Debugging Revisited

Toward Understanding the Debugging Needs of Contemporary Software Developers

Lucas Layman, Madeline Diep
Fraunhofer Center for Experimental Software Eng.
College Park, Maryland, USA
llayman@fc-md.umd.edu
mdiep@fc-md.umd.edu

Meiyappan Nagappan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University
Kingston, Ontario, Canada
mei@cs.queensu.ca

Janice Singer
National Research Council
Ottawa, Ontario, Canada
janice.singer@nrc-cnrc.gc.ca

Robert DeLine, Gina Venolia
Microsoft Research
Redmond, WA, USA
rdeline@microsoft.com
ginav@microsoft.com

***Abstract*—We know surprisingly little about how professional developers define debugging and the challenges they face in industrial environments. To begin exploring professional debugging challenges and needs, we conducted and analyzed interviews with 15 professional software engineers at Microsoft. The goals of this study are: 1) to understand how professional developers currently use information and tools to debug; 2) to identify new challenges in debugging in contemporary software development domains (web services, multithreaded/multicore programming); and 3) to identify the improvements in debugging support desired by these professionals that are needed from research. The interviews were coded to identify the most common information resources, techniques, challenges, and needs for debugging as articulated by the developers. The study reveals several debugging challenges faced by professionals, including: 1) the interaction of *hypothesis instrumentation* and software environment as a source of debugging difficulty; 2) the impact of log file information on accurate debugging of web services; and 3) the mismatch between the sequential human thought process and the non-sequential execution of multithreaded environments as source of difficulty. The interviewees also describe desired improvements to tools to support debugging, many of which have been discussed in research but not transitioned to practice.**

***Index Terms*—debugging, software engineering, interview, professionals, qualitative analysis, program comprehension.**

I. Introduction

The standard definition of debugging is: "To detect, locate, and correct faults in a computer program" [1]. In practice, each step of "detecting, locating, and correcting" faults is a complex task requiring problem solving, expertise, and tool support. Even though there is a research corpus on debugging, we know surprisingly little about the challenges developers face in industrial environments. At an ICSE 2011 panel on "What industry wants from research", industry representatives urged researchers to focus on problems that are important to practitioners.

This paper seeks to fill part of that gap by conducting exploratory research with professional developers to: 1) understand how professionals use information and tools to approach debugging; 2) identify new challenges in debugging in contemporary software development domains, such as web services and multithreaded/multicore systems; and 3) identify how they articulate their desires for improved debugging tool and process support.

We present the qualitative analysis of interviews with 15 professional software developers at Microsoft. Our findings are exploratory, yet reveal some interesting new issues on debugging in a contemporary professional environment, including the disruption caused by the nature of web service and multithreaded applications to the debugging process. We also confirm popular notions and prior research of what debugging is and how it might be improved. We offer seven observations on our analysis findings, including description and discussion of the challenges articulated by the developers.

The remainder of this paper is organized as follows: Section II provides a review of background literature on debugging and its relationship to theories of program comprehension and maintenance; Section III describes our research method; Section IV describes our subject demographics and the bug examples they provided; Section V provides analysis of resources, methods, and practices used in debugging; Section VI describes debugging challenges articulated by the subjects; Section VII presents the desired advances in debugging proposed by the subjects; Section VIII discusses our limitations; and Section IX summarizes the work.

II. Background and Motivation

Studies of debugging and program comprehension have a rich history. Many studies focus on how developers collect information used in maintenance tasks. For example, Robillard, et al. [2] and Sillito et al. [3] studied the strategies used and questions asked by developers performing software maintenance tasks that involved multiple debugging steps. Ko, et al. [4] reported a fine-grained, controlled study of the source code relationships and IDE components used by developers performing a software maintenance tasks. Roehm, et al. [5], performed an observational study of professional developers to determine how they comprehend programs – a component of the debugging task.. While these studies are examples of research that focus on the detailed cognitive aspects of individuals' debugging and program comprehension processes, our study provides developer's *own articulations* of debugging resources, methods, and challenges.

Our study elaborates on general models of debugging behavior. Kessler and Anderson [6] and Katz and Anderson [7] provided one of the first models of debugging behavior. In two controlled student experiments, they observed a four-step model of debugging behavior: 1) comprehend the system; 2) test the system; 3) locate the error; and 4) fix the error. Gilmore [8] presented a study conducted with 80 subjects who debugged 10 versions of two programs with two bugs each. His results showed that the key aspect of program comprehension during debugging is to understand the relationship between the problem statement and the changes that solved the bug. Von Mayrhauser and Vans [9] presented a model for program comprehension during debugging based on an observational study of four developers. They captured the actions developers performed, the process followed, and their information needs. They found that the most frequent action was using prior knowledge to generate hypotheses, and that the domain knowledge of the developer drove information need.

These prior studies are valuable for understanding the mechanics of the debugging process. We offer a complementary perspective: we identify how professional developers define debugging and its challenges. This perspective helps to paint a more complete picture of debugging in the context of new domains (e.g. cloud computing systems, massively parallel back end servers, UI rich front end clients, etc.) and the technologies used to build, comprehend, and debug software in industry.

III. Research Method

We interviewed 15 developers at the Microsoft campus in Redmond, WA, USA to understand debugging from a professional developer's perspective. We then performed qualitative coding analysis on the interview transcriptions. This section describes the interview and qualitative analysis method.

A. Interview method

Study participants were solicited through a company-wide email distribution list. Interested participants were asked to provide a short definition of debugging and demographic information related to experience and expertise. The 15 interviewees were selected from the survey respondents to obtain reflections from a range of experience and problem domains (e.g. web services, operating systems, programming languages, gaming). Due to time limitations, fifteen participants were chosen for a one-hour interview from several dozen respondents. The subject sample was constructed to provide representation of diverse expertise and application domains. Participants were not compensated.

Interviews were conducted in the subjects' offices by two researchers – one focused on the questions and one taking notes. The interviews lasted approximately one hour and were audio recorded. The interviews were semi-structured and the questions focused on four areas: 1) demographics; 2) debugging; 3) bug reproduction; and 4) test-driven development. Portions of the interview related to test-driven development are ignored for this paper. The full set of interview questions may be found at: http://goo.gl/FBfLW. For purposes of this study, the most relevant interview questions on debugging and bug reproduction are (paraphrased):

- "In two sentences, how would you define debugging?"
- "Think of a recent debugging session and describe to me the bug and how you went about fixing it."
- "What tools did you use to solve the bug?"
- "Which information sources did you consult?"
- "What were the biggest challenges in solving the bug? In debugging overall?"
- "What additional information and/or tools would be helpful for debugging and bug reproduction?"

We asked developers to recount specific bugs ground the developers' thoughts on debugging in specific examples rather than solely on generalities. Further, the concrete examples give us confidence that the challenges and needs faced by these developers are real rather than perceived.

B. Analysis method

Our goals for analysis were twofold: 1) to identify the *most common responses* from the interviewees to the questions in the previous section 1; 2) to identify *themes* in the responses to those questions. Our first step was to *code* the interview transcripts. Coding is a method for identifying topics and themes in qualitative data [10] that also prepares qualitative software engineering data for quantitative analysis [11].

The audio recordings of the interviews were transcribed by a professional service familiar with software development terminology. All 15 interview transcripts were coded in group meetings by the first three authors. Coding analysis was completed over 15 sessions of 2-4 hours each for a total of approximately 114 person hours of effort. Coding and analysis was performed using Weft QDA (http://www.pressure.to/qda/) and FreeMind (http://freemind.sourceforge.net/). The authors made frequent revisions to the coding scheme as it changed throughout analysis.

We performed *open coding* [10], wherein subjects responses to the interview questions are categorized according to topic. The topics were grouped into seven main categories according to the research question:

- Demographics
- Definition of debugging
- Descriptions of actual bugs
- Information used in the process of debugging
- Method, processes and tools used in debugging
- Challenges in debugging
- Needs that improve the state of the practice in debugging

The coding scheme is hierarchical (Fig 1). Codes at the same depth in the hierarchy within the same branch represent distinct concepts (e.g., the nodes below the "current environment" subcategory in Fig. 1 are distinct concepts). A subject's response to a question could be and often was coded into multiple categories.

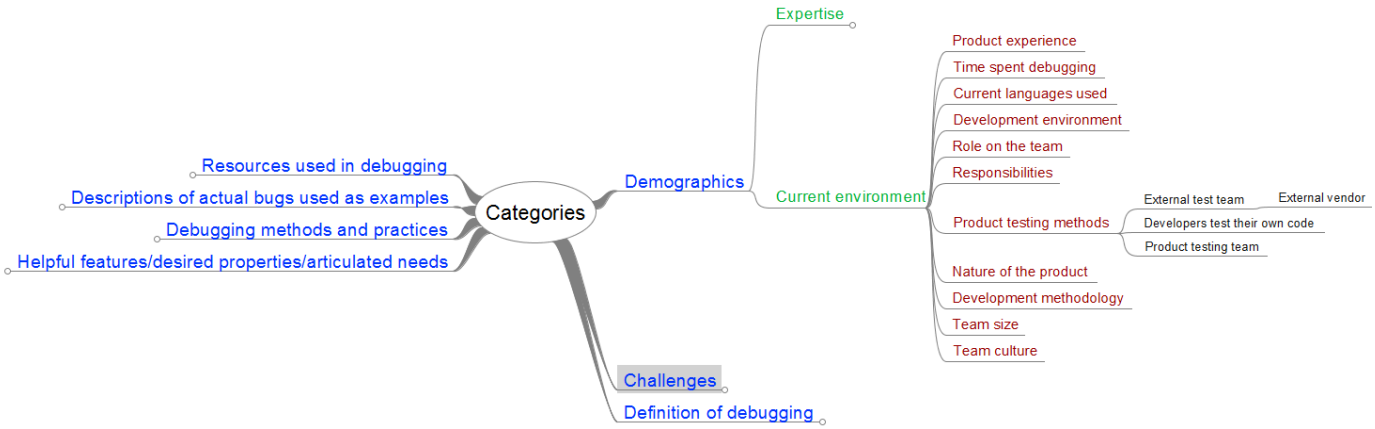

Fig. 1. Coding scheme excerpt demonstrating the coding hierarchy. Main categories are blue and subcategories are green.

After coding three transcripts, the initial coding scheme had seven main categories (e.g., demographics, challenges) with 22 sub-categories (e.g., environmental challenges, expertise challenges) and 66 unique codes. After the 5th transcript, the sub-categories in the Challenges and Helpful Features categories were reorganized to be more self-descriptive. The revised coding scheme contained seven main categories, 45 subcategories, and 107 unique codes. The categories and subcategories remained stable for the next 10 transcripts while new instances of specific tools, challenges, and needs were added. After coding the final transcript, we consolidated a number of related codes in the Challenges category according to common themes. The final coding scheme contains seven main categories, 55 subcategories, and 195 unique codes.

In our analysis, we provide counts of the number of subjects whose responses were coded into the corresponding category. We count only the number of individuals – not the number of times a code was mentioned. The counts are the number of individuals whose responses were coded to that category and any of its descendants. Throughout this paper, we use numbers in parentheses to indicate the number of subjects who were coded to a category, e.g., "internet resources (9)".

IV. Subject Demographics and Bug Descriptions

The subject sample and the bugs they described to anchor the interview are described in this section.

A. Subject demographics

The 15 interviewees were selected to provide a diversity of expertise and application domain so that a diverse set of debugging experiences could be obtained in the interviews.

Years of professional development experience – The experience of the subjects ranges from less than one year to 30 years. Note that we only counted the number of years that subjects worked in professional setting as software developers. Figure 2 shows the distribution of the subjects' experiences.

Time spent debugging – The time spent debugging varies depending on the phase in the development cycle. Some subjects described a phase in the development cycle dedicated mostly to debugging where they spent 90-100% of their time debugging. In other phases, subjects reported spending 4% to 50% of their time debugging. Subjects also report that more time is spent on debugging unfamiliar code, such as legacy code.

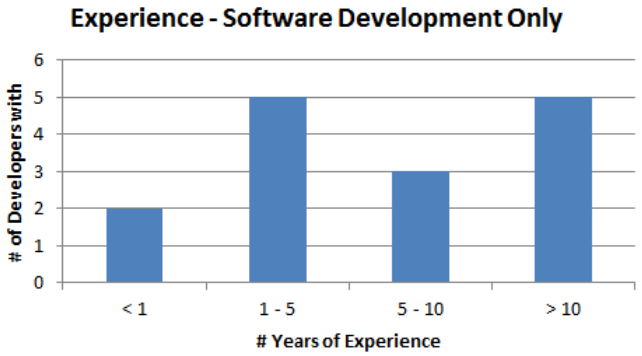

Fig. 2. Years of professional software development experience

Current languages used – All of the subjects use high-level programming languages, including C, C++, and C#. A small number of the subjects work with scripting languages (Python and JavaScript), markup and style sheet languages, and the organization's proprietary programming language.

Development environment – Only nine subjects provided information about the development environment they use. Five of the subjects used the Visual Studio IDE, while the rest of the subjects utilize a set of tools, such as Source Insight for editing, browsing, and viewing the code. The IDEs often provide powerful debugging tools, such as the ability to step breakpoints, to step through code, to add conditional breakpoints, and to step through failure stack traces all while executing the program.

Role on team – Eleven out of the twelve subjects providing information about their role on the team were developers, one was a tester, and three did not provide this information. Among the developers, four assumed additional roles, such as tester, development lead, or support.

Product testing methods – Eleven of the subjects provided information about the product testing methods. Seven of the subjects unit test their products. Seven of the subjects have a

separate test team, and one of the test teams was external to the development organization.

Nature of the product – Subjects develop diverse types of products, including the organization's internal products (such as testing infrastructure and tool support for proprietary programming languages), products supporting the organization's external products (such as libraries and tools for developing device-specific applications, service deployment, UI framework and rendering, and shared navigation control across the web services), and external products for paying customers (such as web services, cloud storage technology, crawling and indexing components of image/video search services, client-side support for data centers, etc.)

Development method – Almost all subjects use some type of iterative development life cycle with release cycles ranging from 2 weeks to six months. However, they do not use a specific methodology in rigorous manner. Several of the subjects use Agile practices/methodologies, such as test-driven development and Scrum, while a small number of subjects follow an iterative waterfall-like model.

B. Bug descriptions

Subjects were asked to describe a recent bug (or bugs) and how they went about fixing it. The root causes of the bugs were categorized as part of the open coding analysis. Table I presents the root cause categories and the number of interviewees whose example bug fit each category. A subject may be counted in multiple categories if he/she described multiple bugs of different types. One subject did not describe a specific bug.

TABLE I. ROOT CAUSES OF SUBJECT'S EXAMPLE BUGS

# subjects	Bug category
6	Logic error
3	Specification-related error
2	Bug in external component
2	Multithreading error
1	Device configuration error
1	Find-and-replace error
1	Incorrect API used
1	Localization error
1	Uninitialized variables
1	Unknown

Logic errors were the most common type of bug described, wherein the program logic was computing incorrect values or otherwise resulting in unexpected behavior. Specification-related errors were attributed by the subject to an incorrect requirement specification. A bug in an external component was a bug that the subject could not fix at the root because he/she did not own the defective component. Instead, the developer had to create a workaround or implement additional fault tolerance. Finally, multithreading errors are bugs attributed to concurrency issues, such as race conditions, starvation, and shared resource corruption.

V. DEVELOPER DEFINITIONS OF DEBUGGING

Our first research question is: "how do professional developers *define* debugging?" Table II presents the coded responses of the subjects; each subject may appear in more than one category.

TABLE II. SUBJECTS' DEFINITIONS OF DEBUGGING

# subjects	Debugging definition
13	Finding and solving reported defects
6	Verifying that code is running correctly
5	Program comprehension
1	Customer troubleshooting

The distinction between "finding defects" and "verifying correctness" is valuable. In the former case, "if something is wrong… you figure out why, what's the cause, how to fix it, and how to verify that you fixed it" (Subject 11). By this definition, debugging is a *reactive* process wherein a failure (or other error) has been observed and the task is to fix the error. In the latter case, "debugging is the process of exercising your code with an objective of finding or preferably not finding any bugs, and once that happens then you have the option of addressing those bugs" (Subject 10). By this definition, debugging is a *proactive* process where developers verify their implementation to prevent failures.

In addition to notions of fixing and preventing faults, five developers explicitly mentioned program comprehension as an element of the debugging process. Debugging is also a process for "rounding out a mental map of the software" (Subject 2) and "observing the flow of the code and the operations to try to get a better understanding for what's going on" (Subject 5). As articulated in [9], an important part of hypothesis building is to understand the actual and intended behavior of the system.

The outlier in Table II is noteworthy. This subject had an expanded take on what it means to debug in a web services environment: "If you suddenly have a problem [in a service], it can be for reasons that are outside of the code […] a network switch might be down, or the latency has increased ... So all of these issues that are actually troubleshooting the system, I would include those in debugging. It's the same kind of skill set that you need to solve it, but it doesn't necessarily imply looking at the code" (Subject 7). This subject indicates a trend that we will visit in more detail later: distributed and web service systems introduce new challenges in debugging.

VI. INFORMATION AND METHOD USED IN THE DEBUGGING PROCESS

The developer definitions of debugging indicate that the debugging process can be both proactive and reactive, involves a strong element of program comprehension, and expands to resources beyond the source code. Next, we want to understand the specifics of how the debugging process (in all its forms and applications) is carried out. The interviews were coded to answer the following questions:

1. What *information* is used in the process of debugging?
2. What *methods or techniques* are used in the process of debugging?

To frame our discussion, we provide a model of the debugging process (Fig. 3). The model was created post-analysis and bears elements of previous models of debugging and program comprehension [8], [9]. We provide it here solely

to organize and describe our findings – this model has not been validated beyond its initial derivation from the data.

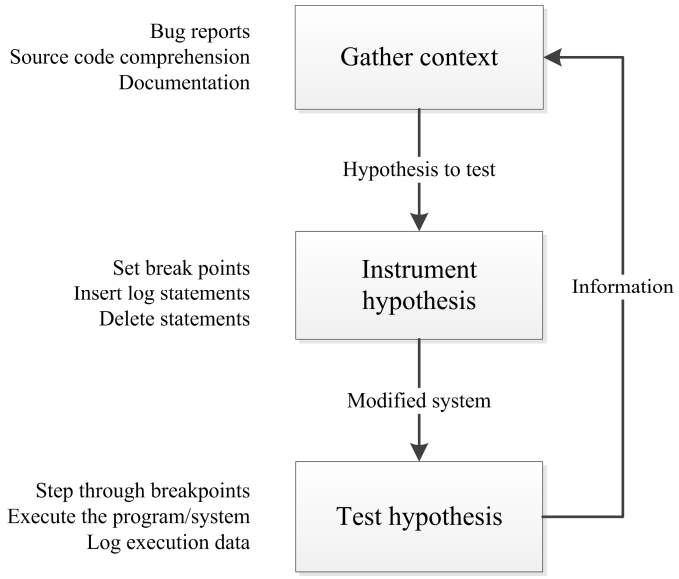

Fig. 3. Debugging process model

The debugging process is one of iterative hypothesis refinement. In our sample, nine subjects described debugging as forming a hypothesis and then testing it: "Generally, the way I go is that I will build a hypothesis. I mean just looking at the code: potentially what could be going wrong. And once you have a hypothesis then I'll just try to figure it out if it's right or wrong" (Subject 8). Eight subjects provided examples of how they refined their hypotheses through a divide-and-conquer approach to disproving scenarios: "You eliminate things. And so it's kind of like detective work" (Subject 12) and "most of the time you need to come up with a theory of what might be wrong and try to isolate and figure what parts in place are interfering with each other" (Subject 14).

The debugging process model in Fig. 3 is comprised of three, iterative steps: 1) gather context to generate a hypothesis; 2) instrument the hypothesis; 3) test the hypothesis. This model is similar to previous hypothesis-generation then hypothesis-test models of debugging and program comprehension [8], [9]. Note that the actions of "fixing the bug" or "changing the code" are generalized under "instrument hypothesis". This model leads us to our first observation, which we elaborate on in the remainder of the section:

Observation: The distinction between *instrumenting* and testing hypotheses is critical to understanding several challenges faced in debugging distributed, parallel, and web- or cloud-based systems.

A. Step 1: Gather the context to generate a hypothesis

Reproduction steps and failure information are critical to understanding the system state before, during, and after the point of failure. This understanding, in turn, is used to help identify potential causes of the failure. Additionally, debugger tools provide runtime insight into the system to help the developer form a mental model of the program. This mental model of the program and potential changes to the system are augmented by additional external resources. Taken together, these *context gathering activities* provide information that forms the basis of hypothesis generation regarding the origin of bugs in a program or the impact of change made to a program.

All 15 subjects recounted a variety of information used in the debugging process (Table III). We provide synopses highlighting some of the important observations in the subject data below.

TABLE III. RESOURCES USED IN DEBUGGING

# subjects	Resources used in debugging
15	Debugger tools
14	Bug information
12	Communication with others
9	Internet resources
7	Custom code/manual debugging data
6	System state information (variables, packets)
5	Searching the source repository
4	Code browsers
3	Printed publications
2	Production health/status/monitoring systems
2	Build information
1	Personal library of technical tidbits
1	Shared internal development team resources
1	Product documentation

1) Debugger tools are domain dependent

The subjects described tools that vary according to the subject's application domain: traditional debuggers for source code, browser and JavaScript testing tools for web development, packet sniffers for networking, and hardware and operating system profilers for device driver development. These tools provide information on the current system state. In all, the 15 subjects mentioned 16 distinct debugger tools. There is no one-size-fits-all tool for debugging, but rather a toolbox.

2) Debugging requires sources of information beyond the code

Whether finding the source of a bug or evaluating a potential change to a program, the answers come from many sources. There is often not enough information about a bug or desired fix, necessitating communication with team members, be it face-to-face (10) or via email (5), between external specialists (5) or customers (5). The need to communicate with other developers to properly reproduce a bug has been documented in prior research [12]. The Internet is often consulted to assist in problem solving, such as API documentation (7), search engines (5), and online discussion groups (2).

3) Reproduction steps and system state are essential to accurate fault diagnosis

Specific sub-categories of bug information used in the debugging process are presented in Table IV. Accurately reproducing a fault is essential to identifying the root cause and potential repair action. Twelve subjects expressly mentioned reproduction steps provided by a tester or other bug reporter,

which sometimes included screenshots (2) and video (2). Log files (10) play a key role in diagnosing bugs, including the specific strategies for inserting log statements and collating results to provide meaningful, non-superfluous information (4). Log files were mentioned more often than stack traces and crash dumps (7) as a source of fault information.

TABLE IV. SOURCES OF BUG INFORMATION

# subjects	Types of bug information
12	Reproduction steps
10	Log files
7	Stack trace / crash dump
6	Test results
3	Bug database
2	Observed system behavior
2	Heap snapshot
1	Other customer complaint

B. Step 2: Instrument the hypothesis

Once developers have gathered context and formed a hypothesis regarding the cause of a bug or the impact of a change, the next step is to test that hypothesis. Our model of the debugging process (Fig. 3) adds a step before hypothesis testing: instrumenting the hypothesis. To instrument a hypothesis is to *modify the program source or execution to gather information to test the hypothesis.* Examples of instrumenting hypotheses include setting breakpoints, inserting log statements, and modifying source statements. By examining how developers instrument their hypotheses, we can better answer the question "what *methods or techniques* are used by professional developers in the process of debugging?" Table V presents the coding analysis results.

TABLE V. HYPOTHESIS INSTRUMENTATION METHODS

# subjects	Hypothesis instrumentation methods
7	Inserting breakpoints and watch variables
4	Inserting log statements
2	Removing irrelevant code
2	Tweaking - modifying existing code

1) Gradations of disruption in instrumentation

We assert that all forms of hypothesis instrumentation are disruptive to the original execution of the program, save for executing the program in its production environment. Executing the program in a development environment with a debugger attached (7) introduces performance overhead and environment dependencies that impede simulation of actual program execution. Inserting log statements (4) or other print statements are relatively benign but alter the control flow of the program and add additional dependencies that may influence hypothesis testing. Most interesting are deletion or modification of the source code to help test hypotheses. For example, "Commenting out code to try to narrow in on what is causing the problem" (Subject 13) and "make a change to the code, try rerunning it, and see what happens… just make a change to make sure that it [meets] my expectations" (Subject 9). This approach is disruptive to the original control flow of the program by creating the greatest potential for unintended side effects that may make hypothesis testing inaccurate.

Observation: Modifying the source code is beneficial to isolating the cause of bugs, but may cause additional, unwanted behavior changes.

2) Debugging over logging – when possible

In general, there appears to be an order of preference for instrumenting the system for hypothesis testing. A debugger environment (e.g. the Visual Studio debugger) offers rich information, such as the call stack (4), exception code (4), runtime values of variables (4), and the ability to pause execution (1). In contrast, log files and log statements (4) can provide information on program state but cannot be manipulated at runtime to provide additional information. Logging provides imperfect information, but is the "first line of defense" (Subject 5) for production systems: "so eventually I iterate, and you iterate adding logging, looking at the logs, and just trying to understand how the code worked" (Subject 2).

Observation: Context information and control of execution are the perceived advantages of debuggers over logging.

C. Step 3: Test the hypothesis

The final step of the debugging process is to test a hypothesis by executing the program with instrumentation to observe its behavior. Table VI provides the code categories for hypothesis testing methods.

TABLE VI. HYPOTHESIS TESTING AND COMPARISON METHODS

# subjects	Hypothesis testing and comparison methods
7	Stepping in the debugger
4	Comparing against examples
2	Comparing against an oracle
1	Analyzing network packets
1	Backtracking
1	Printing out hard copies of code

Testing a hypothesis is not always as simple as stepping through a debugger. Some environments, such as the web or distributed systems, cannot be easily debugged using a traditional debugger. Distributed cloud services can only be debugged using logged information and performance monitors. Hypothesis testing involves comparison of the actual behavior of a system against some control: either a mental model of the system, example expected output, or a testing oracle.

1) Web and cloud service environments introduce lag time in hypothesis testing

The predominant environment for debugging is on the desktop using an IDE debugger. This provides instant feedback to the developer and facilitates rapid hypothesis iteration. Some domains do not have the benefit of this rapid feedback, such as cloud and web services. These uncontrolled operational environments also mean the desired parts of the system may not be exercised. This can cause the debugging process to take

"several weeks" as many generations of log statements are created, deployed, and the results collected (Subject 2).

Observation: Uncontrolled environments (such as the web) make traditional debugging impossible because of the lag time between instrumenting and testing a hypothesis.

2) Some environments limit the debugging resources available for hypothesis testing

Not all developers can make use of a debugger: "Because of the world that we live in and the Web, we do not have the ability to actually debug in a production environment, I mean, using an actual debugger" (Subject 4). Another subject had a similar comment, "you can never really attach a debugger to a front-end, the machines responsible for rendering the UI. If you attach a debugger on somebody's […] request as it goes to that machine, it's not going to give accurate results, and they're not going to be very happy" (Subject 3). The distinction between instrumenting and testing a hypothesis in our debugging model (Fig. 3) was prompted in large part by the lag time and lack of control in some software development environments.

Observation: In general, developers are limited to using debuggers (and the rich data contained therein) when their testing or productions environments permit.

VII. Debugging challenges

Our observations in the previous section alluded to challenge areas in need of improvement to support debugging in a modern software development setting. All fifteen subjects were asked to describe the biggest challenges in solving their bugs and in the debugging process overall. The responses to this question were used to answer the question "what *challenges* are faced by professionals when debugging?" The subject responses fell into the six categories listed in Table VII.

The subjects described a diverse set of 46 distinct challenges. Communication challenges include, for example, difficulty gaining access to expert knowledge (2) or design rationale (1), and asking the right questions of both experts (1) and search engines (1). Debugging process challenges reflect the inherent cognitive difficulties of debugging, such as losing one's frame of thought (2) and the time-consuming venture of navigating source code (1). Six subjects had the common complaint of not being able to accurately reproduce a reported failure. Below, we provide themes from the more interesting challenges from the highlighted categories in Table VII.

TABLE VII. Debugging challenges

# subjects	Debugging challenges
11	Environmental challenges
7	Multithreaded/multicore
6	Information quality
6	Communication challenges
6	Unable to reproduce failures consistently
4	Debugging process challenges

A. Environmental challenges – tool usability and domain support

Nine respondents cited of shortcomings in the debugger. Subjects complained of debuggers having a steep learning curve and non-intuitive UIs (3). Others (2) gave examples of the debugger providing incorrect debugging symbols: "as long as you have the debugging symbols for the modules you can just click on a frame in the call stack and here comes the source code and it's pointing at the line that you're at ... if you don't have the right symbols or something and it doesn't work, all of a sudden it throws me for a loop" (Subject 12).

Other subjects complained of *debugger performance in specific domains*, such as poor performance when using remote debugging (1). Two subjects described the challenge of tracking data between threads in a multi-threaded application and identifying the correct thread to attach the debugger: "In the Web world, one of the things that we face is when I try to attach a debugger to a process that's running inside [a webserver] … there can be dozens of them going, and in that case, I literally have to sit there and go through each one and attach it until I find the right one" (Subject 5). Two other subjects described the difficulties of debugging code that has been optimized by the compiler: "With release builds, since there are so many optimizations, I mean source level debugging is really kind of bad. For example, you'll be jumping up and down, and it makes no sense" (Subject 18).

B. Multithreaded and multicore challenges – Heisenberg's multithreading debugging principle

"You know you're stepping through [a thread]… but this other thing is still running, or is it not running?" (Subject 13). The human problem-solving process is in many ways sequential. If A, then B. If A, then not B, something is amiss. "It's sort of human to imagine things as a sequential workflow. So, we make assumptions that once you start doing something it will end that way, and in a multithread environment that's not the case" (Subject 14). However, when debugging multithreaded applications, developers are forced to project a sequential analysis onto a non-sequential program execution: "Sometimes you have to freeze the other thread … But then, oh, wait, that deadlocks the thread that you're trying to work in. And you end up having to … thaw him to let this other guy go, then you get a break[point] on that guy… it consumes a lot of time because you actually have to go figure out what these threads are and how they're dependent" (Subject 13). Race conditions between threads, in particular, were identified as the most difficult bugs to track down: "The biggest challenged was a race condition involving like 40 threads or something. It was difficult to track it down because you had to kind of freeze frame the threads and then let them go selectively" (Subject 8).

The sequential human thought process versus the non-sequential execution of multi-threaded programs is, we believe, the root cause of the challenges of debugging multithreaded programs. The analysis enabled by IDE debuggers requires human interpretation, which in turn necessitates a sequential view of the application under test. By casting the multithreaded

program sequentially, developers introduce highly disruptive hypothesis instrumentation (Section VI.B.1).

Observation: Instrumenting hypotheses to debug multithreaded applications drastically changes the runtime environment. This change decreases the likelihood that the original fault will manifest and increases the likelihood that the true fault will be masked by other behavior, such as a deadlock caused by breakpoint.

C. Information quality – piecing together log data

Six subjects described current challenges in working with log files. Application logs provide (useful) information to help debug failures in production. Log files often record *events* (i.e. actions) that occurred in the production system, sometimes including specific failure information such as stack traces or exceptions. Two subjects noted that event-based logs provide *insufficient information* to diagnose the failure: "There've been cases in the past where we've had a series of crash dumps which just tell us this line of code is crashing; and we have no reason to know why it is because we can't see the local context information. If we had that variable active, we would've solved those bugs very efficiently" (Subject 3).

Three subjects commented that *collating and analyzing log information* presents a challenge. These subjects work in an online services environment where logs from production systems were often their only insight into failures: "If there's no log, I don't even know [a failure] is happening … because I have no way to access the machine" (Subject 11). These production systems generate a "huge amount of data" (Subject 7), and this huge amount of data is sometimes "really hard to put together" (Subject 11). In a web or cloud environment, a single event initiated by the user (such as purchasing an item from an online store) may be serviced by multiple instances of the software on multiple machines. If the user-initiated event fails, the challenge is to piece together what happened from multiple logs across multiple machines with potentially different configurations. For example, "when [the client request] goes from the [front end] to the middle tier, it can go again to any of the 20 boxes… somebody reports a problem, you open the boxes, and do a search on all of them and see where it is … When there is an error happening on this application that spans across machines … you want to be able to get the logs from all machines" (Subject 7). This challenge is akin to understanding the *provenance* [13] of a transaction across multiple machines to help developers debug failures that occur in the transaction.

VIII. Desired improvements to support the debugging process

The subjects provided diverse answers when asked "what *additional information and/or tools* would be helpful for debugging and bug reproduction?" The responses were coded to 33 categories. Table VIII lists the topics indicated by multiple subjects. We provide discussion of the more interesting and novel responses below.

TABLE VIII. Debugging challenges

# subjects	Debugging challenges
6	Capture and replay of production events
3	More contextual information in runtime logs/stack traces
3	Integrating data from different sources
3	Bi-directional debugger
3	Debugging tool training
3	Multithreaded support
2	Automatic breakpoints upon entry into a class
2	Automated log analysis
2	Program context
2	Visually showing the execution trace

A. Improving failure reproduction

Six subjects expressed a desire for better support to capture and replay events that occurred in a production system around the time of a failure (green highlight in Table VIII). "If we can replay the input going into that code around when the crash happened, that would be really, really good" (Subject 4). The subjects' desires for replay focus on recreating system state from the system input around the time of the failure. For example, "[our program] was being pelted incessantly by what seemingly is randomly generated operations … a log of what those things are would have been so valuable" (Subject 12) and "What is the network traffic? How to playback that traffic?" (Subject 11). While tools exist to support capture and replay of system state for debugging purposes [14–17], the application of such tools in real production systems with respect to scalability and security concerns is unclear.

B. Better support for log file analysis

Five subjects (the union of the pink highlights in Table VIII) expressed a need for better information captured in log files. One challenge is obtaining more contextual information about the program state from log files: "In debugging from a production crash, I think it would help a lot if we had the reliable local variables and reliable heap information ... because sometimes the logs aren't completely aligned with the crash dumps, it's hard to figure out what the program was actually doing" (Subject 3). "We have very good debugging tools, to be honest. One [improvement] could be a standardized way of putting some more source level debugging information in the logs" (Subject 8). The practical limitations of adding more information to log files are obvious, "in an ideal world, you'd have the stack trace for every line that was written to the log because you never know what you're going to need. But, of course, there's going to be a lot of overhead" (Subject 15).

The subjects also expressed a need for tools to integrate information from multiple log files. For example, "[it would be nice] to bring in the window from around each crash dump and do a comparison of the log messages around each crash to see whether there are patterns" (Subject 3) or to match all log files "from a specific point in time" (Subject 3). Further, log file information could be integrated with other information for even greater benefit: "if I can create the linkage between different pieces: source code, deployment logs, and all those with something related … then form one I can easily navigate to another one without receiving too much noise. That would be

really helpful" (Subject 11). These proposed improvements reduce the search hypothesis space by removing noise and providing relevant information. Similar notions of reducing the navigation space for improved debugging performance have been observed in (e.g., [2]).

C. Multithreaded support

Many of the suggestions for multithreaded debugging support involve more information and control in the debugger (blue highlights in Table VIII). One subject offered a basic usability improvement: "just being able to say *freeze all the other threads* other than the one I'm currently working on, *thawing the other* ones, and stuff like that" (Subject 13). Another beneficial ability in the debugger would be to determine *which thread owns a lock*: "If I'm waiting on a lock, being able to see who actually currently has that lock. It would be nice to be able to see that easily" (Subject 13). For example, "I'd sometimes like breakpoints to be *thread-specific*. [If] there's a bunch of worker threads that are running the same code, I don't want the other guys hitting that breakpoint and confusing me" (Subject 13).

One subject's program used a queue where threads deposited data into a queue, and then separate threads pulled the data from the queue for processing. He expressed his desire to *trace data across threads*: "oftentimes the thing that I was trying to follow was a particular message or piece of data and it was crossing a bunch of threads" (Subject 15). The subject suggests data tagging where the debugger would notify the user when particular data are passed through the system.

These suggestions all focus on supporting the sequential analysis of multithreaded applications, which we discussed previously in Section VII.B. Only one subject mentioned the need for better automatic multithreaded fault detections tools, such as tools that examine for deadlocks or race conditions (Subject 13). Though numerous such tools exist, it is unclear how they are used by development teams and if it would be beneficial or how to integrate such tools into the development environment alongside the debugger.

D. Backward debugger

Three subjects mentioned the concept of a *backward debugger*: one that allows the user to step backward in execution. For example, "let's say you have a crash … I would like to be able to stop there and backtrack logic. So, basically review" (Subject 4) and "If you're trying to figure out why something is broken, the process is stepping forward deeper into the code … but then you have to work your way back and frequently that involved re-executing the process. If there was some sort of history … that would be useful" (Subject 15). This concept is not novel and has been successfully implemented (e.g., [18], though no general solution for integration into common IDE debuggers has been.

Observation: Better support for reproduction, log file analysis, multithreaded debugging, and backward-in-time debugging is needed. While all of these concepts have been demonstrated in research, scalable, real-world solutions are needed.

IX. Study limitations

This study is subject to the limitations described below.

A. External validity

The coding results (bug descriptions, demographics, debugging information used, debugging techniques, challenges, and needs) are limited to the 15 subjects selected from Microsoft. This sample is not representative of the software development population as a whole. Furthermore, our model of debugging may not generalize to all individuals in all domains. The goal of this work is exploratory, and to guide future research paths into contemporary debugging problems.

B. Internal validity

The group of potential study participants was self-selected, and the interviewees were selected based on subject expertise and domain area. As such, the subjects are not necessarily a representative sample of the population within Microsoft, or the greater software developer population as a whole. Rather, subjects were intentionally selected to obtain a diverse set of responses and examples. The interviews were limited to 60 minutes, and not all interview questions were asked of all subjects. Thus, the analysis of the most-frequently-mentioned topics may not be accurate within the sample as all subjects were not given an equal chance to respond to all questions.

C. Construct validity

The interview questions reflect the research interests of the authors and thus do not capture every facet of the debugging milieu. The questions were structured to be open-ended so that respondents could provide their own examples of bugs, the challenges they face in debugging, and the needs they have to improve the state of the practice. The diverse responses of the interviewees suggest that the questions and sampling were appropriate for understanding the breadth of debugging concerns within the sample. The model of debugging derived from these interviews and coding analysis is convergent with previous models of debugging and program comprehension, thus providing some indication that the questions asked were appropriate to converge with past findings.

X. Conclusion

In this paper, we explore debugging in a professional setting based on the qualitative analysis of 15 interviews with developers at Microsoft. We provide discussion of information sources and debugging methods used by these professionals based on descriptions of actual bug investigations provided by the developers. The primary goals of this research were to explore what information and processes professionals use to debug, and to identify the challenges introduced by the professional environment.

The study also reveals several debugging challenges faced in some professional settings: 1) the interaction between *hypothesis instrumentation* (altering the source code and debugging environment) and some software environments (e.g., web services) introduce a lag time between hypothesis instrumentation and testing; 2) the impact of log file information on accurate debugging of web services; and 3) the

mismatch between the sequential human thought process and the non-sequential execution of multithreaded environments as source of difficulty.

The interviewees identified several potential *areas of improvement*: 1) better tools for capturing and replaying failures; 2) better tools and processes for analyzing log files and relating them to source code; 3) better techniques for controlling thread execution and monitoring data flow in multithreaded debugging; and 4) widespread support for a backward (or rewind) debugger that allows developers to step backward in program execution. Some of these desired improvements are not new, and researchers have attempted to answer the challenges for capturing and replaying failures [16], analyzing log files [19], multicore and multithreaded debugging [20], and backward debugging [21]. The question is obvious: why do these solutions not transfer from the state-of-the-art to the state-of-the-practice?

We hope that researchers will take these findings as a call-to-arms to help solve the debugging challenges of professionals with techniques and tools that support the professional work environment. We hope that researchers can leverage these findings to develop new techniques in new challenges areas, and that they will investigate new tools and techniques *in situ* to provide practical, grounded results.

ACKNOWLEDGMENT

We would like to thank the participants in this study for their time and helpful insights. We would also thank the National Research Council of Canada, the Fraunhofer Center for Experimental Software Engineering, the National Sciences and Engineering Research Council of Canada (NSERC), and Microsoft Research for funding and facilitating this research.

REFERENCES

[1] ISO/IEC/IEEE, "24765:2010(E) Systems and software engineering - Vocabulary," ISO, Geneva, Switzerland, 2010.

[2] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, 2004.

[3] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT Int'l Symp on Foundations of Software Engineering*, 2006, pp. 23–33.

[4] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance masks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[5] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?," pp. 255–265, Jun. 2012.

[6] C. M. Kessler and J. R. Anderson, "A model of novice debugging in LISP," in *First Workshop on Empirical Studies of Programmers*, 1986, pp. 198–212.

[7] I. R. Katz and J. R. Anderson, "Debugging: an analysis of bug-location strategies," *Hum.-Comput. Interact.*, vol. 3, no. 4, pp. 351–399, Dec. 1987.

[8] D. J. Gilmore, "Models of debugging," *Acta Psychologica*, vol. 78, no. 1–3, pp. 151–172, 1991.

[9] A. von Mayrhauser and A. M. Vans, "Program understanding behavior during debugging of large scale software," in *Seventh Workshop on Empirical Studies of Programmers*, 1997, pp. 157–179.

[10] G. W. Ryan and H. R. Bernard, "Data management and analysis methods," in *Handbook of Qualitative Research*, 2nd ed., N. K. Denzin and Y. S. Lincoln, Eds. Thousand Oaks, CA: Sage, 2000, pp. 769–802.

[11] C. B. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 24, pp. 557–572, 1999.

[12] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams." Minneapolis, MN, pp. 344–353, 2007.

[13] J. Freire, D. Koop, E. Santos, and C. Silva, "Provenance for computational tasks: A survey," *Computing in Science & Engineering*, vol. 10, no. 3, pp. 11–21, May 2008.

[14] T. J. Leblanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transactions on Computers*, vol. C–36, no. 4, pp. 471–482, Apr. 1987.

[15] C. Dionne, M. Feeley, and J. Desbiens, "A taxonomy of distributed debuggers based on execution replay," in *Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, 1996, pp. 203–214.

[16] A. Orso and B. Kennedy, "Selective capture and replay of program executions," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, Jul. 2005.

[17] W. Jin and A. Orso, "BugRedux: reproducing field failures for in-house debugging," pp. 474–484, Jun. 2012.

[18] H. Agrawal, R. A. De Millo, and E. H. Spafford, "An execution-backtracking approach to debugging," *IEEE Software*, vol. 8, no. 3, pp. 21–26, May 1991.

[19] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '11*, 2011, vol. 46, no. 3, p. 3.

[20] A. Spear, M. Levy, and M. Desnoyers, "Using Tracing to Solve the Multicore System Debug Problem," *Computer*, vol. 45, no. 12, pp. 60–64, Dec. 2012.

[21] G. Pothier, É. Tanter, and J. Piquer, "Scalable omniscient debugging," *ACM SIGPLAN Notices*, vol. 42, no. 10, p. 535, Oct. 2007.