

Monitoring, Analysis, and Testing of Deployed Software

Alessandro Orso
Georgia Institute of Technology
Atlanta, GA 30332-0765
orso@cc.gatech.edu

ABSTRACT

Modern software is increasingly ubiquitous, commoditized, and (dynamically) configurable. Moreover, such software often must be able to operate in a varied set of heterogeneous environments. Because this software can behave very differently in different environments and configurations, it is difficult to assess its quality purely in-house, outside the actual time and context in which the software executes. Consequently, developers are often unaware of how their systems actually behave in the field and how their maintenance activities affect such behavior, as shown by the countless number of incidents experienced by users because of untested behaviors. On the bright side, the complexity of today's computing infrastructure and of modern software also provides software engineers with new opportunities to address these problems. The ability to collect field data—data on the runtime behavior of deployed programs—can provide developers with unprecedented insight into the behavior of their deployed systems. We believe that the collection and analysis of field data can provide disruptive advances in the state of the art in software engineering. In this paper, we discuss our vision and a research agenda that can help fulfill such vision.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Algorithms, Experimentation, Reliability

Keywords

Testing deployed software, field data, runtime behavior

1. INTRODUCTION

Quality-assurance activities, such as software testing and analysis, are notoriously difficult, expensive, and time-consuming. As a result, software products are typically released with faults or missing functionality. The characteristics of today's software are making the situation even worse. Modern software systems are increasingly ubiquitous, commoditized, and configurable. Moreover, these

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

Copyright 2010 ACM 978-1-4503-0427-6/10/11 ...\$5.00.

systems are often built through aggregation of third-party components, the set of interacting components can vary widely (*e.g.*, in the case of the increasingly popular service-oriented architectures), and the nature of the interactions among components can be highly dynamic. Finally, the environments in which such systems must operate are often rich and heterogeneous. More generally, software complexity is growing, together with the complexity of the environments in which software executes.

Because this software can behave very differently in different environments and configurations, it is extremely difficult to assess qualities such as performance, reliability, and security outside the actual time and context in which the software executes. Developers' traditional approaches to system understanding and validation were developed decades ago, for a different kind of software. Such software assessments—performed by a relatively small group of experts, in-house, on their own computing platforms, using input workloads they generated—can still provide useful information, but are frequently inadequate for today's systems. Consequently, developers are often unaware of how their systems actually behave and perform in the field and how their maintenance activities affect such behavior and performance. In fact, the examples of incidents experienced by users because of untested behaviors (*e.g.*, due to configuration problems and unforeseen interactions) are countless. We experienced this problem directly, when an analysis tool developed at Georgia Tech was consistently crashing for one of its users. After investigating the issue, we found that the problem was due to the use of the tool on a specific combination of versions of the Java Virtual Machine and of the Solaris Operating Systems [20]. Although this problem could have been discovered through in-house testing, such a discovery would have required the testers to exercise the software in that specific configuration.

If the complexity of today's computing infrastructure and of modern software introduces new problems for software engineers, however, it also provides new opportunities that may help in addressing these problems, if suitably leveraged. In particular, many of today's software systems are deployed in a large number of similar (when not identical) instances, used by a multitude of users, and executed on powerful computers that are mostly interconnected. This situation provides a unique chance for developers to collect *field data*, that is, data about the runtime behavior of deployed programs collected by monitoring the programs while they are in use.

Field data can provide developers with unprecedented insight into the behavior of their deployed systems, which can help them in a number of quality assurance activities. In fact, in recent years, we have seen an increasing interest in this topic from both researchers and practitioners; and the development of techniques that collect data from deployed applications to support in-house software engineering tasks is an increasingly active and successful area of re-

search (e.g., [4, 5, 8, 14, 21, 23]). These existing approaches are a promising beginning, but we believe that they just scratch the surface of the possibilities offered by field data.

Our vision is that further research in this area can help leverage the untapped potential of field data by going way beyond the state of the art and giving developers the ability to answer richer questions about the users' experience with their software. For example, we envision developers being able to investigate the following issues: what bugs users are experiencing most frequently and in what system, platforms, and library combinations they are occurring; what are the possible reasons for a given observed field failure or misbehavior; whether a test suite is exercising the code in the same way the users are exercising it and, if not, how to improve such test suite; whether some subsets of users are experiencing poor performance; which users would be affected by a given modification in the code and how; and how to recreate in the lab failures experienced by the users.

In short, on the one hand, software engineers face a situation in which traditional in-house quality-assurance techniques are often ineffective due to their inability to cover the spectrum of behaviors manifested by software systems after deployment. On the other hand, the ability to leverage field data and resources promises not only to produce a new set of approaches, techniques, and tools that overcome the limitation of the state of the art, but also to help solving new problems that it was not possible to address before.

2. OVERALL VISION

It is generally recognized that executions in the field can manifest a quite different behavior than in-house executions, especially for software that can operate in different configurations and environments. In previous work, for instance, we provided concrete evidence of this difference in behavior while experimenting with a technique that leveraged field data to perform impact analysis and regression testing [19]. Being able to capture, analyze, and understand these behavioral differences could greatly benefit many quality-assurance tasks. In this context, the overarching vision that we propose in this paper is a paradigm shift that will transform software testing and analysis in activities performed throughout the lifetime of the software to improve it based on the way it is used.

Figure 1 provides a high-level depiction of this vision. *Software producers* who want to perform a quality-assurance task on their previously-released software perform an augmented version of the *in-house task* that leverages field data. A tool supporting the task communicates to *remote agents* on the user sites a set of *data collection directives* for the task at hand. When users run the software, they transparently run a version instrumented by the remote agent's *instrumenter*, which adds probes to the code for producing *runtime data*. These data are then processed by appropriate *runtime monitors*, and the resulting *processed data* is stored in a *remote repository* for later use by a module that implements the *remote task* (i.e., a part of the task that can be performed on the user platforms). Depending on the task performed and the data collected, the remote task may send back *field data* to the software producer's *local repository* and/or provide new data-collection directives to the instrumenter. Finally, the software producer leverages the collected field data to perform the in-house task and evolve the software under analysis based on the results of the task.

Note that this is just an intuitive and simplified representation that is meant to be general enough to encompass different solutions. It does not imply that the software under analysis resides on a single machine, nor that instrumentation will necessarily be used to collect field data. In fact, this overall vision represents a general scenario that can be instantiated in many different ways.

3. RESEARCH AGENDA

In this section we discuss a set of research directions that can help fulfill the vision presented in the previous section. They are mainly intended as a starting point for further discussion, rather than a comprehensive list.

Recording user executions in the field. Two of the main limitations of in-house testing and analysis are that fielded behaviors are difficult to foresee (when testing) and to reproduce (when debugging). Researchers have tried to overcome these issues by collecting and using partial information about program executions (e.g., [14, 20]). Collecting such information can help, but developers often discover what they need to observe during a program execution only incrementally. Ideally, we would like to be able to faithfully record and later reproduce at will user executions (or relevant parts of them). Although some research has been performed in this direction (e.g., [12, 13, 18, 24]), most existing approaches are either too computationally expensive to be used in the field or require a specialized hardware or OS. Overall, most existing approaches still have a long way to go before they can be used online on real user executions. Further research and novel approaches (e.g., approaches that leverage hardware mechanisms and virtual machines to perform efficient record replay [1]) could help bridge this gap and make these techniques practically viable.

Post-mortem dynamic analysis of field executions. If practical record replay techniques were available, developers could perform various kinds of dynamic analyses while replaying real user executions. Consider, for instance, memory error detection tools such as Valgrind's Memcheck [15]. These tools are used very successfully in-house to identify memory problems, but may miss problems that occur only in some specific configuration or for some specific runs. Unfortunately, the overhead imposed by these runtime memory-checking tools is too high for them to be usable on deployed software. However, they could be run on recorded user executions by leveraging free cycles on the user machines and allow for discovering memory issues that actually occur in the field and may affect the users. The results of the analysis could then be sent to the developers, and aggregated across user sites (e.g., to rank the problems found based on how widespread they are and the number of users they potentially affect).

Debugging of deployed applications. Imagine the common case of a program deployed in a large number of similar instances, used by a multitude of users, and executed on powerful computers that are mostly interconnected. Imagine now to be able to partition the program in subsystems and assign the recording of each subsystem to one or more user sites. In such a scenario, when a failure (e.g., a crash, an exception, the violation of an assertion) occurs at a given site in a subsystem that is being recorded at that site, the corresponding execution could be saved for later analysis. The recorded execution could then be sent to the developers, who would use it for traditional debugging. Although being able to debug failing user executions this way would be extremely useful in itself, we could try to go even further. Specifically, we could define (semi) automated debugging techniques that can be performed remotely, on the sites where the applications fail and send back to developers only relevant information about the potential faults.

User based regression testing. Recorded user executions could also be used to support and enhance testing during program evolution. A typical way to perform regression testing is to keep a

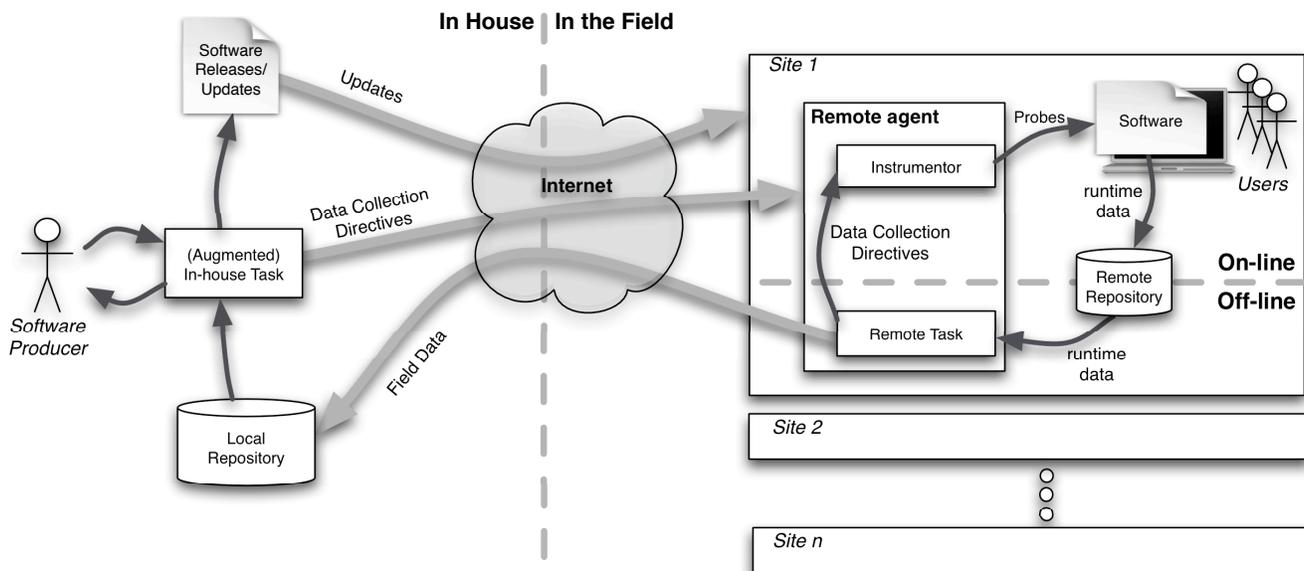


Figure 1: Overall vision.

regression test suite and rerun it on the changed program. The effectiveness of regression testing, like the effectiveness of testing in general, highly depends on how well the used test suite represents the way the program is used in the field. Unfortunately, regression test suites often exercise the application in a very different way than the actual users [19], which may result in unforeseen differences in behavior between old and new versions and, ultimately, dissatisfied users. Record-replay techniques could alleviate this problem by generating regression test cases based on user executions. These test cases would have the advantage of testing the software exactly in the way it is used in the field. In addition, subsystem and unit test cases could be extracted from these complete executions to have a set of faster and more focused test cases that would be more amenable to regression testing [7, 12].

Classifying program outcomes. When analyzing the dynamic behavior of deployed software, one fundamental aspect that is often overlooked is the ability to classify such behavior. For instance, it would not be possible to perform debugging without knowing when an execution fails or misbehaves. Many of the techniques that collect field data (e.g., [14, 22]) sidestep this problem by simply focusing on program crashes or analogous obvious failure manifestations. In practice, however, many failures do not result in a system crash, but rather in a wrong or anomalous outcome. Furthermore, passing and failing are not the only outcomes of interest for a software system. For a program that may fail in different ways, for instance, it would be useful to be able to classify an execution based on the kind of failure it manifests, rather than just classifying it as failing. In other cases, the outcome of interest may have nothing to do with a failure and could be, for example, the performance of the software. In general, many tasks would benefit from the ability of automatically classifying executions according to a given behavior of interest.

Predicting program outcomes. An even more ambitious goal than classifying user executions would be to build models of the behavior of the software that are good enough to allow for predicting

likely program outcomes. Such a capability could also greatly benefit a range of techniques. For instance, techniques for self adaptation and repair could leverage behavior prediction information to reconfigure a system when its performance is likely to decay or implement some corrective action when the software is likely to malfunction. This is a challenging research area that has triggered the interest of many researchers, who developed early solutions to subsets of the problem, or to related problems, using some form of machine learning (e.g., [9, 10, 26]). (The general idea behind these techniques is to train the learners on a set of executions whose characteristics are known and then use the learners to classify unknown executions.) Despite the existence of these techniques, many issues are still open, including the problems of how to get reliable training data, how to use the learning models online without affecting the user experience, and how to handle the difficult problem of false positives.

Efficient data collection. No matter what kind of field data gets collected, the amount of data collected can make an approach impractical. In some cases, the information collected is large (e.g., program traces). In other cases, the data may be compact, but the number of sites from which it is collected may be large. In the worst case, both situations may occur. There are some good solutions to this problem that are based on sampling (e.g., [14]), but there are types of data—and techniques that rely on such data—for which sampling is not appropriate (e.g., data used for path-sensitive analyses that may need to observe multiple elements, unknown a priori, along a path). For these cases, researchers should investigate different techniques to reduce the amount of data gathered and study the tradeoffs between amount of information collected and accuracy of the results. This is another crucial problem that must be solved to make techniques that leverage field data practically viable.

Multi-version data collection. Most data-collection techniques work only on single versions. The inability to handle data coming from multiple versions effectively (i.e., without keeping a separate database for each version) can considerably limit the effective-

ness of analyses performed on this data. Therefore, one relevant research avenue involves the definition of ways in which data coming from different versions can be used together, at least in part, and analysis results computed for an earlier version do not need to be recomputed from scratch when a new version is deployed. Note that this is a more general problem, which is present also when the supporting data is collected in house and not in the field. However, it becomes considerably more relevant in this context, in which the collection of data is more problematic.

Improving testing suites based on real usage. The typical way in which test suites are developed is that an initial version is built before the first release. Then, when users report problems with the software, the test suite is extended to include the relevant, and previously ignored, cases revealed by the users. Developers could improve this process by collecting data on the way a program is actually used in the field and leveraging this data to (1) assess whether an existing test suite is representative of actual usage and (2) modify it accordingly. These techniques could go from simple collection and comparison of coverage information [21] to very sophisticated statistical analysis of various program spectra. Being able to perform this type of early steering of test suites could help discover potential issues before they occur on the user machines, with obvious advantages for both users and developers.

Ensuring user privacy and security. Although techniques that collect field data to support in-house software engineering tasks represent an increasingly active area of research, privacy and security concerns have prevented widespread adoption of many of these techniques and limited their usefulness. One way to address this issue is to limit the kind and amount of data collected. Typical examples of this approach are crash reporting systems (*e.g.*, [2, 16, 17]), which collect information about stack traces, registry values, and environment at the time of a crash. Although this information can be useful to correlate different failures and perform a first investigation, it is often too limited. In fact, recent research has shown that the effectiveness of techniques that relies on field data increases when they can leverage more and more detailed information, such as complete execution recordings [1, 5] or path profiles [4, 11]. Unfortunately, this kind of detailed information is bound to contain sensitive data that users would likely be unwilling to share. Therefore, in order for techniques that leverage field data to become widely adopted and achieve their full potential, new approaches for addressing privacy and security concerns must be developed.

One possible way to address this issue would be to use privacy-preserving techniques for anonymizing field data. These techniques have been used successfully on databases [25] and may be tailored to work in this context. Another alternative would be to perform most of the analysis of field data on the user machines, and collect only the final results of the analysis. For some analyses, such as the memory leak detection technique that we described above, the information collected would be highly unlikely to reveal confidential information. Finally, one specific solution to the privacy problem, in the context of execution recording techniques, is to anonymize the inputs of recorded user execution so that they (1) are as different as possible from the ones used by the user, and (2) reproduce the same behavior as the original execution (*e.g.*, [3, 6]).

Development of a general support infrastructure. Building an environment such as the one depicted in Figure 1 will require the development of support tools and infrastructure. Moreover, ideally, these tools and infrastructure should be developed so that they are generic enough to support future techniques with differ-

ent data collection and analysis requirements. Although this task may appear as mostly an engineering effort, this is not necessarily the case. Based on our experience—and on the experience of other researchers and collaborators—we believe that building such an infrastructure will involve a number of new and difficult systems issue and the investigation of new, non-trivial solutions. This is especially true if the goal is to create tools and infrastructure that are designed to be adaptable and extensible, so that they can benefit the broader research community.

Web applications. Although field data collection and analysis are applicable in general, there is a golden opportunity that we can exploit in the case of web applications. Because of their nature, web applications simplify dramatically some of the issues that we listed above. For example, it is almost straightforward to replay the server side of a web applications, as most of its inputs are logged. Also, the code on the client side of a web application generally gets reloaded every time the user (re)opens the page containing that code. It is therefore simple not only to update the instrumentation in an application, but also to provide different instrumented versions (or even different versions altogether) to different users. This aspect in particular can open the door to advanced techniques that were not conceivable before, such as techniques that perform regression testing by pre-releasing a new version, transparently, to a small subset of users and collect information on the performance of that version to guide a broader distribution. For these reasons, web applications are an ideal target for the techniques discussed in this paper.

4. CONCLUSION

We believe that the collection and analysis of field data—runtime data about deployed software—can enable a disruptive leap forward in software testing and analysis research and practice. Although there is considerable interest in this area, both in the research community and among practitioners, existing approaches are still preliminary and fail to achieve the full potential of techniques that leverage field data. In this paper, we presented our vision and a research agenda that can help fulfill this vision.

Acknowledgments

This work was supported in part by NSF awards CCF-0916605 and CCF-0725202 to Georgia Tech.

5. REFERENCES

- [1] Replay-Based Debugging. <http://communities.vmware.com/community/vmtn/general/guestdebugmonitor>, 2010.
- [2] Apple Crash Reporter, 2010. <http://developer.apple.com/technotes/tn2004/tn2123.html>.
- [3] M. Castro, M. Costa, and J.-P. Martin. Better Bug Reporting with Better Privacy. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–328, 2008.
- [4] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44, 2009.
- [5] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *Proceedings of the 29th IEEE and ACM SIGSOFT International Conference on Software Engineering*, pages 261–270, 2007.

- [6] J. Clause and A. Orso. CAMOUFLAGE: Automated Sanitization of Field Data. Technical Report Technical Report GIT-CERCS-09-14, Georgia Tech, 2009.
- [7] S. Elbaum, H. N. Chin, M. Dwyer, and J. Dokulil. Carving Differential Unit Test Cases from System Test Cases. In *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2006)*, 2006.
- [8] S. Elbaum and M. Diep. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.
- [9] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-based methods for classifying software failures. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 451–462, November 2004.
- [10] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying Classification Techniques to Remotely-Collected Program Execution Data. In *Proc. of the European Software Engineering Conf. and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 146–155, september 2005.
- [11] L. Jiang and Z. Su. Context-aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 184–193, 2007.
- [12] S. Joshi and A. Orso. SCARPE: A Technique and Tool for Selective Record and Replay of Program Executions. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, October 2007.
- [13] S. King, G. Dunlap, and P. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Usenix Annual Technical Conference*, pages 1–15, Anaheim, CA, April 2005.
- [14] B. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, 2004.
- [15] Memcheck: A Memory Error Detector. <http://valgrind.org/docs/manual/mc-manual.html>, 2010.
- [16] Microsoft online crash analysis, 2010. <http://oca.microsoft.com/en/duf20general.asp>.
- [17] Mozilla Quality Feedback Agent, 2010. <http://www.mozilla.org/quality/qfa.html>.
- [18] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32th Annual International Symposium on Computer Architecture (ISCA-32)*, June 2005.
- [19] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European Software Engineering Conference and 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, pages 128–137, September 2003.
- [20] A. Orso, J. A. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proceedings of the ACM symposium on Software Visualization (SOFTVIS 2003)*, pages 67–76, June 2003.
- [21] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 99)*, pages 277–284, May 1999.
- [22] A. Podgurski, D. Leon, P. Francis, W. Masri, M. M. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 465–474, May 2003.
- [23] A. Porter, C. Yilmaz, A. M. Memon, D. C. Schmidt, and B. Natarajan. Skoll: A process and infrastructure for distributed continuous quality assurance. *IEEE Transactions on Software Engineering*, 33(8):510–525, 2007.
- [24] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, pages 158–167, 2000.
- [25] V. S. Verykios, E. Bertino, I. N. Fovino, L. P. Provenza, Y. Saygin, and Y. Theodoridis. State-of-the-art in Privacy Preserving Data Mining. *ACM SIGMOD Record*, 33(1):50–57, 2004.
- [26] C. Yilmaz, A. S. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan. Main Effects Screening: a Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems. In *Proceedings of ICSE 05*, pages 293–302, May 2005.