Browne, T. Lee, and J. Werth, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment," *IEEE Transactions on Software Engineering*, **16**(2), 1990, pp. 111-120.

[4] Danis, C. and Halverson, C. The Value Derived from the Observational Component in an Integrated Methodology for the Study of HPC Programmer Productivity. P-PHEC 2006, Austin TX. Pp 11-21.

[5] Ebcioglu, K, Sarkar, V., El-Ghazawi, T. and Urbanic, J. An Experiment in Measuring the Productivity of Three Parallel Programming Languages P-PHEC 2006, Austin TX. Pp 30-36

[6] HackyStat (www.hackystat.org/hackyDevSite/home.do). Retrieved 8/31/09

[7] Hochstein, L., Carver, J. Shull, F. Asgaril, S., Basili, V., Hollingsworth, J. and Zelkowitz, M. (Nov., 2005).A case study of novice parallel programmers. In *Proceedings of the 2005 ACM/IEEE SC/05 Conference*. IEEE, 2005.

[8] Hochstein, L ., Basili, V., Zelkowitz, M., Hollingsworth, J. and Carver.J. (September 2005) Combining self-reported and automatic data to improve effort measurement. ESEC/FSE 2005.

[9] Hofer, A. and Tichy, W.F. (2006) *Status of Empirical Research in Software Engineering.* http://wwwipd.ira.uka.de/~exp/otherwork/StatusEmpiricalResearch2006.pdf

[10] HPCS. High Productivity Computing Systems. http://www.highproductivity.org Retrieved 8/31/09

[11] Istanbul. http://live.gnome.org/Istanbul Retrieved 8/31/09

[12] Lawrance, J. (2009) Unpublished Dissertation and personal communication.

[13] Nystrom, N.A., Urbanic, J., and Savinell, C. Understanding Productivity Through Non-intrusive Instrumentation and Statistical Learning. P-PHEC 2005, San Francisco.

[14] Nystrom, N., Weisser, D. and Urbanic, J. The SUMS Methodology for Understanding Productivity: Validation Through a Case Study Applying X10, UPC, and MPI to SSCA#. P-PHEC 2006, Austin TX. Pp 37-45

[15] PERCS. Productive Easy-to-use Computing Systems. http://www.research.ibm.com/hptools Retrieved 8/31/09

[16] Perry, D.E., Staudenmayer, N.A., Votta, L.G. (1994) *People, Organizations and Process Improvement.* In IEEE Software, July. Pp 36-45

[17] Perry, D.E., Staudenmayer, N.A., Votta, L.G. (1995) *Understanding and Improving Time Usage in Software Development.* In Process Centered Environments. Fuggetta and Wolf, Eds. John Wiley and Sons Ltd.

[18] Schneiderman, B. and Carroll, J. (1988) Ecological studies of professional programmers. Communications of the ACM. 31(11) ACM.

[19] Shull, F, Carver, J., Hochstein, L. Basili, V. (2005) *Empirical study design in the area of High-Performance Computing.* 4th International Symposium on Empirical Software Engineering (ISESE '05). November 2005.

[20] Slogger : https://addons.mozilla.org/en-US/firefox/addon/143 Retrieved 8/31/09.

[21] X10 programming language. www.research.ibm.com/x10/; http://en.wikipedia.org/wiki/X10_(programming_language) ; http://x10.codehaus.org

Language and Library API Design for Usability of Ruby

Usability over Simplicity

Akira Tanaka

National Institute of Advanced Industrial Science and Technology (AIST)

akr@fsij.org

Abstract

The Ruby programming language is designed for easy use. The usability is an important feature since the productivity of programmers depends on it. This paper describes that the design method obtained through the experiences of developing Ruby. The design method can be used to make other languages and libraries easy to use.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Object-oriented languages

General Terms Programming Language, Library, Usability

Keywords Syntax, Library, API, Usability, Simplicity,

1. Introduction

There is no formal way to establish easy-to-use programming languages and library APIs. But the easiness is an important factor for programmers' productivity.

Ruby is the object oriented programming language created by Yukihiro Matsumoto. It is intended to be easy to use. I, Akira Tanaka, feels it has good usability. However it is not simple. It has complex behavior. It is difficult to explain why it is easy.

Basically, the design of Ruby puts importance on good properties of programming languages: succinctness, good naming convention, etc[2]. However, some of Ruby's behaviors violate such good properties to improve the usability. The concrete design method for such behaviors is not well explained.

For example, these two expressions has the different meaning.

- `obj.meth + var`
- `obj.meth +var`

The difference of this example is a space between `+` and `var`. The former has a space, the latter not. In the former, `meth` is a method call without an argument and `+` is a binary operator: `obj.meth() + var`. In the later, `meth` is a method call with an argument and `+` is a unary operator: `obj.meth(+var)`. The space resolves such an ambiguity caused by the fact that the parentheses of a method call is not mandatory.

Here, the semantics depends on white spaces. Such a design is curious form a view point of computer scientists. It is not simple but complex, against tradition, and hard to understand.

But such a curious behavior does realize the usability. By understanding the reason why Ruby adopts it, we can get an insight into the usability of programming languages.

This paper constructed as follows to explain a design method for programming languages and library APIs to be good usability. It is obtained from experiences of Ruby developer (committer).

- Although Ruby is designed to be easy to use, the design method is not explained concretely (Section 2). The explanation will have following benefits.
  - We can make libraries and other languages easy to use as Ruby.
  - We can avoid usability regression when modifying Ruby.
- The design of Ruby focus usability (Section 3). We don't mind complexity of the design if it realize usability. In general, simplicity is good property but it is not the primary goal of Ruby.
- We design Ruby incrementally to improve usability (Section 4). We find flaws of Ruby and fix them. Several issues should be considered for usability:
  - How many frequency of the flaw occur?
  - What the appropriate design according to the frequency?

  - Does the fix prevents other fixes in future?
  - If the fix has an incompatibility, how it should be dealt with?
- We describes future work ((Section 5). We will explain various techniques for usability used in Ruby as design patterns (pattern language). Since the techniques are empirical and sometimes conflict, design patterns should be good format for the explanation. They will accelerate Ruby design process. We also describes possible technology to support incremental design.

2. Usability of Ruby

Ruby is designed to be easy to use programming language. The design policy, such as succinctness, is described briefly in [2, 3] for language itself and [4] for libraries. But the description is not enough to put it into practice. Especially the practice is difficult when usual good properties conflicts usability.

The difficulty causes several problems:

- When we want to realize the usability similar to Ruby in other languages and libraries, it is difficult to determine what behavior should be imitate. We want to ignore the behavior which doesn't contribute to the usability. But it is clear to determine.
- When we want to modify Ruby, it is difficult to consider the modification will degrade the usability or not.

These problems can be solved by understanding how the usability of Ruby is implemented.

The examples follows are explained in following sections.

- optional parenthesis for succinctness and DSL
- blocks for common usages of higher order functions
- shorter names for frequently used methods for succinctness

3. Unusual Design

In this section, we describe the design of Ruby which intend to be easy to use and violates usual language design.

In usual language design, there are several good property: consistency, simplicity, orthogonality, flexibility, succinctness, intuitiveness, DRY (Don't Repeat Yourself), good name, generalness, naturalness, meets programmers' common sense, etc. In the design policy of Ruby, they are also good properties.

However, sometimes Ruby overrides the properties by usability. I.e. the design of Ruby prefer usability over the properties when they conflict. Ruby don't need consistency including rare usage. Ruby don't need succinctness including rare usage. Ruby don't need orthogonality including rare usage. Ruby don't need simplicity including rare usage.

For example, continuation (call/cc) on dynamic typed languages endorse consistency between arguments and return value because it pass former to later. This mismatch, multiple values of arguments v.s. single value of return value, can be solved by that function call can have multiple return values as Scheme. However continuation is rarely used in Ruby, the consistency is not important.

The design of Ruby is not intended to simplify the behavior. Actually the whole behavior including rare usage is complex. Some of the complexity is intentional for usability.

In general, simplicity is a good property. It derives many things from few principles. So programmers don't need to memorize many things except the principles. Another benefit is that simplicity ease programming language research. But Ruby prefer direct usability over such benefits.

In this section, we describe several examples of Ruby's complex design for usability.

3.1 Succinctness over Simplicity

In this section, we explain the example shown in the section 1. The example shows us Ruby depends on a space in a method call. If we want to choose a simple behavior, the following can be considered.

- make the parenthesis of method call mandatory.
- even if the parenthesis is optional, define the behavior regardless of the space.

If the parenthesis is mandatory, the ambiguity of + operator doesn't occur. The + of `obj.meth()+var` is a binary operator. The + of `obj.meth(+var)` is a unary operator. Also, the syntax rules can be reduced because we don't need rules for the parenthesis omitted.

Even if the parenthesis is optional, the behavior regardless of the space simplify the information notification between the tokenizer and the parser.

We didn't choose the simple behavior for Ruby. The reason behind it is succinctness.

There are many situations which don't cause ambiguities even without the parenthesis. The method call is not ambiguous if no arguments are given, only one argument is given and it is a variable, etc. If we require the parenthesis, Ruby loses succinctness for such situations.

3.2 Intuitiveness over Simplicity

The example in the section 1 also show Ruby's design policy which prefer intuitiveness over simplicity. The intuitiveness is for average programmers. Although programmers vary, they have many shared knowledge. For example, there are common textbooks and the programmers can understand pseudo code in the textbooks. Programmers who know application domain can understand the notation used by the domain. So programmers have common intuition in a degree. The detailed reason are follows.

- DSL

  DSL, Domain Specific Language, is a language which correspond to a target domain. It can represent logic in the domain intuitively. DSLs are classified as external DSLs and internal DSLs. An external DSL is an independent programming language. An internal DSL is a library in some programming language. The library provides vocabulary for the domain.

  The parenthesis of method call have an impression of function call. The impression hides the impression of the domain. So the syntax with optional parenthesis appropriate for DSL. It expose the impression of the domain. So programmers easily sense the logic in the domain.

  For example, Ruby has a DSL to manipulate Ruby runtime. The DSL is constructed by methods to load a library, define/remove a constant, define/remove a method, etc. `require` method loads the library `foo` as follows:

  ```
  require 'foo'
  ```

  `require` method is used without parenthesis in general. This reduces the impression of function call and programmers consider this as a declaration. Since the parenthesis is noise in the domain, it increase the cost to read/write/understand the code. Therefore the syntax with optional parenthesis avoid the cost.

- Proximity

  The syntax with optional parenthesis has benefits as above. However it causes the ambiguity. Ruby uses the Gestalt law of proximity to resolve the ambiguity. The law means that near objects are perceived as grouped together. `obj.meth +var` is grouped as `obj.meth` and `+var`. Ruby parses the expression as the perception. So the semantics of the expression is similar to the perception. This reduces the cost to read/write/understand the expression.

- Utility Methods
  The class library of Ruby also prefer usability over simplicity. For example, Array class has `push` and `pop` method. `push` inserts an element at the end of the array. `pop` deletes an element at the end of the array. Since the array size is changed dynamically, programmers can use the array as a stack intuitively.

  Such utility methods tends to be increased because method addition is a major way to introduce a new feature. So the class tends to have more feature and be more complex.

These design decision means that we choose usability over simplicity in Ruby.

3.3 Usage Frequency

The frequency of usage can also be a reason to override simplicity.

For example, a method name should match the following regular expression:

```
[A-Za-z_][0-9A-Za-z_]*[!?]?
```

I.e. it start with an letter or an underscore, followed by zero or more digits, letters and underscores, optionally followed by `!` or `?`.

This syntax is not simple because the last `!` or `?`. If we choose simple syntax, we can consider a syntax without the last character like C or a syntax with various character in any position like Scheme.

This complex syntax is chosen to use the naming practice of Scheme in Ruby. Scheme uses function names which ends with `?` for predicates and `!` for destructive functions. It is just a convention in Scheme because the syntax is not special for the usage. On the other hand, Ruby's syntax is specialized for the usage. This complexity realize the usage in non-S-expression language and prevent too cryptic method names.

`!` is mainly used for destructive methods as Scheme. However Ruby uses `!` only for some of destructive methods. It is not consistent. This is also because usage frequency. Since most Ruby programs are imperative style, there are too many destructive method calls to pay attention. So Ruby uses `!` only for methods valuable to pay attention, such as there are both destructive and non-destructive method and programmers carefully choose them.

The big feature of Ruby, block, is also uses usage frequency. Ruby's block is similar to higher order function in functional languages. For example, `map` can be used as follows in Ruby, Scheme and Haskell.

```
Ruby: [1, 2, 3].map {|x| x * 2 }
Scheme: (map (lambda (x) (* x 2)) '(1 2 3))
Haskell: map (\x -> x * 2) [1, 2, 3]
```

Ruby's `map` is a method of Array class which takes a block. In above example, `{|x| x * 2 }` is a block.

Ruby's block is not an expression. The syntax of block is defined with the syntax of method call. So, a block can be described only with a method call. The block is passed to the method as a hidden argument which is separated from usual arguments. This differs from lambda expression in functional languages. Scheme and Haskell can describe lambda expression as an individual expression. It is passed to `map` function as a usual argument.

This causes following pros and cons.

pro succinct description because it don't need keywords such as lambda.

pro one can terminate the method by break statement in the block.

con a method can take only one block.

Ruby's blocks are limited from higher order functions because only one block can be given for a method. But this is not a big problem because usage frequency. Since it is rare that we need to specify two or more functions, the block's benefits surpass its problem by the limitation.

The library design also utilize the usage frequency. For example, Ruby defines `p` method which is usable anywhere. It prints arguments for debugging purpose which is easy to understand for programmers. The method name, `p`, is inconsistent with other methods because it is too short in the sense of Ruby naming convention. It is intentional because debug printings are very common. In general, too short names are incomprehensible and tends to conflict. But `p` has no such problem because almost all Ruby programmers knows it.

This kind of naming convention, assigning short names for features frequently used, are called Huffman coding which term is borrowed from data compression.[1]

Huffman coding is applied for writing and reading programs. For writing, shorter and too short names reduces number of types. However too short names, such as `p`, is can be problematic for reading. So too short names should be used only if it is sure that most programmers have no problem with reading. `p` is an example of such name as explained above. In most case, names can be shorter until single word which can be understand the meaning by programmers.

Ruby uses the frequency of usage for usability. This means Ruby focus major usage and don't focus rare usage. This "focus" is implemented in various levels of Ruby: syntax, semantics and library API.

4. Incremental Design

Ruby is designed to realize the usability using various techniques usability described in section 3. However, we cannot define the complex behavior at once.

Therefore we need incremental design for usability. The design should be refined by feedback. Since we cannot find the best design at beginning, this process is unavoidable. We must find flaws and fix them.

The "flaw" means a bad usability. The process to improve the usability is follows.

- Find flaw of usability
- Design the fix the flaw
- Deal with the incompatibilities

4.1 Find flaw of usability

At first, we must find flaw to refine the design. There are several starting point to find it.

- No feature
- Not enough feature
- Feature is available but not easy to use
- Feature is available but difficult to find it

But we don't provide all features requested in the programming language and the standard library. If the flaw causes a trouble frequently, it is an important problem. If the flaw is difficult to avoid in an application but easy to fix in the programming language and the standard library, it is appropriate to fix by them.

We can estimate the frequency by investigating the similar requests in the past. Also, existing programs can be investigated for a code to avoid the flaw. For example, when we guess a code snippet is an idiom, single method which replace the idiom will improve the usability.

Since Ruby is developed in the bazaar model, any Ruby programmer can find flaws of Ruby. Such flaws are discussed in the mailing lists. Sometimes flaws are found in discussion, so open discussion is useful.

The archive of the mailing lists is useful to investigate the requests in the past. The source code search engines, such as Google Code, is useful to investigate existing programs. We can search idioms and other candidates to improve usability in many programs.

4.2 Design the fix the flaw

In general, there are two or more ways to fix flaw. So we need to design the fix for better usability. Since incompatibilities should be avoided, method addition is a good fix in general. Section 4.3 details about dealing with incompatibilities.

When we add a method, we must define its name and behavior.

The good method name is a name which is easy to understand the behavior. However Huffman coding is applied for methods which is frequently used. So we estimate the frequently of the method.

If the method is frequently used, it should have a short name or define as an operator. Since most programmers knows operators in the language already, operators are easier to adopt. This happens even if programmers doesn't sure precious behavior of the method. They have some expectation on operators and common method names such as `A << B` appends A to B, `A[B]` extract something by B in A, etc.

However the frequency is just an estimate. It can be failure. For example, we tends to assign operators to primitives but primitiveness doesn't mean it is used frequently. If we used a too short name or an operator for a feature, we may have trouble in future. When we find another feature which should be used more frequently, it is difficult to find a name shorter than that. If an operator is used, it is very difficult to find a name easier than the operator. We will need incompatible renaming to preserve Huffman coding.

Therefore short names and operators should be used only if we are certain that the feature is used frequently. If we are not certain, a longer name should be used. It doesn't causes problems in future. We can alias it with a shorter name when we are certain. It doesn't cause incompatibility because longer names are still usable.

The method should be implemented experimentally to examine the behavior.

This examination is easy in Ruby because Ruby's classes are open. It means we can define new methods in the existing classes. For example, we can define `to_proc` method in the builtin class `Symbol` as follows:.

```
class Symbol
  def to_proc
    lambda {|obj, *args|
      obj.send(self, *args)
    }
  end
end
```

The `to_proc` method is an example which is already taken by Ruby. The method is experimented by a third party at first. It is re-implemented in Ruby later. Recent Ruby has the method by default.

The classes can be bigger because we prefer method addition. The big classes are useful to try various methods. If we add a class for new feature, we must create the instance of the class to try the feature.

The method may have two or more names because shorter names are defined later. Although this violates minimalism, Ruby doesn't intend to be minimum. Perl has a slogan TMTOWTDI (There's More Than One Way To Do It). Ruby also has similar nature.

4.3 Deal with the incompatibilities

Improving usability may break compatibility. So, we should consider language and library design without incompatibility in future improvement.

If we change a programming language and a library, it can cause incompatibilities. The incompatibilities break application programs. So they should be avoided if possible.

Various changes can be classified as follows.

- compatible changes
  - new syntax
  - new class
  - new method
  - relax method arguments
  - define undefined behavior
- incompatible changes
  - remove class
  - remove method
  - restrict method arguments
  - change return values
  - change side effects

Strictly speaking, the new methods can also conflicts because applications can add the method by open class. However they are not big problem in practice because we don't use open class extensively. We assume new methods doesn't cause incompatibility here.

Since incompatibility should be avoided, we should choose compatible changes such as method addition.

However several techniques to avoid future incompatibilities in method addition.

- Arguments should be checked strictly. We can add new features by relax the arguments in future.
- Short names and operators should be used only if we are certain to they are used frequently. This reduces a possibility that we cannot find a shorter method name for methods more frequently used in future.
- Describe undefined behavior explicitly in the manual. We can add new features by changing and defining the behavior.

If we really cannot avoid incompatibilities, we can use following practices to reduce pain for application programmers.

- Incompatibilities should be introduced when the major version number is incremented. The programmers can update the application at a time for each major version.
- Warnings should be generated before incompatibilities introduced. The warnings notify that the application doesn't work well in the next major version.

The incompatible change and its warning can be implemented at a time in Ruby. Ruby has two develop branch: stable and development. The warning is implemented to the stable branch. The incompatible change is applied to the development branch. The inconsistency between the warning and the change can be avoided in this style of development. Also, application programmers can try the development version to study the incompatibility.

In Ruby, application can use open class to implement a new method in an older Ruby which don't have the method. For example, `to_proc` method in Symbol class can be implemented for the older Ruby by the compatibility definition as follows. Note that `:foo.respond_to? :to_proc` returns true if the symbol, `:foo`, has `to_proc` method.

```
if !(:foo.respond_to? :to_proc)
  class Symbol
    def to_proc
      lambda {|obj, *args|
        obj.send(self, *args)
      }
    end
  end
end
```

So application can use new methods even in the older Ruby by defining the methods.

The compatibility definitions can be removed when the older Ruby is fade out and the application discontinue support for it. No other code need to be modified at the time.

5. Summary and Future Work

This paper explains Ruby language and library is designed for usability utilize the usage frequency. The incremental design process for the usability is also explained.

However the design principle is not popular even in Ruby community. So, sometimes third party libraries are not easy to use as Ruby.

The incremental design process is not supported well by the implementation. There are ideas for mechanism to support the process.

5.1 Usability of Ruby in Future

It is important to explain the design principle of Ruby to preserve the usability of Ruby.

There are change requests for Ruby which the main reason is simplicity and doesn't focus usability. It is possible to spoil the usability if the request is accepted.

So, it is important to popularize the usability principle. If the principle is popular, the requests which degrade the usability will be decreased.

Currently we work on "language patterns" which are design patterns for designing easy to use languages and libraries. It describes DSL, structure by white spaces, etc.

The format of design patterns is appropriate for this kind of knowledge. It's because the techniques are rules of thumb. Sometimes the techniques conflicts each other. For example, the p method is bad name but the name is supported by Huffman coding rule. This knowledge is not possible to formulate as axioms and theorems.

The explanation by the design patterns provides vocabulary to discuss usability of programming languages and libraries.

5.2 Incremental Design in Future

If we can reduce problems by incompatibilities, we can accelerate improvement of the usability of Ruby.

There are several possible mechanisms to reduce the problems.

Since Ruby is dynamic language, most warnings are generated at runtime. Some of the warnings inform the application will be broken with future Ruby. They are only useful when the application is updated, useless otherwise. Since many useless warnings hides real warnings, we can't produce many warnings for incompatibilities. So, it is useful that a mechanism which selects warnings to generate. If the warnings for incompatibilities are not generated in useless cases, we can add many warnings.

The module mechanism can also be improved for treating incompatibilities. Since Ruby has open class, method addition can cause incompatibilities. The incompatibilities can be reduced by name spaces for method names. We are considering the module systems for method names such as selector namespace, difference-based modules[5], classboxes[6], etc. They eases library usability improvement because an old method and new method can coexist even if they have same method name.

Acknowledgments

The author make grateful acknowledgment for the valuable comments from Yukihiro Matsumoto (matz), Kenichi Handa, Mikiko Nnishikimi, Satoru Tomura, Yutaka Niibe (gniibe) and anonymous referees.

References

[1] Larry Wall. Programming is Hard, Let's Go Scripting... December 2007, `http://www.perl.com/lpt/a/997`

[2] Yukihiro Matsumoto. The Power and Philosophy of Ruby. O'Reilly Open Source Software Convention (OSCON), July 2003, `http://www.rubyist.net/~matz/slides/oscon2003/`

[3] Yukihiro Matsumoto. The World of Code (in Japanese). Nikkei BP, May 2009, ISBN 978-4-8222-3431-7

[4] Akira Tanaka. open-uri, Easy-to-Use and Extensible Virtual File System. International Ruby Conference, October 2005, `http://www.a-k-r.org/pub/rubyconf2005-presen.pdf`

[5] Yuuji Ichisugi and Akira Tanaka. Difference-Based Modules: A Class-Independent Module Mechanism Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP), pages 62–88, LNCS 2374, 2002.

[6] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling Visibility of Class Extensions. In Computer Languages, Systems and Structures, Volume 31, Number 3-4, pp. 107-126, May 2005.

Work In Progress: an Empirical Study of Static Typing in Ruby

Mark T. Daly Vibha Sazawal Jeffrey S. Foster
University of Maryland, College Park
{mdaly,vibha,jfoster}@cs.umd.edu

Abstract

In this paper, we present an empirical pilot study of four skilled programmers as they develop programs in Ruby, a popular, dynamically typed, object-oriented scripting language. Our study compares programmer behavior under the standard Ruby interpreter versus using Diamondback Ruby (DRuby), which adds static type inference to Ruby. The aim of our study is to understand whether DRuby's static typing is beneficial to programmers. We found that DRuby's warnings rarely provided information about potential errors not already evident from Ruby's own error messages or from presumed prior knowledge. We hypothesize that programmers have ways of reasoning about types that compensate for the lack of static type information, possibly limiting DRuby's usefulness when used on small programs.

1. Introduction

In recent years, there has been considerable interest in lightweight, general-purpose scripting languages. The exact definition of a scripting language is debatable, but one common feature is *dynamic typing*, in which types are strongly enforced but are not checked until the last possible moment during execution. While dynamic typing is flexible and admits a range of interesting and useful coding patterns, it also risks runtime type errors that could be found proactively by a static type system.

In this paper, we describe an in-lab pilot study of programmer use of types in Ruby, an object-oriented, dynamically typed scripting language. We collected data from four skilled programmers as they completed two small programming tasks in Ruby, one using the standard Ruby interpreter, with dynamic typing, and one using Diamondback Ruby (DRuby), which adds static type inference to Ruby [Furr et al. 2009b,a]. DRuby includes type system features like intersection and union types, parametric polymorphism, structural object types, and optional and variable type lists for method signatures. Prior experience shows that DRuby finds errors in a range of existing Ruby programs, when used by DRuby's authors [Furr et al. 2009b,a]. In our study, we aim to understand whether DRuby's static type system actually helps typical Ruby programmers find and fix errors—if not, why not, and if so, how could we improve DRuby's type system to better serve programmers' needs?

Based on qualitative analysis of participant experiences, we made three tentative findings. First, using an open coding technique [Strauss 1987] to classify DRuby error messages produced during participant trials, we found that under 20% of DRuby's error messages were *informative*. Second, in interviews, participants reported that they did use types as part of their reasoning process during development. These two findings seem to be at odds—DRuby's type error messages are not helpful, but types themselves are important. We believe the disparity can be explained by the small scale of the programming task studied: In small, single-author programs, developers can rely on their own memory and naming conventions to track type information.

Finally, we found that all four participants used IRB, the interactive Ruby shell, to explore ideas during development. IRB even served as a documentation source for method names and return types. This suggests that DRuby should also offer an interactive interface, possibly by integrating DRuby with IRB.

2. Background

The Ruby Programming Language Ruby is a strongly typed, object oriented programming language whose concise syntax and flexible, dynamic type system is intended to provide programmers with the latitude to write programs in whatever way they wish. The language's creator asserts that, "I want to make Ruby users free. I want to give them the freedom to choose... if there is a better way among many alternatives, I want to encourage that way by making it comfortable" [Venners 2003]. The success of Ruby is reflected both by the considerable community of users and enthusiasts who contribute to its evolution, and by its use as a host language for the popular Ruby on Rails web framework.

In our experience, Ruby's plasticity is a double-edged sword: because the language's interpreter performs few

static checks, extensive testing may be required to find programming errors. As a result, practices such as "test-driven development" [Beck 1999], wherein tests are written even *before* code is written, are popular in the Ruby community. However, as is well-known, testing is necessarily incomplete, which raises the question, could static analysis benefit Ruby programmers?

Diamondback Ruby: Static Type Inference for Ruby Diamondback Ruby (DRuby) is a static type inference system for the Ruby programming language. DRuby has been used to identify type errors in a number of small, existing Ruby programs, with most programs requiring little modification to be compatible with DRuby's analysis [Furr et al. 2009b]. Subsequent work showed how to scale up DRuby to highly dynamic language constructs and larger programs [Furr et al. 2009a]. While these results are promising, it is difficult to predict how and to what end such a type inference system would be used by programmers. We would like to know, does static type inference present information to programmers that helps them correct errors?

3. Method

Our pilot study of programmer behavior consists of an inductive, two-treatment, repeated measures experiment in which participants solve short Ruby programming exercises. The experimental conditions differ in either applying DRuby or not to the participant's source code each time the participant executes the Ruby interpreter.

Tasks We gave the participants two programming exercises: writing a simplified sudoku solver and writing a maze solver. The former problem is a simplification of problem found on the *Ruby Quiz* website [Gray 2008], and the latter was inspired by the "Gang of Four" design patterns book [Gamma et al. 1995]. The exercises are of approximately equal difficulty and have little overlap, to discourage direct code reuse. We also aimed for exercises that are complex enough to warrant using DRuby while remaining solvable within the experimental protocol's time limits.

Protocol Each exercise consists of three components: a textual problem description, starter code, and set of test cases that target the top-level API participants are expected to implement. The problem description defines the programming task, describes any data input and output formats, and provides pseudo-code for algorithms that the participant can use to solve the problem. The starter code consists of any boilerplate we expected not to vary among solutions. For the sudoku solver, we supplied a method to iterate over the cells of a serialized sudoku puzzle and a method to calculate the grid region of a given cell. For the maze solver, we supplied methods to parse textual maze definitions. Finally, the test cases give participants a way to run their solution, and we deem solutions that pass all test cases to be correct. The programming task packages as presented to participants are available online at `http://www.cs.umd.edu/~mdaly/druby_pilot_problems.tar.gz`.

Experimental Setup We recruited four participants from a local Ruby users group. We targeted participants in this way because the behavior of novices may not reflect that of more practiced participants [Mayer 1981], and we expected users group members to be comfortable with Ruby. All participants indicated that they are quite familiar with the Ruby programming language. Participants may or may not have used DRuby prior to this study—previous experience (or lack thereof) was not a prerequisite for participation.

The pilot was conducted in a laboratory setting. Participants selected their first exercise, and were allowed as much time as they wished to digest the textual problem description. After participants indicated they were done reading the problem description, we allowed them one hour of programming time. (Participants were not shown the time, but some chose to monitor it themselves.)

Participants used a single development platform, with a standard keyboard, mouse, and monitor. We configured the platform with Emacs, Vi, and TextMate, which are popular with Ruby on Rails programmers [Bray 2007]. We also gave participants access to the Ruby core documentation and, except for the first participant, the Internet. (The first participant was not given Internet access to prevent the use of existing code in this study, but we quickly realized this was a mistake. Participants who followed were simply asked not to copy existing solutions.)

The use of DRuby was randomly selected for one of each participant's problems. DRuby was enabled automatically for executions of the selected problem, requiring no additional action by participants. DRuby is a drop-in replacement for the Ruby interpreter that first performs static type inference and then runs the standard interpreter.

We recorded screenshots and audio as participants worked. Whenever a participant ran the Ruby interpreter or DRuby, we made a snapshot of the source code and the output of the interpreter or DRuby. (The first participant's output had to be recreated after the study due to issues with our software.)

At the end of the first problem, participants could take a break at their discretion before beginning the other problem. After the two programming periods had finished, we asked participants to complete a short questionnaire, and we also interviewed the participants informally to assess their reaction to DRuby and to the study as a whole.

4. Participant Experiences

Next we discuss the experiences of our four participants, ordered chronologically.

Participant A Participant *A* indicated that he is equally comfortable with Java and Ruby, and is somewhat familiar with the C programming language.

Participant *A* only finished about a quarter of each exercise. The reason is that he was given *no* starter source code,

which is what our protocol originally stipulated. As a result, participant *A* barely got to write the portions of his solution that might have lead to type errors, rendering use of DRuby mostly moot. We added starter code for subsequent participants to address this issue, and the other participants were able to nearly complete all their exercises.

Although participant *A* made very little use of DRuby, he did take preemptive action to avoid a type error in which data read from a file must be explicitly coerced into an integer. He identified this particular error without the assistance of the Ruby interpreter, DRuby, or any other automated means. Screen recordings show him adding an integer constant to certain variables that store data from a file, and then writing explicit coercions for these variables at an earlier point in the program. While the arithmetic operation may have lead him to find this potential error, we do not know for sure.

In our interview, participant *A* discussed the role of types in Ruby programming. He indicated that he maintains imprecise mental knowledge of types: *"I know that types are there. When I read in a file, I know that I've got a string; [when] I split on newline, then I know I've got an array, so in my head... I have usually an idea that I've got an enumerable. I'm not sure if it's an array or something else..."*

Participant B Participant *B* said that he is quite familiar with Ruby, but is most familiar with Java. He indicated that he is as familiar with C# and Groovy (a Java-like dynamic language) as he is with Ruby, and somewhat so with Python.

Participant *B* encountered some bugs in our data collection tools during the course of writing his solutions. This interfered with some executions of his program and caused him to make some unnecessary edits to his code. Using his feedback, most of these issues were corrected.

In his first programming problem, participant *B* encountered a significant type error: where participant *A* caught the necessary string-to-integer coercion step early on, participant *B* did not discover this until the Ruby interpreter raised a "TypeError" exception. After encountering this error, participant *B* spent several minutes making extensive edits to his program to solve the problem. This occurred during the trial that did not use DRuby.

During his interview, participant *B* described how he continuously keeps type information in mind to supplement the lack of type annotations in Ruby source code. Discussing his experience with Ruby, he stated, *"...with a dynamic language, I'm just kind of subconsciously* always *thinking about types."* In contrast, he explained that, *"...when I'm coding Java, I'm not even thinking about [types], because it's already done for me. So if I make a mistake, the compiler is doing that for me. So, I'm almost consciously just not caring, and so I don't really worry about keeping that stuff in mind..."*

Participant C Participant *C* stated that he is equally familiar with Java and Ruby. Additionally, he indicated some familiarity with C++ and Scheme.

Participant *C* encountered a type error in which he used a single-element array where a value was expected as the contents of an array cell. This error resulted in a failure of the supplied test case, which rather confusingly reported that "4 != 4." The strange error message occurred because of the default printing method for Ruby arrays: a single-element array is printed as just the element itself, without brackets (unlike multi-element arrays). This error happened during the trial that did not use DRuby, and required several minutes of the participant's time to diagnose and correct.

In his interview, participant *C* said that he might not benefit from the sort of error messages he saw reported by DRuby. He explained, *"I do find myself...regularly checking the types of objects to make decisions, usually when I'm making rendering decisions, 'how do I want to render this,' where knowing 'does this object respond to a certain method' [i.e., what DRuby could report] isn't really what I need to know."* This position is understandable given that many of the DRuby error messages he saw concerned calls to methods that had not been implemented. Moreover, when asked to consider shortcomings of Ruby's standard dynamic type system, he stated that he has not been disappointed: *"...my expectations were lowered and then adjusted, so it was more, 'don't rely on types.'"*

Participant D Unlike the previous participants, participant *D* said he is equally familiar with Perl, C, and Ruby. He also said that he is quite familiar with C++ and moderately so with Haskell.

Interestingly, participant *D* made few, if any, type errors during his development. With the exception of some (Ruby interpreter) errors due to uninitialized hash table cells, none of the error messages produced by participant *D*'s test executions indicated a type mismatch.

In his interview, participant *D* said that the relatively small scope of the solutions he was asked to write made DRuby's error messages rather ineffectual. He said, *"it would usually be faster to run the test suite without running the static checks, because [the programming challenges] were such small programs,"* and that, *"[DRuby] usually told me things that I already knew, like...I hadn't implemented a particular method yet—I knew I hadn't implemented a particular method yet, but wanted to see the initialization go through."*

5. Results

Because of the limited number of participants in our study, it is difficult to come to definitive conclusions. Nevertheless, we were able to inductively formulate several tentative hypotheses using the data we gathered; we expect to investigate these more fully in future studies.

DRuby's Error Messages: Correct but Not Informative To analyze the DRuby error messages that our participants received, we assigned each error message to one or more cat-

egories using open coding. Open coding is a method of inducing hypotheses from qualitative data by comparing fragments of data with each other, assigning attributes (called codes) to each fragment, and grouping fragments together into categories based on those codes [Strauss 1987].

To categorize the DRuby error messages, we considered each message with respect to other simultaneously reported messages, any warnings produced by the Ruby interpreter in the same execution of the participant's program, any code changes made by the participant since the last execution of the program, and all DRuby messages that preceded it.

We ended up with seven primary codes for DRuby error messages: a) *Duplicate*: multiple messages representing the same error for different sites in a single execution; b) *Intentional*: the result of an intentional edit with obvious consequences; c) *Expected*: seen in an earlier execution or expected from starting conditions; d) *Identical*: same as a warning message reported by Ruby; e) *Additional*: not reported by Ruby for that execution; f) *New*: previously unreported error; g) *Recurrence*: previously seen message from a reintroduced bug. In the example output:

```
[ERROR] instance Sudoku does not support \
  methods print_puzzle
  in method call s.print_puzzle
  at ./sudoku.rb:33
  in typing ::Sudoku.new
  at ./sudoku.rb:32

[ERROR] wrong arity to function, got exactly \
  1 arguments, expected no arguments
  in solving method: initialize
  in typing ::Sudoku.new
  at ./sudoku.rb:32

[ERROR] wrong arity to function, got exactly \
  1 arguments, expected no arguments
  in solving method: initialize
  in typing ::Sudoku.new
  at ./sudoku.rb:34

sudoku.rb:32:in `initialize': wrong number of \
  arguments (1 for 0) (ArgumentError)
        from sudoku.rb:32:in `new'
        from sudoku.rb:32
```

the first DRuby message (prefixed with `[ERROR]`) is coded as *Additional*. The second and third would be coded as *Identical* since the same warning is reported by Ruby (the final message), and the third as *Duplicate* because it is the same as the second. If these errors occurred in a previous execution or if this was one of the first executions of the program (when the programmer has not had a chance to write any methods yet), these would also be marked *Expected*; otherwise the first error would be coded as *Recurrence* or *New* depending on whether or not it had occurred and been fixed before. The *Duplicate*, *Expected*, and *Identical* codes were applied to messages very frequently. The *Additional* and *New* codes were applied less frequently, and the *Intentional* and *Recurrence* codes were applied to very few messages.

These codes were grouped into categories representing whether a message did or did not provide information to the programmer in excess of what they could be expected to already know or could have obtained through using Ruby alone. Messages were assigned to one primary category, either *Informative* or *Not Informative*, based on the codes they had received: a message was assigned to *Informative* if it had been given at least one of *Additional*, *New*, or *Recurrence* exclusively; otherwise, it was assigned to *Not Informative*.

The primary theme that emerged from our analysis is that DRuby did not reliably contribute much useful information. While a limited number of error messages were classified as *Informative* by our open coding scheme, the majority were not: excluding data from participants *A* and *B*, who experienced problems with the protocol and data capture software that were already discussed, 13.4% of error messages were classified as *Informative*, and only 20% executions where DRuby reported at least one error contained any *Informative* error messages. These percentages are lower if participants *A* and *B*'s data is included.

That said, none of the messages produced by DRuby were incorrect, and it may be that DRuby is useful for larger projects but not for the small programs in our study. One of DRuby's key advantages over standard testing is that it analyzes all code paths, including obscure ones—of which there may be few in small programs with straightforward control flow. Further investigation will be required, however, before we can make this claim with confidence.

Programmer Conventions as Type Annotations The coding technique applied to DRuby's error messages shows that DRuby did not report much useful information. However, our interviews indicated that types are part of participants' reasoning processes. This disparity is troubling, as we would expect programmers to find type errors more easily with the aid of DRuby. There may be, however, other mechanisms at work that prevent type errors in the first place.

While it is always wise to give methods and arguments names that indicate what they mean or do, they can also be used to encode type information. In this example (from participant code, as are all those that follow), the parameter names indicate their types directly:

```
def validate_digits(array, str)
```

This is not the only way that participants encoded type data into their method definitions. The type signature of the method:

```
def set_value_at(x,y,value)
```

is also partly obvious in the context of a program that uses a two-dimensional grid. It is a reasonable guess that `x` and `y`

are integer coordinates; `value` could have any type, but the programmer would probably be able to easily remember its specific type. Other method definitions in the same program include:

```
def get_value_at(x,y)
def row_values(x)
def col_values(y)
def grid_values(cx,cy)
```

Again, arguments that include `x` and `y` in their names are probably integers. Each method's name contains `value` or `values`, and so will probably return the same type of objects that `set_value_at` takes as an argument. These are not precise type signatures, but programmers do not necessarily need precision when dealing with small programs.

Another convention also appeared in participants' code:

```
def open?(sym)
```

The use of a question mark at the end of methods does not change a method's behavior, but is a general Ruby convention: "Methods that act as queries are often named with a trailing `?`, such as `instance_of?`" [Thomas et al. 2004]. In this case, we are asking if an object is open or not for some symbol, and so expect `open?` to return a boolean.

All of our participants wrote method definitions that appear to specify some amount of argument or return type data. Ad-hoc conventions may have helped to limit type errors, but further study would be required to know for certain.

Sources of Type Information: Ruby as its Own Reference Several participants indicated that they rely on their own memory to compensate for the lack of explicit type information in Ruby. Moreover, in reviewing interview tapes and screen recordings, we found that participants used several resources when their memory was insufficient: They gathered information from the Ruby Core documentation, the Internet at large, the `ri` utility (a command-line tool for accessing Ruby documentation), and IRB, the interactive Ruby shell. Based on our recordings, IRB is by far the preferred method for exploring features of Ruby; participant *A* even said in his interview that he uses reflection in Ruby to look up method names. (The "methods" method can be invoked on a class to get a list of its methods.) IRB was the only information resource employed by all four participants; one participant used IRB for everything from experimenting with certain Ruby constructs, to manually loading and executing portions of his program, to looking up a particular method's return type.

If programmers prefer IRB over other forms of Ruby reference material, then tools like DRuby may be more effective if they provide interactive documentation as well. In its current form, DRuby provides type documentation through rich type annotations written in comments. A more effective form of documentation may integrate annotations into the output produced by IRB (as is done, for example, by OCaml's interactive shell).

6. Threats to Validity

One key threat to the validity of our study is the scale of the programs written by the participants. In our experience, many Ruby programs are created by writing larger, reusable libraries and then writing small main programs; our study captures only the latter. Additionally, one very common use of Ruby is to write programs in Ruby on Rails, which is not included in our study—Rails code is not statically analyzable by DRuby by itself [An et al. 2009].

Another uncontrolled variable is the effect of DRuby itself on participants' workflow. During this study, DRuby typically took about 80 times longer to analyze and run participants' code than when run just under Ruby. This delay in execution was clearly noticeable, and may have motivated participants to test their programs less frequently when using DRuby—this change in debugging practices may have affected their development processes, though we cannot know for certain.

7. Related Work

While a great deal of research has been conducted regarding human factors in software development, little work has focused specifically on the effect of type systems on programmer behavior. Gannon [1977] studied the error rates in solutions to programming problems written in untyped vs. statically typed variants of a programming language. However, in Gannon's study, participants were graduate and advanced undergraduate students, while our participants were recruited from a users group for the language being studied. Additionally, type systems in particular and programming languages in general have evolved a great deal since Gannon's work.

Ng Cheong Vee et al. [2005] explored the effect of various kinds of compiler error messages on both novice and "mature" students using Eiffel, categorizing errors based on log data collected during the study. Yang et al. [2000] investigated manual type checking practices in Standard ML, but used existing code containing errors rather than having participants write their own programs. Recently, Hanenberg [2009] completed preliminary research on the effect of typed vs. untyped variants of a novel language, finding that programmers worked faster in the untyped version.

While DRuby was selected for this research, other static type inference and checking systems for dynamically typed languages exist. Morrison [2006] developed a type inference approach that is used by the RadRails IDE for Ruby on Rails. Because this inference system is built into a specific IDE, however, it was not well suited to our study. Several type inference systems for Python have been developed by Aycock [2000], Cannon [2005], and Salib [2004]; additionally, Ancona et al. [2007] have created a statically typed subset of Python that can be compiled to CLI or JVM bytecode. Similar systems, such as CMUCL [MacLachlan 1992] and SBCL [SBCL 2008], have been developed for Lisp.

8. Future Work

Our pilot study allowed us to gain insight into the practices of Ruby programmers and to refine our experimental protocol. There are several interesting directions for future work.

An alternate approach to our study would be to conduct surveys. Ayewah and Pugh [2008] surveyed users of FindBugs, a static analysis tool for Java, to gain an understanding of how it is used in practice. They also have investigated the use of FindBugs in industrial settings [Ayewah et al. 2007]. However, similar studies with DRuby (or other static type systems for dynamic languages) would first require a sizable user-base, which we do not believe currently exists.

Another direction would be to scale up our study to larger programs. We could ask participants to identify errors in existing software projects of varying size and complexity, and observe whether DRuby helps them find and fix bugs.

A study of programmers working as a team might also be interesting. Code changes by multiple developers may cause inconsistencies in their respective understandings of a program's types, creating opportunities for type errors. This might also allow us to investigate the use of conventions as annotations, as the type information encoded by one programmer may not be obvious to another.

Acknowledgments

We wish to thank Michael Hicks and the anonymous reviewers for their helpful comments on earlier versions of this paper, Mike Furr and Elnatan Reisner for assisting in the testing of our protocol, and our study participants. This research was supported in part by DARPA ODOD.HR00110810073 and NSF CCF-0915978.

References

J. D. An, A. Chaudhuri, and J. S. Foster. Static Typing for Ruby on Rails. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, Nov. 2009. To appear.

D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: Reconciling Dynamically and Statically Typed OO Languages. In *DLS*, 2007.

J. Aycock. Aggressive Type Inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

N. Ayewah and W. Pugh. A report on a survey and study of static analysis users. In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-051-7. doi: http://doi.acm.org/10.1145/1390817.1390819.

N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-595-3. doi: http://doi.acm.org/10.1145/1251535.1251536.

K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, October 1999.

T. Bray. Ruby survey results, November 2007. `http://www.tbray.org/ongoing/When/200x/2007/11/26/Ruby-Tool-Survey`.

B. Cannon. Localized Type Inference of Atomic Types in Python. Master's thesis, California Polytechnic State University, San Luis Obispo, 2005.

M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the twenty fourth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2009a.

M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static Type Inference for Ruby. In *OOPS Track, SAC*, 2009b.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, 1977. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359763.359800.

J. E. Gray. Ruby quiz, February 2008. `http://www.rubyquiz.com/`.

S. Hanenberg. What is the impact of type systems on programming time? – first empirical results. Proceedings of the 2009 Workshop on Evaluation and Usability of Programming Languages and Tools, October 2009.

R. A. MacLachlan. The python compiler for cmu common lisp. In *ACM conference on LISP and functional programming*, pages 235–246, New York, NY, USA, 1992. ISBN 0-89791-481-3. doi: http://doi.acm.org/10.1145/141471.141558.

R. E. Mayer. The psychology of how novices learn computer programming. *ACM Comput. Surv.*, 13(1):121–141, 1981. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356835.356841.

J. Morrison. Type Inference in Ruby. Google Summer of Code Project, 2006.

M. Ng Cheong Vee, B. Meyer, and K. L. Mannock. Empirical study of novice errors and error paths. Unpublished technical report, 2005.

M. Salib. Starkiller: A Static Type Inferencer and Compiler for Python. Master's thesis, MIT, 2004.

SBCL 2008. Steel Bank Common Lisp, 2008. `http://www.sbcl.org/`.

A. L. Strauss. *Qualitative Analysis for Social Scientists*. Cambridge University Press, Cambridge, United Kingdom, 1987.

D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2nd edition, 2004.

B. Venners. The Philosophy of Ruby: A Conversation with Yukihiro Matsumoto, Part I, Sept. 2003. `http://www.artima.com/intv/rubyP.html`.

J. Yang, G. Michaelson, and P. Trinder. How do people check polymorphic types? Proceedings of the 12th Workshop on the Psychology of Programming, April 2000.

What is the Impact of Static Type Systems on Programming Time?

Preliminary Empirical Results

Stefan Hanenberg

University of Duisburg-Essen,
Institute for Computer Science and Business Information
Systems, 45117 Essen
e-mail: stefan.hanenberg@ icb.uni-due.de

Abstract

Although static type systems are an essential part in teaching and research in software engineering and computer science, there is hardly any knowledge about what the impact of type systems on the development time for a piece of software is. On the one hand there are authors that state that static types decrease an application's complexity and hence its development time. On the other hand there are authors that argue that static types increases development time since they restrict developers to express themselves in a desired way. This paper presents some preliminary results of an experiment that studies the impact of a static type system on the development speed of an application. The results reveal that the existence of the type system significantly turned out to increase the development time under the experiment's conditions.

Categories and Subject Descriptors D.1.0 [**Programming techniques**]: General

General Terms Measurement, Experimentation, Languages, Human Factors.

Keywords *Type Systems, Empirical Study, Development Time*

1. Introduction

Type systems (see for example [1, 10]) are one of the major topics in research, teaching as well as in industry. In research, new type systems appear frequently either for existing programming languages (as for example the introduction of Generics in Java) or new programming languages constructed for studying a certain type system.

In teaching, students are educated in the formal notation of type systems as well as in proofs on type systems (see for example [1, 10]). In industry, type systems become important for different reasons. Possibly a programming language in use evolves by introducing a new type system. If this new type system should be applied, developers need to be educated which produces additional costs. Maybe existing libraries or products should be adapted to match the new type system which produces additional costs as well. Finally, additional tools might be required due to the new type system (such as tools that measure the current state of the software product) which potentially produces additional costs as well.

For industry it is important to determine whether such an investment is reasonable, i.e. whether the expected benefit of type systems represents some future revenues. This means, it is necessary to understand what the advantages and maybe additional costs of using a type system are.

In industry, it is observable that untyped programming languages such as PHP or Ruby become more and more important for specific domains such as website engineering. Furthermore, untyped programming languages such as TCL or Perl are still used in software development. For future developments of such languages it seems valid to ask, whether new releases of such languages should provide a type system – assuming that a type system has a positive impact on software development.

However, while type systems are well-studied from the perspective of theoretical computer science, there is hardly any knowledge about, whether a type system plays a relevant role in the practical application of a programming

This paper was published in the proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) at the ACM Onward! Conference. October, 2009. Orlando, Florida, USA.

language. There is even a number of researchers that advocate the use of untyped programming languages instead of typed ones (see for example [14]) – and who emphasize the tradition of untyped programming languages such as Smalltalk.

This paper contributes to this question by a first, small evaluation of a larger data set gathered in an experiment whose focus is on type systems. Our focus in the experiment is on development time. Hence, topics such as readability, understandability and maintainability of typed / untyped programs are out of the scope of this paper.

We will show that within the experiment the use of the untyped programming language turned out to have a significant positive impact on the development speed, i.e. subjects using the untyped programming language turned out to be significantly faster than the ones using the typed programming language.

Section 2 briefly discusses frequent arguments for and against type systems. Section 3 introduces the experiment. Section 4 analyses the data. After discussing some related work in section 5, section 6 discusses and concludes this paper.

2. Typed vs. Untyped Programming Languages

In literature, there is a number of arguments for or against type systems. For example, [2] argues pro type systems as follows:

- Type systems can capture a large fraction of recurring programming errors
- Type systems have methodological advantages for code development
- Type systems reduce the complexity of programming languages
- Type systems improve the development and maintenance in security areas

On the other hand common arguments against type systems can be found for example in [4, 9, 14]:

- Type systems unnecessarily restrict the developer
- No-such-method exceptions which are caused at run-time because of missing type-checks do not occur that often
- No-such-method exceptions mainly occur because of null-pointer exceptions (which occur in typed programming languages as well)

The arguments for and against type systems seem to be valid but contradict each other. For industry, a common problem here is that it is unclear which arguments should be trusted. If for example the decision is whether the new type system of Java 1.5 should be used, additional costs for training, etc. become relevant. However, without the information whether there is any return on investment, it is hard to make a corresponding decision.

3. Experiment

The study introduced here is part of a larger experiment which focuses on the impact of type systems. While the data of the experiment is not yet completely evaluated, this paper focuses on a subset of gathered data. Section 3.1 gives an overview of the whole experiment and the part of the experiment this paper focuses on. Section 3.2 briefly describes the programming language used in the experiment. Section 3.3 describes the experiment setting, section 3.4 describes the measurement and 3.5 discusses the validity of the experiment.

3.1 Experiment Overview

In the experiment, 49 subjects (undergraduate students, selected using convenience sampling [15]) were asked to write a simplified Java parser, whereby the data used within this paper relies only on 26 subjects. The specification of the simplified Java parser to be written was delivered via a context-free grammar. The subjects had passed already fundamental Java programming courses as well as fundamental courses on formal languages.

Based on an interview the subjects were divided into two groups where the intention of the interview was to detect more experienced developers among the students and to equally divide them into the two groups with the same capabilities. Each group was trained in a programming language written for the experiment. The only difference between both languages was, that the one had a type system while the other one had not.

After training, each subject had a fix amount of time to implement the parser, whereby an additional task was to implement a simple scanner (the data collected for the scanner is in the focus of this paper).

3.2 Programming Language

For the experiment, a new object-oriented programming language (similar to Smalltalk or Ruby) called Purity was written in two versions: a typed as well as an untyped version. The reason for writing a new language was, that we wanted to exclude any influence on the experiment caused by subjects who already know the language being used. Consequently, the language was being taught as part of the experiment and the language and its documentation is not available in public (in order to prevent subjects to learn the language outside the experimental setting).

Purity is a simple object-oriented language with single inheritance and late binding. Corresponding to the language design of Smalltalk, Purity does not distinguish between primitives and objects. The language provides blocks, which is the Smalltalk version of closures. Purity contains few language constructs. Similar to the language design of Smalltalk, `if` is not a language construct itself,

but is provided by corresponding methods in class Boolean. The only loop provided is the `while`-loop which is represented by a method in block objects. Field access is only provided via methods, i.e. only an object itself is allowed to access its fields. The language does not provide any constructs for multithreading.

The programming language includes a simple API with 29 basic classes such as Integer or String or LinkedList. Additional to the programming language, a small IDE was provided, which includes a class browser, a test browser (for running the application and tests of applications) and a console window.

The type system of typed Purity is non-generic and nominal and can be compared to the type system of Java up to version 1.4. The typing of blocks was implemented according to the proposal that can be found in [6].

3.3 Experimental Setting

The programming language was taught in 16 hours to the subject. After that, each subject had to implement a task. The task was to implement a Parser for a simple Java-like grammar within 27 working hours divided into 4 working days. The corresponding context-free grammar was described by the Backus-Naur notation.

These 27 hours were controlled and took place in the experimental environment. Coffee-breaks etc. were not included in these 27 hours. The subjects were permitted to arrange their time freely, i.e. they were permitted to arrive at different times. The subjects were not permitted to take any material from the experimental environment. I.e. the language itself, handbook, etc. always stayed in the experimental environment.

3.4 Measurement

As already stated, the here described paper is not based on the complete measurement of the experiment. Instead, we only rely on the first part of the experiment, the implementation of the scanner. The task was to remove characters such as white space or line feet from the input string and to create a list of tokens.

While the experiment was running, all changes in the code base were logged so that it was later on possible to reconstruct all user inputs at a certain point in time: whenever the developer added or removed a class or whenever a method was added to (or removed from) the system, a corresponding log entry was generated in the background. For the scanner we defined a set of 15 test cases that represents the minimal functionality required in the system. These test cases were not delivered to the subjects.

Based on the log entries and the test cases we were able to determine the point in time when subjects fulfilled these test cases. We did that by reconstructing all developer entries and after any change in the code we rerun the tests. We considered the point in time when all test cases were fulfilled as the reference point where the developer finished a minimal scanner.

The data used here in the paper comes from 26 subjects, 13 from the group with the typed language and 13 from the group with the untyped language.

3.5 Threats to validity

There are some points that threaten the validity of the experiment.

Untyped		
Subject	sec	hours
1	8141	2.26
2	23041	6.40
3	2851	0.79
4	42501	11.81
5	24666	6.85
6	13975	3.88
7	7260	2.02
8	11224	3.12
9	18317	5.09
10	7335	2.04
11	16994	4.72
12	12766	3.55
13	18334	5.09

Typed		
Subject	sec	hours
14	40970	11.38
15	6305	1.75
16	32199	8.94
17	28463	7.91
18	32112	8.92
19	29509	8.20
20	28752	7.99
21	8972	2.49
22	23413	6.50
23	56985	15.83
24	3653	1.01
25	40864	11.35
26	46032	12.79

Figure 1. Experiment results

First, it can be argued that 26 hours training is not enough for learning a new language (and a new IDE). However, it should be emphasized that the language, its API as well as its' IDE was kept very simple, so that we considered the training to be sufficient. The only problematic element we identified was that the subjects had problems to understand the semantics of blocks due to the Java background of the subjects. However, since this problem was equal for all subjects, we consider it to be less problematic for the internal validity of the experiment.

	sum	max	min	arith. mean	median
untyped	57.61	11.81	0.79	4.43	3.88
typed	105.06	15.83	1.01	8.08	8.20

Figure 2. Descriptive data (in hours)

A second point is more problematic: it was not explicitly requested from the subjects that they have to finish the scanner completely before continuing with writing the parser. As a consequence, we cannot be sure at what point in time the subjects switched to the parser task without finishing the scanner task. In fact, this is the reason why we consider in this paper only 13 subjects of each group: for these subjects it was obvious that they spent the whole time on the scanner until the test cases were fulfilled.

4. Analysis

We start the analysis by describing the experiment's results and then performing significance tests in order to check whether there are significant differences in the development times using the untyped and typed programming language.

4.1 Results and Descriptive Statistics

Figure 1 shows the measured results of the experiment, i.e. the number of seconds (and hours) required by each subject to fulfill the test cases. For example, on the left hand side, the third line says that subject number 3 required 2851 seconds (respectively 0.79 hours) in order to fulfill the minimum requirements. Subject number 24 (which is a subject that used the typed language) required 3653 seconds (respectively 1.01 hours).

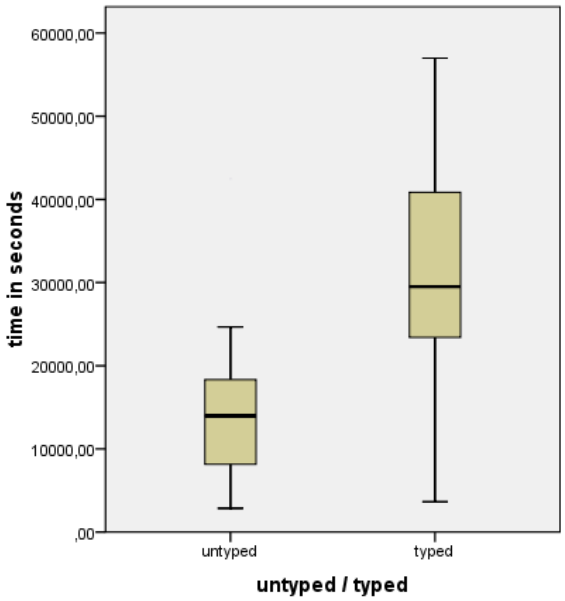


Figure 3. Box plot

Based on the experiment results, we computed the descriptive statistics (see Figure 2). Here, we see that the sum of development times (and hence the arithmetic mean), the maximum, the minimum and the median of the group with the untyped programming language is less than the corresponding values of the group with the typed programming language. Furthermore, we see that the differences are quite high (for example, the sum of development times of the untyped solution is 207,450 seconds, while the sum of times for the typed solution is 378,229 seconds). a more obvious representation of these differences, Figure 3 shows a box plot for the untyped and typed group, which visualized the lower quartile, the median and the upper quartile of the underlying data set.

Because of these results, it seems reasonable to assume, that there is a significant difference between both times which will be checked in the next section.

4.2 Checking Significant Difference in Means

In order to check whether there is a significant difference between both values it is necessary to formulate a hypothesis which needs to be checked by a corresponding significance test.

Since the experiment design is a one factor design (with the variable typing), and since each subject solves the programming task only once either using the typed or using the untyped programming language, a significant test for independent samples is required.

The most conservative test that can be applied here is the (non-parametric) Mann-Witney U-test which does not assume the underlying data to be normally distributed (cf. [3]) and which compares for two samples whether they come from the same population.

		N	Mean Rank	Sum Ranks
Language	Untyped	13	10.23	133
	Typed	13	16.77	218

Figure 4. Ranks

The underlying hypothesis (and alternative hypothesis) to be tested is

> **H0**: The data for typed and untyped development times are drawn from the same population (i.e. equals medians)
>
> **H1**: The samples come from different populations

Figure 4 shows the resulting ranks for the sample. There, it can be seen that the mean rank for the untyped language is 10,23 while the mean rank for the typed language is 16,77. Based on this, we computed the p-value, i.e. the probability that there is no difference between the medians of development times, which is p=0.029. Since $p < 0.05$, the null hypothesis can be rejected, i.e. both samples come from different populations. This implies that the means of both samples significantly differ. By checking the mean ranks, it turns out that the development times using the untyped language are significant lower than the development times using the typed programming language (mean rank 10.23 in comparison to 16.77).

4.3 Computing Difference in Means

While the previous section determined that there is a significant difference in the means of the development times of typed and untyped solutions, it did not state how large this difference is (note that the use of the arithmetic mean or the median is not valid here).

	t	df	sig (2-tailed)	mean diff.	95% confidence interval	
					lower	Upper
Language	-2.524	24	0.019	-13140.3	-23887	-2393

Figure 5. T-Test results

First, we check, whether we can assume that the underlying data is normally distributed. This is done using the Shapiro-Wilk test (cf. [3]) which computes a value p. In case p is larger than 0.05, the test does not reject the assumption that the underlying data is normally distributed.

Computing the p-values reveals for the untyped programming language a value p=0.11 and for the typed programming language a value p=0.59. Hence, in both cases the test does not reject the hypothesis of a normal distribution of the underlying sample.

Hence, we can perform a t-test for independent samples which will reveal the size of the difference between both samples.

Figure 5 shows the results of the t-test. Again, we can see that there is a significant difference between both samples (with p=0,019), but we have this information already from the previous section. However, the important information is the 95% confidence interval which states that with the probability of 95% the development time using the untyped language is between 2393 and 23887 seconds less than the development time using the typed language. Hence, the advantage of using the untyped programming language in the experiment is approximately between 40 and 400 minutes.

5. Related Work

We are aware only of two works in the area of empirical evaluations of type systems. The first one by Prechelt and Tichy [13] concentrates on the impact procedure argument type checking. Prechelt and Tichy check the impact of typing by letting the subject perform a programming task in a typed as well as in an untyped language with the result that there is a significant positive impact on the developer's productivity. It is noteworthy, that the results of the experiment in [13] contradict the results of the experiment introduced here.

The second experiment we are aware of is the one performed by Gannon [5]. There, a controlled experiment was performed that compared the use of a typed vs. an untyped programming language. The experiment also revealed a positive impact of static types.

One further experiment by Prechelt [12] could be considered as an experiment that compares typed and untyped programming languages. However, the main focus of the experiment is not the typing issue (but the question whether or not the language is a script or compiler language) and there is no data available from within the development progress.

In [8] we made a first evaluation of the costs for using the untyped language based on the same data as used here in this paper. There, it turned out that between 0% and 24% of all test runs of the untyped solutions failed due to a NoSuchMethod exception. Based on these results, it is even more surprising that the untyped solutions turned out to take less time than the typed ones, because it seems clear that additional runtime costs caused by NoSuchMethodException (which do not exist in the typed solutions) slow down the development speed.

6. Conclusion and Further Work

In this paper we presented preliminary results of an experiment that explores the impact of a type system on the development time of a piece of software. We showed that within the experiment the use of the type system has a significant negative impact on the development time, i.e. the type system caused an additional overhead.

Based on an computation of the 95% confidence interval we were able to approximate the difference between both approaches and it turned out that the use of an untyped programming language took between 40 and 400 minutes less than the typed solutions. Compared to the maximum time required for the typed solution (56985 s according to Figure 1) this means that the untyped solution required between 4% and 42% less.

The interesting point in the study is that we rather assumed that developers benefit from the typed language, especially, because the language in use was new to them. Hence, we rather expected a positive impact of typing because it is commonly assumed that typing improves the ability to learn new APIs – since typing restricts a possible faulty use of a new API.

It must not be forgotten that there are some threats to validity that threaten the experiment's results. The most problematic point here is the to understand how developer differ with each other. Here, it would be more desirable to have some more objective metrics available that better permit to classify subjects.

Although this difference is quite impressive, a much deeper analysis is required on the data. First, we would like to measure the execution time of test runs for untyped solutions and compare it with the time developers required

to solve typing problems – maybe the effect of the difference is, that the developers using the type language wrote less efficient test cases. Hence, maybe the result of the experiment is mainly derived from the execution times of the applications and not from the effect of typing.

While we already made some preliminary studies about the frequency of NoSuchMethodExceptions in untyped solutions, a comparison of their execution time and the time required to solve typing errors in the typed solutions is up to future work.

Another interesting point about the experiment results is, that it contradicts studies such as [13] where a positive impact of typing could be measured. We are currently not aware of how to explain this contradiction. Possibly, the effect of typed programs is better in some situations while it is not in others. However, since the knowledge about different "kinds of programs" is currently quite restricted, it is currently unknown whether different "kinds of programs" have a different impact on the use (and the impact) of type systems.

Even though the experiment seems to suggest that typing has not a positive impact (at least not such a huge positive impact as often claimed) it must not be forgotten that the experiment has some special conditions: the experiment was a one-developer experiment. Possibly, typing has a positive impact in larger projects where interfaces between developers need to be shared.

A further point that needs to be emphasized is, that the experiment here addresses only pure programming time. Possible influence on design time, readability or maintenance of software cannot be concluded from the experiment's results.

While the discussion about typing mainly concerns the design of languages in known areas such as object-oriented programming or functional programming, our intention is in the future to study the impact of typing in newer approaches such as aspect-oriented programming. We did a first evaluation of aspect-oriented programming already in [7], but did not consider any typing effects there.

A general conclusion is that definitively more studies are needed in order to determine the impact of type systems. Since type systems play such an important role in software development it is rather tragic that the discussion about the need for typed and untyped programming languages is mainly driven by personal experiences and personal preferences of researchers instead of empirical knowledge received from an adequate research method.

References

[1] Bruce, K.: Foundations of Object-Oriented Languages: Types and Semantics, MIT PRESS, 2002

[2] Cardelli, Luca: Type Systems, In: CRC Handbook of Computer Science and Engineering, 2nd Edition, CRC Press, 1997.

[3] Conover, W. J.: Practical nonparametric statistics, 3rd edition, John Wiley & Sons, 1998.

[4] Eckel, B.: Strong Typing vs. Strong Testing, mindview, 2003, http://www.mindview.net/WebLog/log-0025, last access: August 2009

[5] Gannon, J. D.: An Experimental Evaluation of Data Type Conventions, Comm. ACM 20(8), 1977, S. 584-595.

[6] Graver, J. O.; Johnson, R. E.: A Type System for Smalltalk, Seventeenth Symposium on Principles of Programming Languages, 1990, pp. 136-150

[7] Hanenberg, S., Kleinschmager, S., Josupeit-Walter, M: Does Aspect-Oriented Programming Increase the Development Speed for Crosscutting Code? An Empirical Study. Accepted for publication at 3rd International Symposium on Empirical Software Engineering and Measurement, 2009.

[8] Hanenberg, S.: Costs of Using Untyped Programming Languages – First Empirical Results. 13th IFAC Symposium on Information Control Problems in Manufacturing (Track Advanced Software Engineering). Moskow, June 3-5 2009.

[9] Lamport, L.; Paulson, L. C.: Should your specification language be typed, vol. 21, ACM, New York, NY, USA. 1999.

[10] Pierce, B.: Types and Programming Languages, MIT Press, 2002.

[11] Prechelt, L.: Kontrollierte Experimente in der Softwaretechnik: Potenzial und Methodik, Springer-Verlag, 2001.

[12] Prechelt, Lutz: An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical Report 2000-5, March 2000.

[13] Prechelt, L., Tichy W.: A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking, IEEE Transactions on Software Engineering 24(4), 1998, S. 302-312.

[14] Tratt, L., Wuyts, R.: Dynamically Typed Languages. IEEE Software 24(5), 2007, S. 28-30.

[15] Wohlin, C., Runeson, P., Höst, M.: Experimentation in Software Engineering: An Introduction, Springer, 1999

Empirically Investigating Parallel Programming Paradigms: A Null Result

Meredydd Luff
University of Cambridge
Meredydd.Luff@cl.cam.ac.uk

Abstract

The dominant paradigm of concurrent programming has well-publicized usability problems, but the alternatives have not been well analyzed from a usability perspective. I attempted an empirical comparison of programmer productivity using the Actor model, transactional memory, and traditional lock-based concurrency paradigms. The results were inconclusive. I discuss my experiment, present its results, and discuss possible reasons why such experiments are a blunt tool with which to investigate programming language usability.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.2.2 [*Design Tools and Techniques*]

General Terms Human Factors, Experimentation

1. Introduction

With the widespread arrival of multi-core processors, it is becoming necessary to write parallel programs to fully exploit nearly any modern computer.

However, parallel programming has proven extremely difficult. In particular, the dominant paradigm – that of multiple threads sharing writable memory, and controlling access to it with mutual-exclusion locks – has come in for much criticism for its usability failings ([Lee 2006], among many others).

The design of concurrency mechanisms for programming languages is a serious and painfully unresolved problem in HCI. The adoption of multi-core processors is leading us to put ever more weight on an interface widely regarded as inadequate. Arguably, then, this one of the most important problems in the usability of programming systems.

1.1 Contributions

I describe an experiment to directly measure and compare programmer performance using different parallel paradigms to solve a problem. I lay out some background in Section 2, then describe and justify my methods in Section 3.

I present the results of this experiment in Section 4, and show them to be inconclusive.

I discuss possible causes of this inconclusive result, and the difficulty of using controlled empirical experiments to evaluate programmer productivity, in Section 5.

2. Background

2.1 Alternative paradigms

Language designers have proposed many alternatives to the dominant paradigm of lock-based concurrency [Skillicorn and Talia 1996]. I set out to investigate two paradigms that have gained some popularity, and claim superior usability to the current *de facto* standard.

2.1.1 Message Passing

Message-passing systems provide a private memory for each thread of control. All communication is by discrete messages passed to other threads, which process them one at a time. Examples include Hoare's Communicating Sequential Processes [Hoare 1978], or the more dynamic Actor model [Hewitt et al. 1973] most popularly associated with the Erlang programming language.

2.1.2 Transactional Memory

Transactional memory systems have threads of control sharing a single memory space. Their memory accesses are grouped into transactions, and a run-time system detects and rolls back conflicts to ensure that each transaction occurs atomically or not at all. [Peyton-Jones 2007]

Transactional memory is quite similar to classic threading and locking. The difference between manual locking and transactional memory can be compared to the difference between manual and garbage-collected memory management [Grossman 2007].

2.2 Evaluation

Like most programming language features, concurrency paradigms have historically been built on a hunch, and evaluated by anecdote and holy war. Even when parallel programming systems *are* evaluated for usability, researchers compare whole languages and runtime systems rather than the principles they embody (see Related Work in Section 6). This makes their results less informative for the design of new programming systems.

I chose to evaluate these paradigms empirically with a controlled experiment. In this experiment, subjects solved the same problem, in the same language, varying only the concurrency paradigm.

3. Materials and Methods

3.1 Experiment structure

The goal of this experiment was to compare programmer performance using different parallel paradigms, keeping the programming language and environment constant.

I tested three parallel paradigms: the Actor model, transactional memory, and standard shared-memory threading with locks (henceforth "SMTL"). I also tested subjects writing sequential code, as a positive control: it is generally agreed that sequential programming is substantially easier than SMTL, and any acceptably powerful study should show this effect clearly.

I provided all four programming models for the Java programming language. Java is a widely adopted language, taught in the Cambridge undergraduate curriculum, and provides an uncontroversial baseline for this experiment.

Each subject solved the same problem twice: once with the standard (SMTL) Java threading model, and once with Actors, transactional memory, or no parallelism at all (the sequential condition). The whole session took approximately 4 hours per subject. Scheduling was balanced, with half of the subjects solving the SMTL condition first, and the other half solving it second.

Each subject filled out a questionnaire before the experiment, to assess their level of experience and self-perception of skill. After each task, the subject filled out a questionnaire indicating the level of subjective difficulty and opinions of the concurrency model used.

During the experiment, subjects' screens were recorded, and a webcam recorded the subject to monitor off-computer events. A snapshot of the project directory was taken at 1-minute intervals, and instrumented tools logged each invocation of the compiler or run of the resulting program.

3.2 Subjects

Seventeen subjects were recruited from the undergraduate (10) and graduate (7) student population of the Cambridge Computer Laboratory. Of these, eleven successfully completed both tasks.

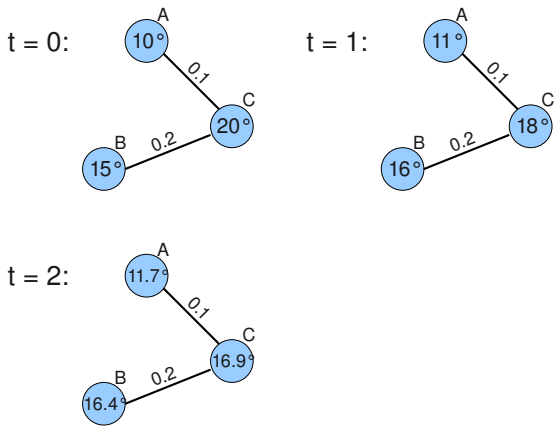
3.3 Task

I chose to minimize variability, at the expense of significant learning effects, by using a single problem. This was an "unstructured grid" problem, in the classification of the View from Berkeley [Asanovic et al. 2006].

I presented a toy physics problem, modelling heat flow between identical blobs, connected by rods of varying conductivity. If the temperature of blob i at time t is represented as T_t^i, the temperature change due to a rod of conductivity k connecting blobs a and b is:

$$T_{n+1}^a = T_n^a + k(T_n^b - T_n^a)$$

A graphical example is shown below:



To reduce the time subjects spent writing irrelevant I/O code, I provided them with code to load a sample data set into memory and pass it as arguments to a function call. I provided the inputs in a deliberately un-useful data structure, and the written instructions instructed each subject to translate the data into their own choice of structure. I instructed the subjects to translate the solution back to the original un-useful format and call a function to check its correctness.

3.4 Implementing Different Paradigms

I aimed to replicate the user experience of programming in each paradigm, while keeping the underlying language as close as possible to idiomatic Java.

This involved trade-offs which made performance or scalability comparisons between different conditions impractical. This is a disadvantage, as scalability is the ultimate goal of all such parallel programming, but other studies have examined the scalability characteristics of different concurrency mechanisms, and I considered usability the more important target for the present study.

3.4.1 SMTL and sequential

Java's native concurrency model is based upon threads and mutual-exclusion locks, so standard Java was used for the SMTL and sequential conditions.

3.4.2 Transactional Memory

For the transactional memory condition, I used an existing transactional memory system for Java, Deuce [Korland et al. 2009]. Deuce modifies Java bytecode during loading, transforming methods annotated with `@Atomic` into transactions. For example:

```
class TransactionalCounter {
  private int x;

  public @Atomic increment() {
    x = x + 2;
  }
}
```

However, at the time of this study, Deuce did not instrument classes in the Java standard library, so it could not be used with idiomatic Java. Instead, I enabled Deuce's "single global lock" mode, which makes all atomic methods mutually exclusive. This preserves the semantics of transactional memory, but prevents us from evaluating scalability.

3.4.3 Actor Model

Implementing the actor model for Java presented a challenge. Java assumes mutable shared memory throughout its design, whereas actors require disjoint memories and messages which the sender cannot change after they are sent.

One of the touted advantages of the actor model is that the system enforces actor isolation, preventing the user from making a class of mistakes. Enforced isolation is therefore necessary to realistically model the desired user experience.

I considered Kilim [Srinivasan and Mycroft 2008], an implementation of the Actor model in Java with an annotation-based type system to enforce actor isolation. Kilim enforces a complete transfer of ownership of mutable objects sent in messages, so that only one actor can refer to a mutable object at any one time. However, this is a substantial departure from idiomatic Java. In addition, no version of Kilim including this type system is publicly available, and the author warned that his pre-release version was unreliable. I therefore concluded that Kilim was not suitable for this experiment.

I also considered using a run- or compile-time system to ensure that only immutable objects – objects whose fields are all `final`, and members are similarly immutable – could be passed in messages. However, Java's standard library is built with mutable objects. I wished to evaluate "Java with Actors", not "Java with Actors and all standard data structures removed", and so this option was also rejected.

Instead, I implemented a reflection-based runtime system, using deep copies for isolation. In my implementation, each actor is represented by an object, to which no reference is held by any other actor. Other actors interact only with a wrapper class, `Actor`, through which they can send messages to this hidden object. Messages are named with strings, and handled by methods of the same name, so an `"add"` message is handled by the `add()` method.

All arguments to messages and constructors are deep-copied, using the Java serialization mechanism. This enforces isolation, by preventing multiple actors from obtaining pointers to the same mutable object. (Isolation can still be violated, by use of `static` fields, but I decided that this could be verbally forbidden without seriously affecting coding style.)

An example use of this framework follows:

```
// In AddingActor.java:
public class AddingActor {
  private int x = 0;

  public void incrementBy(int y) {
    x += y;

    System.out.println("x␣=␣" + x);
  }
}

// In Main.java:

public class Main {
  public static void main(String[] args) {
    Actor a = new Actor("AddingActor");
    a.send("incrementBy", 2);
    a.send("incrementBy", 5);
  }
}
```

Of course, this isolation comes at a significant performance cost, making scalability analysis impractical. I do not claim that this grafting of isolation onto a shared-memory language is an elegant one – merely that it models the user experience I wished to emulate.

4. Results

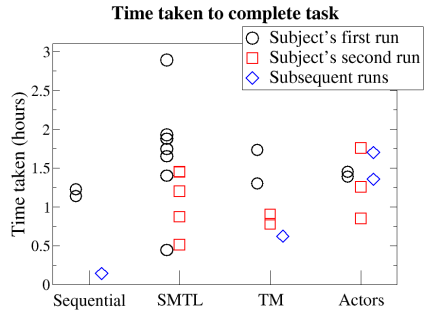
The results of this experiment were inconclusive, showing no significant difference in any objective measurement between the four test conditions. The subjective measurements, however, did show a statistically significant preference for the transactional memory (TM) model over shared-memory threading with locks (SMTL), and suggest a (not statistically significant) preference for the Actor model over SMTL.

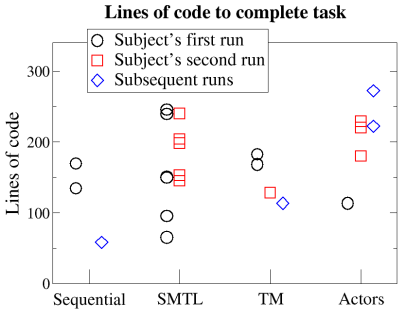
I also document an unexpected phenomenon in the completion data, suggesting a bimodal distribution: Subjects either completed the first task within two hours, or could not within the whole four-hour session.

4.1 Objective Measures of Programmer Effort

Programmer effort is difficult to measure objectively, and sophisticated proxy measurements are contentious. I therefore chose two simple metrics: the time taken for subjects to complete the task, and the number of (non-comment) lines of code in the final program.

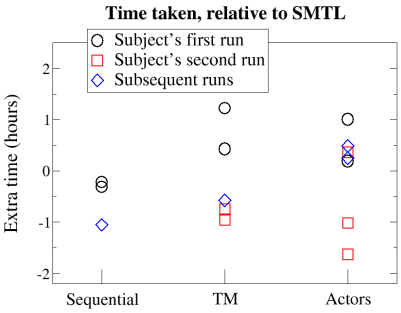
We begin with the aggregate data, showing completion times for every trial on the same graph. In all the graphs in this section, lower numbers indicate better performance.

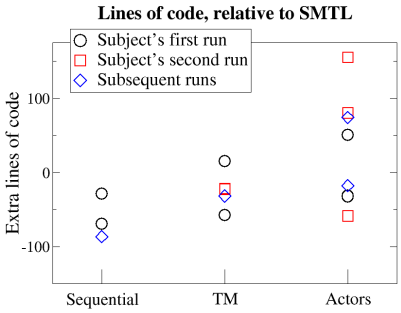




The aggregate data shows no significant difference between the four conditions. This is unsurprising, given the high inter-subject variability shown above.

We now consider within-subject differences in performance between conditions. Each point on the following graphs represents the difference between a subject's performance in the test condition and the SMTL control. Lower numbers indicate better performance than on the SMTL trial; higher numbers indicate worse performance.





Overall, these measurements showed no significant difference between the conditions under test.

In the "time taken" metric, the difference between first and second trials dominates all other variation. The sequential condition (our positive control) shows a suggestive decrease in time taken, but the other two conditions appear dominated by this learning effect.

The learning effect is significant: a paired t test finds a significant difference between subjects' first and second trials ($p = 0.04$).

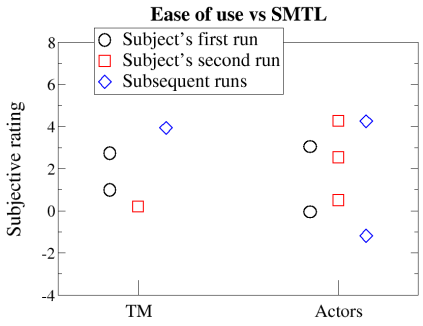
In the "lines of code" metric, we do not see such a difference between first and second trials. The sequential condition again suggests a decrease, but there is nothing particularly convincing here.

In both metrics, the Actor model shows remarkably high variation in performance.

4.2 Subjective impressions

After each task, subjects were given a questionnaire requesting their subjective impression of the task. As well as overall task difficulty, they were asked how the framework compared to writing sequential code and SMTL, and asked directly how easy they found using the framework.

This graph presents the average of all these subjective ratings (sign-normalized such that higher ratings are good, and presented as a difference from ratings in the SMTL condition). The sequential condition is omitted, as most of the survey questions were inapplicable to it.



A within-subjects ANOVA finds that subjects tested on transactional memory significantly preferred it to SMTL ($p = 0.04$).

The preference of subjects tested on the Actor model did not reach the 5% significance level ($p = 0.07$).

4.3 Metric correlation

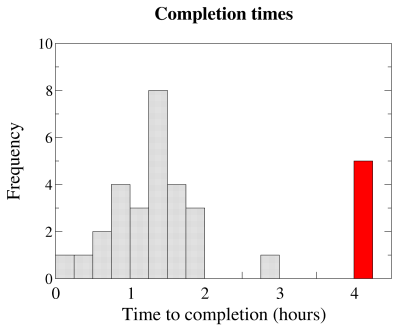
These three metrics (time taken, lines of code, and subjective rating) appear to differ widely. I observed a weak correlation between time taken and lines of code ($r = 0.25$), and between time taken and subjective rating ($r = -0.18$), but none at all between lines of code and subjective rating ($r = 0.05$).

This small correlation suggests that they measure *something*, but their disagreement implies that these metrics should not be wholly trusted.

4.4 Completion times: Does the camel have two humps?

An unexpected phenomenon is visible in the subjects' completion times: Subjects either finished the first run within two hours, or could not finish it at all within the four-hour session. (There was only one exception, who completed the first task in three hours. He did not attempt the second.)

The distribution of completion times for all tasks is displayed below. The solid bar represents subjects who did not complete a task at all. The yawning gap between these five subjects and the rest of their cohort suggests that the distribution is not continuous:



A bimodal distribution of programmer ability has been posited before [Dehnadi and Bornat 2006], but this controversial suggestion distinguished between those who entirely "cannot learn" to program and those who can.

By contrast, these unsuccessful subjects wrote valid code, with control and data structures no less complicated than their successful peers'. They did not give up, or stop debugging their programs, until stopped at the four-hour mark. Post-session interviews indicate that they correctly understood the problem. In short, the circumstantial evidence does not support the idea that these subjects were simply incompetent, gave up, or failed to understand the problem.

This result, then, remains an intriguing mystery.

5. Discussion

This inconclusive result is disappointing, but also instructive.

Broadly, it illustrates the difficulties of empirical research into such a complicated phenomenon as programmer productivity. It also illustrates the power of many hard-to-avoid confounding factors, which I will discuss in a moment.

This study also has something to say about the relative merits of subjective and objective research. It is telling that the subjective survey results actually succeeded in reaching statistical significance, whereas the empirically-observed proxies for effort barely suggested anything.

This is not to say that subjective studies are inherently superior – their substantial biases are the reason we have empirical studies in the first place. However, human introspection can sometimes tell us more readily about difficult-to-measure phenomena such as "effort" than objective (but weak) proxies.

5.1 Possible explanations

I will now consider some confounding factors which might have caused this inconclusive result, even in the presence of large usability differences in the frameworks tested.

5.1.1 Weak metrics

Their dismal internal consistency suggests that the available metrics for programmer effort are not powerful or reliable tools. More accurate instruments would be required to measure any but the largest effects.

5.1.2 Learning effects and subject variability

The learning effect between the two trials for each user greatly interfered with the results. However, designing a study such as this one inevitably puts the experimenter between a rock and a hard place.

If I had designed an experiment where each subject attempts only one task, under one condition, the results would have been swamped by inter-subject variation. It would take an impractical number of subjects to see any effect at all.

The design I actually chose controls for some subject variability – but far from all – at the expense of a short-term learning effect that ends up swamping the results.

5.1.3 Familiarity

Most programmers have experience with the standard threading model, and so come to this study with a substantial familiarity bias in favor of the SMTL condition. By contrast, few subjects had previous experience with transactional memory, and none with the Actor model. This study therefore captured the effort required to learn these models for the first time, as well as the effort of solving the problem.

This might be mitigated with a practice session before the study, in which users gained some experience with the unfamiliar model before attempting the task. However, no length of practice session can eliminate the learning curve entirely, and familiarity is an ever-present confounding factor.

5.1.4 Toy Problem Syndrome

The task in this experiment was, necessarily, a small problem which could be solved in two hours and 200 lines of Java. Most solutions included a single, symmetrical concurrent kernel. By contrast, the usability problems of concurrency are often related to complexity, and the inability to hold the behavior of all threads in the programmer's head at once.

The proverbial "concurrency problem from hell" is an intermittent deadlock whose participants are distributed across

80,000 lines of code written by ten different people. Such situations are difficult to model experimentally.

It is a perennial problem that empirical studies of programming can only ever test toy problems, and it is difficult to imagine any experimental design which could sidestep it. Lengthening the task might have helped slightly, but recruiting volunteers for a four-hour study was difficult enough as it was. There is a reason that much of the work in this field is done by educators, at institutions where course credit may be awarded for participating in such experiments.

5.2 Additional threats to validity

Lest we think that these big problems are the only ones, a number of issues would have qualified even a significant positive result from this experiment:

- The choice of problem greatly affects the suitability of different paradigms. There is unlikely to be One True Model which outperforms all others, all the time. Choice of a problem which can be more easily modelled with shared state, or message passing, could therefore have a significant effect on the results.

  Controlling for this issue would require a large-scale study including representatives of, say, all the Berkeley Dwarfs [Asanovic et al. 2006], in an attempt to cover all common parallel computations.

- Software spends almost all of its life being maintained, but this experiment observed only the initial creation of a program. Safe and easy modification – low *viscosity*, in the terms of the Cognitive Dimensions framework [Green and Petre 1996] – might even be more important than the ease with which a program is first written.

- The frameworks used in this experiment lack features available in industrial implementations. For example, the Actor model implementation lacked multicast or scatter-gather support, which is available in implementations such as MPI [Snir and Otto 1998]. The SMTL condition explicitly prohibited the use of utilities such as synchronization barriers from the package `java.util.concurrent.*`. Subjects repeatedly re-implemented both barriers and scatter-gather patterns during the experiment.

- Actor messages, unlike normal method calls, are only checked at run time, not at compile time. This may have spuriously reduced performance in that condition.

- The unrealistic performance of the experimental frameworks might have cause subjects to mis-optimize their programs. (This was not borne out by observations or discussions after experimental sessions; no subject so much as profiled their code for bottlenecks.)

- The students participating in this study may not be representative of professional programmers. This could go either way: they might be more flexible and open to new techniques, or less practiced at quickly getting the hang of unfamiliar tools.

- The awkward data structures used to provide inputs appear to have had an anchoring effect on the subjects. Several subjects largely copied the provided data structures instead of devising their own, despite emphatic instructions to the contrary.

6. Related Work

The study closest to the present one was conducted by IBM [Ebcioglu et al. 2006]. They compared performance using three languages for supercomputing clusters: MPI (message passing in C) [Snir and Otto 1998], Unified Parallel C [El-Ghazawi et al. 2005], and IBM's x10 language [Charles et al. 2005]. x10 was convincingly superior, but I believe that this is probably due more to its garbage collection, memory safety and similarity to Java than to its approach to concurrency. No significant difference was found between MPI and UPC.

This group also described the methods used to observe subjects during that experiment [Danis and Halverson 2006]. Although the software they used is not publicly available, this significantly inspired my methods.

Hochstein et al. [2008] taught a class of students either MPI or a shared-memory C variant for a research computer architecture. They found MPI to involve significantly more effort, but as VanderWiel et al. [1997] note, most of the extra effort is likely to do with manually packing and unpacking message buffers rather than message-passing *per se*.

That study is part of a larger collaboration between several universities to investigate HPC usability, using students from scientific computing classes [Hochstein et al. 2005].

A number of other studies have evaluated the usability of parallel systems by implementing larger projects (often benchmarks), and discussing the experience. These are not controlled experiments, but they avoid the Toy Problem Syndrome, and can control for the differences in structure between problems. Cantonnet et al. [2004] evaluated UPC on the NAS benchmark suite, Chamberlain et al. [2000] compared Fortran variants, Single-Assignment C and ZPL on a single NAS benchmark, and VanderWiel et al. [1997] compared several C-based languages and HPF over a variety of benchmarks.

Most of this work either predates the multi-core era, or concentrates on distributed-memory supercomputing systems. It is therefore not enormously useful for evaluating the pressing problems facing general-purpose computing today, or the proposed solutions, which make heavy use of language features not available in Fortran or C. However, their methods, and quality of results, are still instructive.

Acknowledgments

I am extremely grateful to Luke Church and Alan Blackwell for their assistance, suggestions and feedback while designing and performing this experiment, as well as to my supervisor, Simon Moore.

References

K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelik. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, December 2006. URL `http://www.gigascale.org/pubs/1008.html`.

Franois Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. Productivity analysis of the UPC language. *Parallel and Distributed Processing Symposium, International*, 15:254a, 2004. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2004.1303318.

Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 46, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1103845.1094852.

C. Danis and C. Halverson. The value derived from the observational component in integrated methodology for the study of HPC programmer productivity. In *P-PHEC workshop, held in conjunction with HPCA*, February 2006.

Saeed Dehnadi and Richard Bornat. The camel has two humps (working title). 2006. URL `http://www.cs.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf`.

Kemal Ebcioglu, Vivek Sarkar, Tarek El-Ghazawi, and John Urbanic. An experiment in measuring the productivity of three parallel programming languages. In *P-PHEC workshop, held in conjunction with HPCA*, February 2006.

Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC*. Wiley, 2005. ISBN 9780471220480.

T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

Dan Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 695–706, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297080.

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359576.359585.

L. Hochstein, V.R. Basili, U. Vishkin, and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 81(11):1920–1930, 2008. cited By (since 1996) 2.

Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 35+, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 1-59593-061-2. doi: 10.1109/SC.2005.53. URL `http://dx.doi.org/10.1109/SC.2005.53`.

Guy Korland, Nir Shavit, and Pascal Felber. Poster: Noninvasive Java concurrency with Deuce STM. In *Systor '09, Haifa, Israel*, 2009.

Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2006.180.

S. Peyton-Jones. Beautiful concurrency. In G. Wilson, editor, *Beautiful Code*. O'Reilly, 2007.

David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30:123–169, 1996.

Marc Snir and Steve Otto. *MPI - The Complete Reference*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262692155.

Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *European Conference on Object Oriented Programming ECOOP 2008*, 2008.

Steven P. VanderWiel, Daphna Nathanson, and David J. Lilja. Complexity and performance in parallel programming languages. *High-Level Programming Models and Supportive Environments, International Workshop on*, 0:3, 1997. doi: http://doi.ieeecomputersociety.org/10.1109/HIPS.1997.582951.

Mashups on the Web: End User Programming Opportunities and Challenges

Nan Zang
The Pennsylvania State University
nzz101@psu.edu

Abstract

In the last 10 years, the social nature of the web has created a shift in the way people use the computer. In particular, advanced programming activities are no longer reserved for those who are professionally trained programmers. Moreover, the increased socialization of the web has encouraged users to create more readily available content for everyones use. However in some cases what a user wants is not necessarily easy to accomplish and in many cases still require programming skills.

In this paper, I describe some past studies of web mashups, an integrated web application that combines data from multiple sources into a single interface. Mashups provide us with a unique opportunity to study how both professional programmers and non-programmers approach an inherently programmatic technology. In some cases the issues encountered by the end user coincide with those of the programmer. Using lessons learned from a survey study, interviews and a think-aloud study, I propose directions for future research.

Categories and Subject Descriptors D.m [*Software*]: Miscellaneous – Software Psychology

General Terms Human Factors

Keywords End-user programming, mashups, APIs, and tools

1. Introduction

The Web, without a doubt, has a major influence on our every day lives. We use it as a tool to gather information and as an avenue of communication. We use it for work and for entertainment. It is everywhere we go, and over the past few years it has become a ubiquitous source of information. This abundance of information on the web is in part due to the evolution of the Internet as a communication medium, but also due to the huge number of online authoring tools and services that has encouraged the average web user to share their own creation on the web. For example, in January 2009, the video sharing web site YouTube announced that there is an average of 15 hours of video uploaded every minute [1]. Essentially, the web is designed for the active participation of its users; the more users, the more content that is created, and the more useful the web becomes.

However not all of these new technologies are easy to understand and use. Many of the novel systems introduced require skills that not every end user may have. For example, web Application Programming Interfaces (APIs) – one of the core technologies used by almost every major web system, including Google, Yahoo, Microsoft and Amazon – requires advanced programming skills to employ. One approach to solving this skill barrier is to introduce easier to use or simpler tools. However, what happens when this easy to use tool does not meet the requirements of the user? The user then needs to find and adopt another tool for a specific task. What if the user cannot find a tool that fits his or her needs? Moreover, with each additional tool needed, the user must commit cognitive resources to learning and remember its use. As with any technology or tool, there may be a substantial learning curve before one can reap the benefits.

Generally, software developers cannot come close to supporting every need of every user in the tools they create. Tasks are generalized and simplified, and tools are created for the common needs. Instead of introducing a new tool for every need, developers employ numerous methods to help users help themselves. For example, in spreadsheet software macro writing functionality is introduced to allow users to create their own solutions to repetitive tasks. However, again this solution requires some technical skills. In most cases, users still need to maintain a level of programming or computational expertise to leverage these features.

The remainder of this paper examines our work in the area of end-user programming. While this work focuses on more novice programmers or non-programmers, we found that in some cases both programmers and these end users

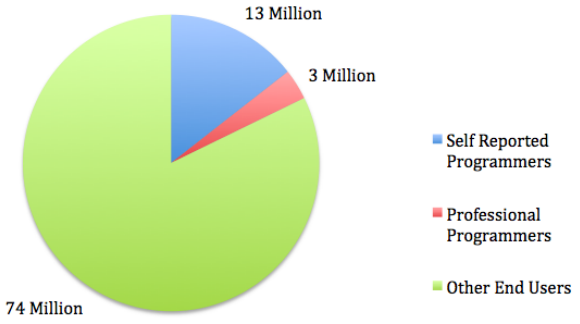


Figure 1. Number of US Programmers and End Users by 2012

encountered similar problems when dealing with programming tasks on web.

2. End-User Programming

The term end user can have many different meanings based on the context in which it is used. In particular, our work has focused on a subset of end users primarily identified as end-user programmers. This group is typically defined as people who participate in programming tasks but whose primary job function is not programming. One early definition of this population explains end users are not:

> " 'casual,' 'novice,' or 'naïve' users they are people such as chemists, librarians, teachers, architects, and accountants, who have computational needs and want to make serious use of computers, but who are not interested in becoming professional programmers. [2]"

However, this description has evolved over the years. As reflected by the research in the early 1990s, the majority of computer users and programming was done either around workplace or in schools [2]. Thus many of the end users described during that period were either professionals who need to get work done or students who are being taught using a computer. If we think about these topics in todays world, the end user population is far more diverse.

Using data from a survey conducted by the US Bureau of Labor Statistics, researchers have estimated that by 2012 there will be over 90 million Americans using a computer at work, with 13 million self-reported programmers, but less than 3 million professional programmers [3]. With this increasing user population, programmers cannot hope to create systems that fulfill the nuanced needs of every user. Instead, designers and developers must find a way to allow users to help solve their own problems. This is the underlying goal of end-user programming (EUP) research.

3. Mashups

Recently there has been an upsurge of research interest in mashups - a web application that combines multiple data sources and presentations into one interface. From an EUP perspective, mashups provide users with the opportunity to integrate and assimilate many different web resources into a personalized view. However, because of the relatively complex programming methods needed to create a mashup, more non-programmers and novice users are left out. Specifically, programmers usually leverage APIs or screen scraping to retrieve online data for a mashup. As a result, numerous tools have been created and studied that attempt to help these naïve end users to create these advanced web applications [4, 5].

Mashups are interesting to us because they provide an opportunity to study end users working with cutting edge technology. Moreover, the goals of a novice user creating a mashup could easily match with the motivations of a more experienced programmer. I would suspect that the majority of the 4361 mashups listed on ProgrammableWeb – a website that tracks mashups – could be used end users to accomplish a variety of goals and complete various tasks on the web [9]. However, the web applications listed there were created by programmers; novice end users do not have the skills necessary to create such mashups. Again, the end users must turn to and rely on developers.

4. Survey of Mashup Creators

We began our inquiry into mashup development by first surveying the developer population [10]. Because mashups tend to be context-specific, we assumed that mashup development would be a solution used by developers, but not a primary job function. So we advertised our survey on numerous web development and API related forums. This effort resulted in a total of 63 responses, with 31 who had created mashups before. When we compared all the developers to the 31 with mashup experience, we found that the 31 developers had more expertise with programming and web technologies.

As a part of this study, we wanted to understand how these developers learned to create mashups. As to be expected for a novel technology, our surveyed developers all taught themselves using the documentation for different APIs. Probing more deeply, we found that these developers found that the documentation is very inadequate and do not provide the correct level of abstraction. Documentation was cited as being very sparse and not regularly updated. One participant specifically pointed out that the Google Maps API would be updated but the documentation did not reflect the changes until much later. Moreover, many developers pointed out a lack of examples and tutorials. Our participants expressed that they realize that they are taking advantage of free and public services, but stress that when approaching a new web service and API, they immediately look for examples and use those to explore. Further, they depend on online doc-

umentation that when they are building mashups, as there was no formalized training in mashup development during the length of the survey. They suggested that having graduated information from beginners to experts by level and a variety of examples that showcased the API's functionality would have helped them when learning to create mashups.

Another insight from this work was that the mashups created by developers are quite limited in variety. There are a limited number of APIs being used for the majority of mashups, and many mashups have only slight differences to each other. The Google Maps API and mapping related mashups were the most common. While it is unclear why this was the case, it is possible that the visual nature of maps combined with the Google Maps API being one of the first major public APIs made this type of mashup more attractive to developers. It may be that while there is an abundance of data on the Web that is interesting, there are fewer useful visualization techniques that are accessible to non-programmers. From a tool developer's prospective, one possible way to further support end users could be to provide these users with simple ways of integrating their data with appealing visualizations.

Overall, we concluded that developers currently use mashups as a solution to data integration problems. The numerous public APIs available serve as a toolset for developers to create interesting applications. The problems encountered by the developers seem to arise from interacting with the APIs and not necessarily with any specific programming language used to create a mashup.

5. Survey of End Users

To more carefully study a novice user population more aligned with EUP research, we distributed a survey to the student population at Penn State University [11]. We speculated that these students would be the types of users that could benefit from being able to create mashups. They grew up with the Internet being prevalent in their lives, and they rely on the web as a tool to solve problems and gather information.

From this study, we identified some key variables that influence how end users approach and think about novel web technologies: Technology Initiative – how active do they pursue new web experiences, and disseminate these to their friends; Usefulness – a self-rating of perceived usefulness of a technology; Sharing hobbies online – the frequency of sharing their own hobby related activities and creations online. Of these three, Technology Initiative and Usefulness seem to play an important role in judgments our participants make about pursuing mashups. We also asked them to rate the difficulty of creating a mashup, assuming that this would be a variable related to their motivations for pursuing mashups. However, this was not the case, as difficulty did not appear to have a major effect on their decisions to pursue mashups in the future.

This study resulted in numerous insights, but most important was the concept of web-active end users. These users are extremely active online and pursue all aspects of online life. While this is a common quality among many college students, the web-active user is motivated to take the additional step by finding better tools to support and enhance their common activities. Moreover, not every web-active end user is the same. When we compared those who had high Technology Initiative to those with lower Technology Initiative, we found that their interests diverged. Users with lower initiative tended to describe social- and people-motivated mashups. For example, one participant wanted a way to bring together all of the different social networking services she used into one interface. Another participant combined pictures of his friends with famous quotes. High initiative users focused on more complex, data-intensive mashups. For example, using news articles as a reference point and integrating additional reference materials, such as Wikipedia articles, other news sites, and video from YouTube. Another participant suggested incorporating product reviews, and missing specs of products when browsing online retailers. This reinforces and clarifies some of the details from the prior study [10]. The types of mashups end users can easily create are important for their initial perceptions of a technology or tool. Being able to support all mashups equally may not be as important as being able to support a specific set of mashups. Focusing support on mashups that are clearly useful to the end user will most likely encourage them to adopt a tool.

A problem area for end user mashups was how end users think about data. We asked them to naïvely describe the steps needed to create a mashup. While most of the respondents could generally describe how to gather data and what to display at the end, the integration of data was not well defined. We decided to study this in more detail since it is the most vital part of the mashup building process.

6. Yahoo! Pipes: Mashups for EUP

The process of building a mashup can be decomposed into three stages: data gathering, data manipulation, and data presentation. End users can easily understand both gathering and presentation; one assumes that to they would know what data they want to mashup and the resulting visualization before approaching a mashup task. To examine data manipulation in more detail, I interviewed 12 students who fit the web-active end user program and also had them create the mashup pictured in Figure 2 using Yahoo! Pipes. I chose Pipes partially due to its focus on data integration, but also because it is essentially a visual programming language. A more detailed report of this work and the results have been reported in [12]. I will briefly review it here.

I began by asking the participants to describe some of the web sites they frequently visited. To assist them I provided five categories of information: News, sports scores,

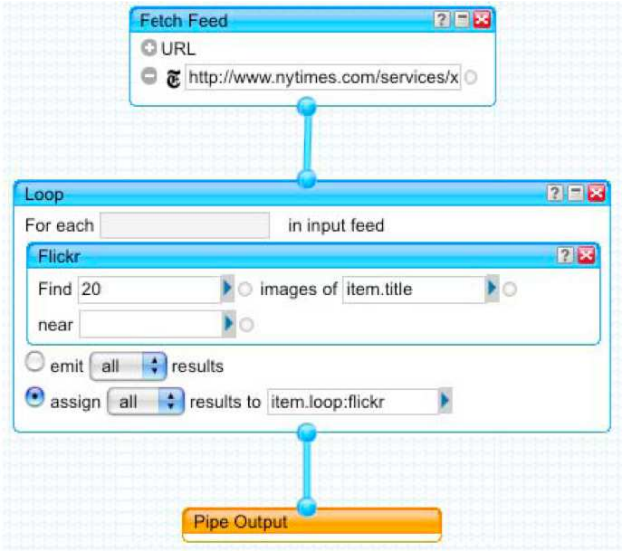


Figure 2. Mashup of New York Times and Flickr using Yahoo! Pipes

weather, shopping, and pictures. They were asked to come up with places online they could find each and then if there were any categories or websites that did not fit. For the most part, users were able to come up with a wide variety of data sources for each category. There were a total of 64 unique information sources. As we would expect, for each of the categories the most popular websites were also the ones mentioned the most by our group. For example, when thinking about shopping information the participants looked to Amazon, and for sports they used the ESPN website. However there were many more obscure webpages mentioned. A common theme in the responses were close connections to local, real world locations. For instance one participant recently moved and no longer had access to a physical copy of her previous town newspaper, but she now she would visit the newspaper's website as a substitute. Websites for local clothing and electronics stores were referred to as common sources of shopping information. I purposely left out social networking to see if the participants would mention the category. All but one referenced social networking and Facebook as one of the sites they spend most of their time on. The participants cited portal services, like MSN, iGoogle, Yahoo, and even Facebook as a common location for much of their general information needs.

When thinking about combinations of data the participants brought up a variety of examples. For example, one participant suggested combining course listings and scheduling with reviews from Rate My Professor. Interestingly, there were few mentions of ideas involving maps and other geographic data; only one participant mentioned location based customizations. This is unexpected because of the large number of mashups created thus far that have been map-based. This points to certain discrepancies with the tools being built. While expert programmers may concentrate on mapping-related mashups, the more naïve and non-programming users focus on other areas. For example, Shopping and Picture based combinations were mentioned the most during the interviews.

Finally, to gauge their understanding of the common underlying data structures on the web, I tested each participant on their comprehension of XML. The reason for this is to better gauge their understanding of the data they were working with in a raw form. Would it be possible for users with similar expertise to understand and then possibly work with XML without much support. I found that in almost every case the participants were able to identify the attributes and content in the code. From their explanations, it seemed that they used tags as a guide to understanding the content, but then made qualify judgments by comparing the two. For example, if a tag was marked "title" and then the content was a garbled string, they would second guess themselves and say "I think it's a title [...]" but not be completely sure, even though they correctly identified the title tag previously. Also, the participants had a hard time understanding HTML embedded into the XML. For example, "description" tags, commonly used as a message body for RSS feeds, many times contain links and images. The HTML representations were often dismissed as being "some code". Generally, the participants were able to pick apart the XML and build an understanding of the underlying data. This is promising because many of the currently available mashup tools, including Yahoo! Pipes, hide the underlying data, claiming that it is added complexity. But these results tell us otherwise. With the proper scaffolding, a novice end user could use the underlying data structures to make better connections between different data sources.

Once they began thinking about different types of data and combining data, I asked them to create a mashup using the Pipes tool. They were given a short tutorial of the interface and then 15 minutes to familiarize themselves with the tool. I used a think-aloud protocol to get at the participant's thought processes while they worked with Pipes. Their actions revealed a mismatch between the Pipes tool to their mental models. Specifically, the names of each model proved to be a major factor in how the end user approached the mashup task. For instance, many users used the Filter module to take the functionality of the Loop module in the mashup in Figure 2. Because theyre deci-sions were solely based on their interpretation of the module name, Filter seemed like a way to "find something that is related to whatever is on the [article] page". Of course, computationally filter actually removes unwanted items in the input feed. This result is similar to prior studies of language use in programming systems [6]. Given enough time, a user may be able to master each module, but currently, they may not even get that far.

7. Discussion

Our work thus far has focused on these Internet users and their perceptions and use of information on the Web, specifically focusing on mashups. From our surveys we realized that mashups are not only for non-programmers, but also those who are experienced developers. Dealing with the myriad of APIs on the web is a test of patience. The lack of adequate documentation, along with the dependency on services beyond the developers control makes mashup development quite difficult. Both non-programmers and developers must have a place to start, and as a relatively new platform the Web poses added difficulty for those who want to program for it. When we looked at end users specifically, the characteristics of a population we call web-active users play an important role. We believe that they are a population of users that are motivated to take on more activities, including even learning to build mashups, if they judge it to be worth while.

A limitation of this work has always been the introduction of the idea of mashups to our participants. It is extremely difficult to describe concepts to a person without giving examples and context. These examples in both our end user survey and in the interview study are biased by this. It most likely would be true that if we introduced a mapping example that participants would think about location based mashup ideas. However, given this possible source of bias, we were still able to illicit many ideas that are unrelated to our initial introduction. We believe that many of our results still provide valuable ideas for developers.

From a tool developer's prospective, it would make sense build a better understanding of this user population and design tools that target their specific motivations. This could include providing simple ways to visualize the data they work with, or packaging information in such a way that is engaging to them. There are currently a small number of tools that really support end user programming on the web, and even few for creating mashups. Tools like Yahoo! Pipes and Microsoft Popfly use visual programming languages to alleviate some of the initial learning curve of code. Using direct manipulation seems to be an accepted method, but does adding these extra layers of abstraction truly help the end user. In particular, as our work has found, many times hiding the underlying "code" may cause even more confusion. From our work we can easily conclude that Pipes is not a tool for non-programmers. The way in which the tool is organized and the modules named suggests that it was mostly built simplify data aggregation for developers. For a less programming savvy user to take advantage of Pipes he/she must be able to specify actions in a programmatic way. This may not be possible for this group of end users; there is a underlying difference in the way in which programmers solve problems.

From our work with mashups, we realize that the basic concepts of this technology are not necessarily a new idea, but the way in which developers integrate APIs and create visualization is quite novel. As a result, even seasoned programmers must learn new techniques. To a greater extent, end users must learn as well. The key difference here is that programmers have already developed a baseline of what is commonly called computational thinking [7]. It is this mindset that allows programmers to solve problems using programming languages. Relating to our work, programmers can decompose problems in a way that acceptable by computer systems. While most EUP research focuses on lowering the skill boundary for end users by developing easier to use tools or creating programming languages that better match the mental models of the developer, it may be more important to find ways to raise the basic level of computational thinking for end users.

Generally computational thinking is a term reserved for educational domains, but I believe it would be useful when thinking about end-user programming. End users must learn additional skills when approaching new tools, regardless of the activity. If this is the case, why not provide additional support that teaches skills they can use in other situations? However, a significant problem is that users must be motivated by their task enough to learn something new. Our work has pointed to possible ways of engaging these users by leveraging their interests and taking into consideration existing activities. By continuing to build extremely simple tools we perpetuate the expectation that everything must be "easy". In the long run, it may be more beneficial to encourage users to increase their computational skills.

8. Conclusion and Future Work

This work has highlighted some key areas that EUP research is taking when working towards addressing end users problems. Great strides have been made to better understand end user programming activities, and many tools have been created to support the specific tasks of the end user that leverages advanced and novel technologies. However, much of the recent work in this area focuses on the building new tools, but not realizing the needs of the end user. Rather than concentrate on the tools and technologies surround the web, we should seek better ways of engaging the user. It does not matter how novel a tool may be, if there are no users then there is no clear impact.

While usability is a concern, this work has also found that there is very little emphasis on educating users as they continue to explore the web. The EUP tools that have been built still require a programming or computational mindset. The way in which the driving technologies operate will continue to be based upon programming concepts. By supporting and even encouraging users to take an additional step and pursue a better understanding of these technologies would surely help alleviate some of the skill barriers to programming.

Our next step is to broaden our prospective to look at the process that users go through to gather information, integrate

it into a meaningful form. A survey study is currently being conducted to investigate how web-active users mentally decompose their activities online. This is akin to creating step-by-step instructions. This survey should allow us to better understand how users think about the information they are working with and how they naturally think about their information processes on the web. We hope to follow up the survey with a paper prototyping task where the participants will work with snippets of information and perform analysis on them in order to accomplish a task. This method may allow us to actually investigate their information integration processes without the complexity of computer interfaces.

Acknowledgments

This research was partially supported by The National Science Foundation (CCF-0405612).

References

[1] E. Schonfeld. YouTubes Chad Hurley: We Have The Largest Library of HD Video On The Internet. *TechCrunch*, January 30, 2009.

[2] B. Nardi. *A Small Matter of Programming: Perspectives on End User Computing* MIT Press, 1993.

[3] C. Scaffidi, M. Shaw, and B. Myers. An Approach for Categorizing End User Programmers to Guide Software Engineering Research. In *ICSE'05 Workshop on End-User Software Engineering* ACM Press, 2005.

[4] J. Wong, and J. I. Hong. Making Mashups with Marmite: To-wards End-User Programming for the Web. In *CHI'07* ACM Press, April 2007.

[5] Yahoo! Pipes. http://pipes.yahoo.com.

[6] J. F. Pane, and B. A. Myers. Usability Issues in the Design of Novice Programming Systems. CMU-HCII-96-101, Carnegie Mellon University, Pittsburgh, PA, 1996.

[7] J. M. Wing. Computational Thinking. In *Communications of the ACM*. volume 49, issue 3. ACM Press, 2006.

[8] A. Cypher. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[9] ProgrammabeWeb. http://www.programmableweb.com. Retrieved Oct 8, 2009.

[10] N. Zang, M. B. Rosson, and V. Nasser. Mashups: who? what? why? In *CHI '08 Extended Abstracts*. ACM Press, April 1, 2008.

[11] N. Zang, and M. B. Rosson. Whats in a mashup? And why? Studying the perceptions of web-active end users. In *IEEE Symposium on Visual Languages and Human-Centric Computing 2008*. September 15, 2008.

[12] N. Zang, and M. B. Rosson. Playing with Information: How End Users Think About and Integrate Dynamic Data. In *IEEE Symposium on Visual Languages and Human-Centric Computing 2009*. September 20, 2009.