Down to Hades and Back – Experiences Gained in Comprehending a Distributed Legacy System

Pavol Dano*†, Andreas Bollin *
*Alpen-Adria University of Klagenfurt, Klagenfurt, Austria
andreas.bollin@aau.at, pavol.dano@aau.at

†Technical University of Košice, Košice, Slovakia
pavol.dano@gmail.com

***Abstract*—Software engineering principles and practices face a lot of challenges. Among them, debugging and comprehending systems is still demanding. Furthermore, comprehending distributed systems, even though there are several papers and approaches to be found in literature, is a challenge of a completely different dimension. This paper describes the experiences gained when trying to comprehend (and debug) a distributed legacy system and it takes a closer look at the different options somebody has in similar situations. It does not propose specific tools or methodologies, but tries to provide guidelines to developers who might be challenged by comparable problems.**

I. Motivation

Among many others, software engineering principles and practices face a lot of challenges. As our programs are getting bigger, are embedded in the real world, and age, they become even more difficult to understand and to maintain. In 1995, Brooks [1, p.182] already mentioned that "software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). [...] In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound."

Following the arguments of Banker, Davis and Slaughter [2], to understand during forward engineering is not without effort, but the situation gets worse if one is changing from the construction phase to maintenance, reverse engineering or design recovery. In 2014, we experienced such a situation at first hand. In lectures at two Universities (Austria and Slovakia) we are using the AMEISE (A Media Education Initiative for Software Engineering) framework [3], [4] which focuses on the simulation of software project management processes. The related project started at the Alpen-Adria University in Klagenfurt in 2001, and, apart from the pedagogical aspects, the software is implemented as a distributed system consisting of any number of clients, one load balancing manager, and up to n simulation cores - all of them running on different engines and summing up to 140.000 LOC in software. Over the past 13 years more than 40 developers contributed to the system, and apart from one co-author of the paper, all stakeholders have left the project.

After a quite stable phase during the past 4 years, we experienced serious problems with the software during a simulation run in Košice at the beginning of 2014. The system became unstable, and we had a hard time to finish the lecture as intended. The need to debug the system became evident as we then had similar problems in Klagenfurt – the system either stopped or skipped simulation steps.

In Klagenfurt, we thus started a project with one developer (doing an Erasmus placement) dedicated to this situation, and the objective of the paper is now twofold. First, we strongly believe that one always can learn from post-mortem project reports, and so the paper tries to motivate others by sharing our experiences (and highlighting drawbacks) in such circumstances. Secondly, it introduces a tailored approach (we called it "pair debugging") which helped us in finally comprehending the underlying system and fixing the problems successfully.

The paper is structured as follows. Section II introduces the background of this work. Section III addresses the challenges one might have to deal with. Section IV then presents the approach that we followed and we think was the basis for successfully fixing all the bugs. Finally, Section V concludes the work with a summary and some outlook of work that needs to be done in the future.

II. Background

A. The AMEISE System

AMEISE is a Client/Server system using a simulation engine called SESAM (Software Engineering Simulated by Animated Models). SESAM has been developed at the University of Stuttgart under the direction of Jochen Ludewig [5]. The major differences between SESAM and AMEISE is that AMEISE is distributed among several server engines and that it heavily relies on helper components built around the simulation core. Additionally, key data of every simulation step is stored in a MySQL database for later assessment and visualization.

The system follows a state-less Client-Server architecture. Due to the high need of resources for the simulation cores, requests from clients are distributed to several server engines via a dedicated load balancing server. Quite from the beginning, a CVS system was used, the development environment was standardized and fixed, and there is a rich set of (honestly still lacking) documentation in at least two languages, German and English. In fact, it seems to be a "typical" SW system produced at Universities, where quality is an issue, but due to

TABLE I: Shortlist of comprehension/debugging techniques and related papers.

Label	Technique	Related paper
T01	Logging & Tracing	[6][7][8]
T02	Path Rules	[9][10]
T03	Slicing & Dicing	[11][8]
T04	Statistical Debugging	[12]
T05	Visualization	[13][14][8][15]
T06	Replay Technique	[11] [14][15]
T07	Controlled Execution	[14]
T08	Breakpoints	[14][15]
T09	Algorithm Recognition	[16][17]
T10	Model-based Debugging	[11][6][18][19]
T11	Assertions/Predicates	[14][**?**][9][10]
T12	Query-based Debugging	[6][20]
T15	Delta Debugging	[21]
T16	Software Reconnaissance	[22][8]
T17	Code Reviewing	[23]
T18	Bug (Defect) Prediction Modeling	[24][7]
T19	Filtering	[21][14][6][7][8] [25][15]
T20	Clustering	[21] [25]
T21	Plan Recognition	[26][17]
T22	Language Consistency Checking	[17][10]
T23	Bug Cliche Recognition	[16][26]

time constraints and varying developers' skills, portions of the code vary in their quality properties.

Up to January 2014, the simulation environment ran stable at our server farm in Klagenfurt and we hosted more than 2.000 trainees over the past 10 years. However, we had a group of 150 students in a lecture at the Technical University in Košice (TUKE) in January 2014, and the system "suddenly" became instable. The system in Klagenfurt was accessed remotely via the WLAN network from TUKE (which worked a year before without problems), and we experienced the following: firstly, some of the client interfaces got stuck, not being able to proceed with any other simulation step. Secondly, some of our server engines raised exceptions (duplicate key entry problems in the database) and stopped working. And finally, the response times got worse during the simulation run, varying between 4 and 200 seconds per simulation step.

At TUKE, we found out that the bandwidth of the WLAN network caused problems. So, we expected that some data packages got lost and database entries were inconsistent – leading to all the other problems. But, as we had the same situation then in Klagenfurt in April 2014 (with fast and stable internet connections), it was evident that the bandwidth of the WLAN was not the (only?) problem.All we knew was that during the last two years the hardware the server farm was running on had been partially upgraded, and as there was no developer with system knowledge available, we set up a new software maintenance project.

B. Approaches and Tools

At first, we took a closer look at techniques and methods related to comprehending and debugging software systems. By looking at more than 100 articles, it turned out that the techniques listed in Table I seem to be the most relevant ones.

We also took a closer look at the problem fields (error classes) that we thought were related to our topic. Hayes et al. [27, p.128–134] did a very useful classification of errors which we then decided to follow. As we were not able to start with specific scenarios or perspectives on requirements, the following abstract and high-level classification seems to be the best we could start with:

Correctness Problems. This class is divided into the following sub-classes: Logical Problems (coding mistakes like infinite loops, code dependency errors, communication typology and inappropriate data distribution errors), Coding Problems (memory leaks and overwrites, and communication miss-specification), and Language Problems (errors caused by erroneous send/receive syntax and semantics, incorrect object definitions and classifications, and inappropriate data distribution maps).

Performance Problems. They result in an inevitable suboptimal system performance. This problems class can be divided into the following sub-classes: System Efficiency (e.g. unnecessary synchronization barriers, inappropriate changes of the data distribution), Redundancy (redundant data access or computations), Data Locality, Coding Efficiency (poor code organization and inappropriate communication topology), and I/O Error.

The process model we followed (see Section IV-A for details) allows for a perennial selection of suitable tools or techniques. At the beginning, however, we did not know anything about the problem type and which flow of control was leading to the problem. We also had no older (correct) version of the code-base available. With the restricting factor that some tools or approaches were also not available or suitable for Java and our environment, we ended up with the following set of techniques to be used in our situation:

1) Looking at problem classes, we assumed logical (code dependency, communication topology), efficiency (synchronization) and coding problems within our system. So, we used techniques T01 (Tracing), T05 (Visualization), and T20 (Filtering) in combination with T21 (Clustering) as our recurring standard techniques.

2) In order to comprehend and narrow down the problem space, especially at the beginning of our endeavor, we enriched our standard techniques by techniques T15 (Delta Debugging), T17 (Code Reviewing) and T18 (Bug Prediction).

III. The Software Analysis Task

The paper is called "Down to Hades and Back" as comprehending (going down) and debugging (coming back again) is at least as challenging as entering the Hades in the Greek mythology and leaving it in one piece again. We experienced such challenges, tried to understand why they appeared, and attempted to learn from it. The challenges are written in sequential order, but, please note that it does not fully match the chronological order in the process we really followed.

A. Methodology Selection (The Tortures)

For a wide variety of specific problems there are a lot of approaches and tools, some of them called methodologies nowadays. But, two questions immediately arise.

(a) The first question is how to find a match between the problem class(es) and the techniques described in literature. It turns out that there are already too many "trees in the forest",

and available textbooks are by far too general. The situation gets even worse when you do not know WHY your system fails as this means that there are even more forests to be considered.

(b) The second question pops up when you know the problem class but find more than one method that could be helpful. Here, one needs to find out where to start in order to prevent from wasting too many resources.

Finding the bug(s) seems to be hard in situations like ours as we lacked in knowledge of the problem class and about the set of suitable methodologies and techniques. The challenge then became a torture as we did not know HOW to start the debugging process. Without guidance, one has to try out one method by the other. Mapped to our metaphor, one needs a guide to find the entrance to the Hades.

We started with the technique of logging and tracing (as we had at least the exception traces pointing to some portion of code) in order to come closer to the assumed error. But, we soon learned that the exception was a side-effect and thus we could not narrow down the portion of the code to be examined. Therefore, we did not apply any third-party library, but found it more practical to enhance our debugging module with a set of methods we then called at hot spots that we wanted to analyze. It took us 1 week till we excluded approaches we could not apply and finally came up with a slightly modified debugging technique explained in some details in Section IV.

B. Facing with Side-Effects (The Observation Sin)

Experienced developers might already have been noticing the mistake we made when using logging and tracing (by adding output statements to the code). We underestimated the fact that we are dealing with a system that has multiple threads running in parallel, most of them distributed. Well, we of course thought that we handled events correctly, but what we did not expect was the fact that not all of the threads were synchronized carefully (which we found out later). By purely adding lines of code we ran into the probe-effect.

It is like Orpheus who escaped with Eurydice from Hades. He was overwhelmed by a desire to look behind him to make sure his wife still followed him. And, when he looked back, Eurydice was pulled back into the Nether Regions. In our case, it took us 2 days to deal with the problem. The way of observation was our sin.

C. Replay Considerations (Time is the Obolus)

We still have not arrived at Hades yet. Another factor that has to be considered is the amount of resources that are needed in order to comprehend the system. So, it is about the Obolus one has to pay when crossing the river Styx. In our case we were lucky to have a batch-facility in AMEISE that allowed us to conduct simulation runs in an auto-play mode. However, the problem only appeared in 10% of all the simulation runs, and also was dependent on the number of server engines running. Even more, one simulation run lasts at least 3 hours. In all, this limits the amount of tests one can do a day. Charon, the ferryman, had raised the price for crossing the river Styx. It took us another 3 days to come up with a setting that allowed us to reliably reproduce the problem.

D. Size of Log Data (Crossing the Data Styx)

Debugging in our case meant to combine the debugging process with reverse engineering steps. Thus, one collects and produces data in various forms. The problem here is that it really is a lot of data which is generated. The diagrams and models we recreated during reverse engineering steps were large and one full simulation round with 20 clients also produces several Gigabyte of log data (in just 3 hours). So, we additionally used filtering and clustering techniques to help us.

There are few other facts that made crossing the river Styx (our log data) even more complicated. For example, the simulation cores are deployed to multiple machines and these machines initialize multiple processes that communicate with each other. As only one log file is produced on the level of the single module, we neither could guarantee that the aggregated log (a log file created by merging all existing log files) would have execution representatives of all possible executions of the system. With that, an approach like the one suggested by Beschastnikh et al. [29] (where AspectJ is used for monitoring socket connections) was not suitable in our situation.

Comprehending our log files turned out to be a challenge. There are still a lot of old debugging statements in the AMEISE system (from previous maintenance activities), and the question now is which of them to reuse, which of them to neglect, and, even more difficult to answer, how to interpret their meaning. A lot of old log-statements were not clear due to a lack of documentation. After a day we decided to disable all of them (before endlessly guessing their meaning) and to introduce *and* document our own log statements.

E. Camouflage Effect (Escaping Hades, but how?)

Another big challenge is that even in simple systems a set of problems can be camouflaged by a single problem. Exactly that happened in our case. Based on past experiences with the AMEISE system (and its stability) we expected a single problem that yields the erroneous behavior, but at the end of the day it turned out that there were several problems (hidden in the code for years) contributing to the observed effect. Greek mythology shows us that there are several ways of how to leave the Hades again. Also, in our case we needed to consider more than one path to succeed. At the end, it turned out that the buggy situation had several reasons and they existed in the code for more than 7 years:

First, due to an hardware upgrade, our processes were much faster and the number of open sockets (due to more polling requests) was too high for the operating system and the Java connections. Additionally, polling delays and timeouts were not adopted to the new hardware speed. Secondly, the database was not configured to handle that many concurrent requests at a time (about 1.5 Mio. SQL requests in 3 hours). Next, some threads, especially those operating on request and response queues, were not synchronized. Also, the load balancing mechanism had a bug in the distribution algorithm (where the new polling time was calculated) – only appearing in extremely rare and special timing conditions. It was the one issue that was most difficult to identify. And finally, the load balancing mechanism had a bug when a simulation core was marked as being down.

F. *Finally – Breaking with Myths*

As mentioned in the introduction section, debugging concurrent and distributed systems is difficult. This section now reflects on the challenges mentioned above and concludes with hints for improving the situation a bit.

(1) At first, there was the problem of selecting the *right methodology* (our tortures). To our experience, available approaches are either too specific or too general to be applicable. Furthermore, there is a lack in papers describing HOW to start with a comprehension and debugging project and HOW to narrow down the problem class.

In our case, we classified the approaches we found in literature so that we have a better overview (see Section II-B), introduced a modified version of peer reviewing (based on discussion and reflection), and made use of code metrics to speed up the process (see Section IV for process details).

(2) Another difficulty is in *reproducing* the erroneous behavior in a controlled manner (which reminded us on the observation problem). The approach to be used heavily depends on the problem at hand, but one should be aware that an observed system might change its behavior. A stepwise approach (less is more) is advisable, not adding too much to a system at a time. And, every step has to be documented and verified in respect to what one expects.

In our situation, we had to extend the code base (by synchronizing events) so that we were able to place reliable measure and log points to the system. We collected statistical data about the run-time behavior and compared every new version with the old one, following the notion of delta debugging.

(3) The next problem to be considered was a restriction in the amount of *available resources* (so, the Obolus to pay). Here, one has to think about a reasonable tradeoff between different methodologies and the time and resources one can spend for it.

In our case, we used versioning for all of our log data as it turned out that we needed to jump back to the results of older experiments and test runs. We also organized a text editor which was able to deal with large log files and made use of scripts, filtering and clustering techniques as often as possible.

(4) Another challenge is then how to *interpret the large amount of data* you collect (crossing the Data Styx). Assigning meaning to the data is crucial, but to our experience one is tempted to collect more than necessary at the beginning. A rule might be to think about ways in how to understand and filter the data right from the beginning. And then, the principle of "less is more" should be followed again. Another issue is that one really needs to take notes (best is to document it in the version control system AND in the artefact). It has to be clear WHY one is doing WHAT and WHERE.

In our case, we defined several problem dimensions (i.e. network, database, memory, file system, time behavior) that we thought to need to take a look at, used unique labels and expressive descriptions, and exported the data in such a way that it can be read by a spreadsheet environment. Then we used simple visualization techniques to compare the assumed behavior and the actual behavior.

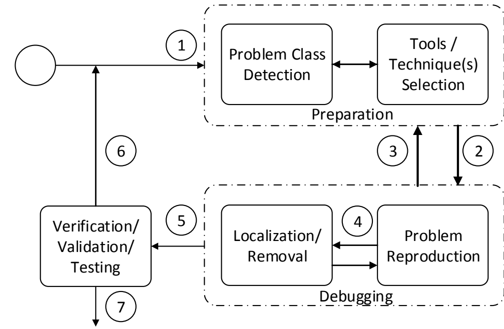

Fig. 1: Necessary steps when comprehending and debugging a system without knowing the error class/techniques to be used.

(5) The last issue is that there might be *a camouflaging problem* hiding other problems (hiding our escape options). As long as we do not know exactly what the problem is, we need to be prepared for multiple problems and solutions. This makes the decision about which technique to follow extremely hard. One heuristic might be to use those techniques that cover most of the assumed problem fields, and we decided to observe several problem dimensions at the same time.

After trying out different techniques, we noticed that we need to follow our own systematic approach, and we called it "pair debugging" (see next section for details).

IV. A Tailored Approach

A. *Comprehend and Debug – Debug and Comprehend*

In our case we had to deal with a situation comparable to the situation described by Zeller in his article "Yesterday, My Program Worked. Today, It Does Not. Why?" [21]. Retrospectively, it turns out that it is clear why our emotions during the comprehension and debugging process had their ups and downs. Figure 1 exemplifies the most important steps we had to follow during first comprehending and then debugging the AMEISE system. Up to now this process can be described as follows:

With the arrival of a problem (or a set of problems) at point (1) we are entering a phase we called *Preparation* and where the detection of a problem/error class and the selection of suitable tools and techniques go hand in hand. It is also the part that complicated our situation and led to the introduction of an own debugging technique helping us with identifying the problem classes. When one is sufficiently convinced that the selected technique might be helpful, (2) then it comes to the sub-process that developers usually call "Debugging". However, debugging might (partially or fully) fail, and (3) it might then be necessary to go back to the preparation phase. Taking a closer look at this process, debugging incorporates at least two sub-processes (4). First, activities for being able to reproduce the problem when needed, and, secondly, activities for the localization and removing of the bug (or bugs). After completing the debugging process (5), quality assurance activities take place. Figure 1 mentions testing, verification, validation as some popular representatives. Finally, this quality assurance process could start a new cycle in case of problems (6) or terminate the process (7).

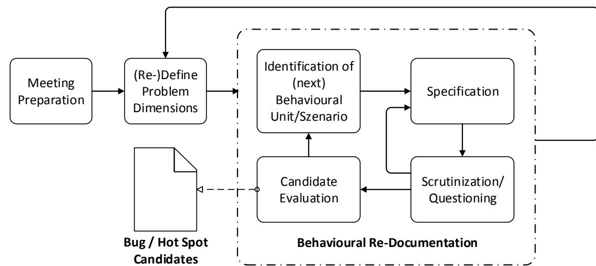

Fig. 2: Process behind pair debugging, the core technique that we used for understanding and localizing the bugs in the AMEISE system.

B. Pair Debugging

Section IV-A mentioned that especially the preparation phase was the challenging part. How to proceed when you do not know where the problem is (so, which error class to start with)? Based on the given situation (where we had one novice developer in the debugging team) we decided to borrow the ideas from team programming and to create our own debugging approach out of it. We called it *team debugging*. It is a mixture of comprehension and debugging techniques, and as such it differs from expert code reviews (class T18).

The process is exemplified in Figure 2 to some extent. First, it starts with the *preparation* for the meeting. Room and resources (whiteboard, scratch paper, pencils ...) are organized, and the pair is formed. A pair is composed of two persons in our case, taking two different (and quite opposite) roles. The first role is the *Developer* who is responsible for digging into the code. This person reads the code on the statement level, whereas the other role is the *Manager*, a person with technical background, but who acts like a member of the project board who has to understand the system from a senior users' and executives' perspective. The overall task is, of course, broken down into manageable pieces (for 1/2 day meetings), but then the preparation phase differs for these two roles.

The *Manager* reads through the available documentation (architecture level). He or she then looks at existing reports, and, as in our case, also at code metrics. The problem report is studied (it is allowed to look at code), and the preparation phase has the objective to prepare the manager for guiding the debugging process. Metrics, results from fault prediction models, and the manager's experience should help in generating a preconceived opinion about possible hot-spots in the code.

The *Developer* reads through the code and tries to understand the flow of control and data as fast and as reliable as possible. This prepares him or her to be a selective respondent during the debugging process.

The next step is then the selection of the problem dimensions. As we really did not know where to start, we decided to look at several dimensions (like time or network connection) at the same time.

The heart of the technique is then the behavioral (re-) documentation of the system. The Manager suggests which unit (component or class) or scenario to start with or to select next (*Identification* step) and the pair agrees about it. Then, the Developer explains what happens in the code and the Manager writes down the specification of it (*Specification* step). In such a way, the produced diagrams are used to document the behavior of the components, but also to write down striking features and questions. The job of the manager is to ask back everything difficult to understand or being too complex (*Scrutinization* step). Here, the look at metrics and other reports beforehand turned out to be very helpful. Questions like "I don't believe that the load balancing algorithm does really work. Can you convince me?" or "Why are we loosing 500 ms here?" finally led, after evaluating them (*Candidate Evaluation* step), to a set of candidates of possible bugs but also areas and hot-spots in the code to look at.

The documentation stays at a quite abstract level, and it has the following three purposes: first, to guide the pair through the system, secondly, to help reviewing large portions of code in short time, and thirdly, to systematically collect areas in (or effects of) the code which might contribute to the problem you try to solve. The document is then used to test assumptions that you have against the effects in the system.

Checking all the candidates needs time, but in our case, we managed to came up with a list of potential problem classes for all three components within three 2.5 hours meetings. The method relies on an old hypothesis, namely that one assume that there are more bugs in areas which are hard to understand and to explain.

C. Limitations

In our case, the approach had its merits as we were able to localize the hot spots in a systematic manner. But, it can not guarantee that all hot-spots are identified promptly, and here at least two limitations should be mentioned.

Firstly, it does not help with the decision of which technique or tool to start with. In our case we had to deal with both, functional and performance bugs, and at the end of the debugging phase it turned out that most of our bugs were performance-related. Focusing on the area of performance engineering might have been working in our case, too.

Secondly, the technique is also not immune to mistakes. In our case, at the first peer-review meeting the Manager said that he did not understand the load-balancing mechanism. He thought that the problem was there. The code was reviewed, but no problem was detected by the Developer. However, due to a misunderstanding the Developer looked at portion of the code out of the scope of the actual problem. The candidate was removed from the candidate list. One week later, by making use of visualization techniques, the issue was brought back to the candidate list – basically it was *the* problem to be solved so that stability was reached again.

V. Conclusion

This contribution presents a real-world scenario which shows the difficulty of comprehending and debugging a distributed Client-Server environment. It explains the impediments (being inspired by Greek mythology) that we had to deal with, but also presents our lessons learned. We also present the process model we followed and think that, put into a broader context, it could be the starting point for a more general model used by other developers in the future.

Our project findings are manifold and are, considered separately, not really surprising. But, there is one issue that turned out to be a necessity right from the beginning: to start with a stepwise approach where *every* decision or action is documented and versioned for later (re-)use. All tools and techniques we looked at seem to have their merits, but it is the human factor which plays the major role in such a project. It is the ability to critically reflect on what is summarized, visualized, abstracted, pointed at, or discussed. And, only by versioning all steps (decisions, modifications, results) during the comprehension and debugging phase, the human developer has the necessary freedom to work on the problem at hand, not worrying about memories fading away.

REFERENCES

[1] J. Frederick Philips Brooks, *The mythical man-month: essays on software engineering - Anniversary edition*. Addison Wesley, 1995.

[2] R. D. Banker, G. B. Davis, and S. A. Slaughter, "Software development practices, software complexity, and software maintenance performance: A field study," in *Management Science*, vol. 44, no. 4. Institute for Operations Research and the Management Sciences, April 1998, pp. 433–450.

[3] R. Mittermeir, E. Hochmüller, A. Bollin, S. Jäger, and M. Nusser, "AMEISE - A Media Education Initiative for Software Engineering: Concepts, the Environment and Initial Experiences," in *Proceedings International Workshop ICL - Interactive Computer Aided Learning, Villach*, M. Auer, Ed., Sept. 2003, iSBN 3-89958-029-X.

[4] A. Bollin, E. Hochmüller, and R. Mittermeir, "Teaching Software Project Management using Simulations," in *Proc. 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T 2011)*, J. B. Thompson, E. O. Navarro, and D. Port, Eds., 2011, pp. 81–90.

[5] A. Drappa and J. Ludewig, "Simulation in Software Engineering Training," in *Proceedings, 23rd International Conference on Software Engineering, IEEE-CS and ACM*, May 2001, pp. 199–208.

[6] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu, "Efficient Data Race Detection for Distributed Memory Parallel Programs," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 51:1–51:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063452

[7] M. Ganai, N. Arora, C. Wang, A. Gupta, and G. Balakrishnan, "BEST: A symbolic testing tool for predicting multi-threaded program failures," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, Nov 2011, pp. 596–599.

[8] W. Wong, S. Gokhale, J. Horgan, and K. Trivedi, "Locating program features using execution slices," in *Application-Specific Systems and Software Engineering and Technology, 1999. ASSET '99. Proceedings. 1999 IEEE Symposium on*, 1999, pp. 194–203.

[9] Z. Xu, J. Zhang, and Z. Xu, "Memory Leak Detection Based on Memory State Transition Graph," in *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, Dec 2011, pp. 33–40.

[10] M. Ducassé and A.-M. Emde, "A Review of Automated Debugging Systems: Knowledge, Strategies and Techniques," in *Proceedings of the 10th International Conference on Software Engineering*, ser. ICSE '88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 162–171. [Online]. Available: http://dl.acm.org/citation.cfm?id=55823.55841

[11] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "Formal Verification of Practical MPI Programs," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 261–270. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504214

[12] J. S. Vetter and M. O. McCracken, "Statistical Scalability Analysis of Communication Operations in Distributed Applications," in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, ser. PPoPP '01. New York, NY, USA: ACM, 2001, pp. 123–132. [Online].

[13] S. Bassil and R. Keller, "Software visualization tools: survey and analysis," in *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, 2001, pp. 7–17.

[14] Rogue Wave Software, "PDF User Guide, Version 8.14.1," Online, http://www.roguewave.com/help-support/documentation/totalview. Page last visited: Nov. 14th, 2014.

[15] S. Shende, J. Cuny, L. Hansen, J. Kundu, S. McLaughry, and O. Wolf, "Event and State-based Debugging in TAU: A Prototype," in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, ser. SPDT '96. New York, NY, USA: ACM, 1996, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/238020.238030

[16] W. L. Johnson and L. A. Gladwin, "Intention-Based Diagnosis of Novice Programming Errors," *IEEE Expert*, vol. 2, no. 3, pp. 94–94, Sept 1987.

[17] C.-K. Looi, "Automatic debugging of Prolog programs in a Prolog Intelligent Tutoring System," *Instructional Science*, vol. 20, no. 2-3, pp. 215–263, 1991. [Online]. Available: http://dx.doi.org/10.1007/BF00120883

[18] T. Jeron, J.-M. Jezequel, and A. Le Guennec, "Validation and test generation for object-oriented distributed software," in *Software Engineering for Parallel and Distributed Systems, 1998. Proceedings. International Symposium on*, Apr 1998, pp. 51–60.

[19] W. Mayer and M. Stumptner, "Model-Based Debugging - State of the Art And Future Challenges," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 4, pp. 61–82, 2007, proceedings of the Workshop on Verification and Debugging (V&D 2006). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S157106610700196X

[20] M. Subramaniam, "Early error detection in industrial strength cache coherence protocols using SQL," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, April 2003.

[21] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7. London, UK, UK: Springer-Verlag, 1999, pp. 253–267. [Online]. Available: http://dl.acm.org/citation.cfm?id=318773.318946

[22] N. Wilde and M. C. Scully, "Software Reconnaissance: Mapping Program Features to Code," *Journal of Software Maintenance*, vol. 7, no. 1, pp. 49–62, Jan. 1995. [Online]. Available: http://dx.doi.org/10.1002/smr.4360070105

[23] M. Mantyla and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?" *Software Engineering, IEEE Transactions on*, vol. 35, no. 3, pp. 430–448, May 2009.

[24] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, May 2010, pp. 31–41.

[25] M. Syer, Z. M. Jiang, M. Nagappan, A. Hassan, M. Nasser, and P. Flora, "Leveraging Performance Counters and Execution Logs to Diagnose Memory-Related Performance Issues," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 110–119.

[26] R. L. Sedlmeyer, W. B. Thompson, and P. E. Johnson, "Knowledge-based Fault Localization in Debugging: Preliminary Draft," in *Proceedings of the Symposium on High-level Debugging*, ser. SIGSOFT '83. New York, NY, USA: ACM, 1983, pp. 25–31. [Online]. Available: http://doi.acm.org/10.1145/1006147.1006154

[27] A. H. Hayes, J. S. Brown, and M. L. Simmons, *Debugging and Performance Tuning for Parallel Computing Systems*. Computer Society Press, 1996, no. BP07412.

[28] P. Emrath, S. Ghosh, and D. Padua, "Detecting nondeterminacy in parallel programs," *Software, IEEE*, vol. 9, no. 1, pp. 69–77, Jan 1992.

[29] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 468–479. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568246