MonDe: Safe Updating through Monitored Deployment of New Component Versions

Jonathan Cook

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
jcook@cs.nmsu.edu

Alessandro Orso

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
orso@cc.gatech.edu

ABSTRACT

Safely updating software at remote sites is a cautious balance of enabling new functionality and avoiding adverse effects on existing functionality. A useful first step in this process would be to evaluate the performance of a new version of a component on the current workload before enabling its functionality. This step would let the engineers assess the component's performance over more (and more realistic) data points than by simply performing regression testing in-house.

In this paper we propose to evaluate the performance of a new version of a component by (1) deploying it to remote sites, (2) running it in a controlled environment with the actual workloads being generated at that site, and (3) reporting the results back to the development engineers. Running the new version can either be done on-line, alongside the current system, or offline, using capture-replay techniques. By running at the remote site and reporting concise results, issues of data security, protection, and confidentiality are diminished, yet the new version can be evaluated on real workloads.

1. INTRODUCTION

In the present era, frequent software updates at remote sites are becoming the norm, as security patches are released for immediate use and other bug fixes or feature enhancements are deployed. However, safely updating software at remote sites is a cautious balance of enabling new functionality and avoiding adverse effects on existing functionality. Currently, assuring that updates do not break existing functionality is done with extensive (regression) testing before deployment, hoping that the test cases are representative of the way the software will be used in the field. Unfortunately, as some of our previous work shows [6], behavior in the field often differs from the behavior exercised in-house using synthetic workloads.

PASTE '05 Lisbon, Portugal

We believe that, in conjunction with testing, a useful step in the software update deployment process would be to evaluate the performance of the update at the deployment sites before actually enabling its functionality. We define an update as the installation of a new version of a component or set of components. Executing the new version on-site will exercise the component with real user workloads and will allow the engineers the opportunity to view its performance over more (and more realistic) test data points than in-lab regression testing does. The use of multiple sites will also make the approach more effective and efficient, due to the possibility of splitting the evaluation among multiple sites and exercising the version over different user workloads.

Therefore, in this paper we propose a novel approach, that we call *Monitored Deployment (MonDe)*, for performing safe software updates at remote sites. MonDe evaluates the performance of a new version of a component by (1) deploying it to remote sites, (2) running it in a controlled environment with the actual workloads being generated at that site, and (3) reporting the results back to the development engineers. By running at the remote sites and reporting concise results, issues of data security, protection, and confidentiality are diminished because the workload data stays at the customer site. Nevertheless, the new version can be evaluated on real workloads. Since the new version is not influencing the system behavior, this monitored deployment does not need to wait for the typically expensive and time-consuming regression testing that precedes full component deployment. Instead, in-house testing and testing in the field can overlap, thus providing greater assurance that, when the new version is functionally deployed, it will behave correctly in the users' environment.

We envision two ways of running the new version of the component in the users' environment. The component can be run either on-line, alongside the current system, or offline, using capture-replay techniques. Both approaches are discussed in this paper. On-line execution is presented in the context of a framework supporting compiled code (e.g., C code) and shared libraries as deployable components, while capture-replay techniques are presented in the context of Java programs, in which each class can be a deployable component. Each system has its capabilities and limitations, which are also discussed in the paper. The contribution of this paper is twofold: (1) it presents the basic idea of remote monitoring of controlled deployment of software updates into the users' environment; and (2) it provides examples of frameworks that can support this approach.

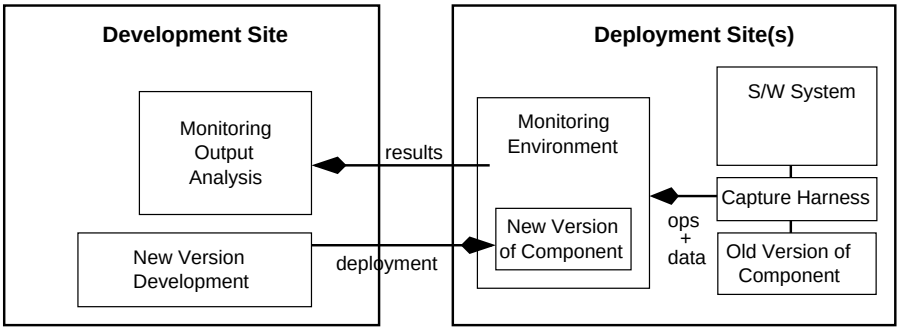

Figure 1: Remote deployment and monitoring architecture.

2. GENERIC FRAMEWORK

The generic framework architecture is shown in Figure 1. At the development site, new versions of components are created and deployed, and remote data about component performance is evaluated and analyzed (this paper will not discuss the output analysis in depth, but rather focuses on deployment-site frameworks). At the deployment site, two capabilities are needed:

- A capture capability, for recording the interactions of the old version of the component with the existing system, thus collecting the operational workload of the component for that site.
- An execution and monitoring capability that uses the collected operational workload to drive the new version of the component in a sandboxed environment where it can be observed.

As stated before, we do not make a restriction as to whether this capture and execution must proceed in parallel or the two can be separated in time. We detail both approaches in the Sections 3 and 4.

2.1 Selecting Part of the System to Monitor

One possible way to decide which parts of the system to monitor after new components are deployed is simply to select for monitoring each changed component. However, this straightforward approach has two main drawbacks.

First, it may result in a large number of false positives in cases in which a change (and its effects) are spread over several components. To illustrate, consider the example of two functions `foo` and `bar` defined in two different components. Function `foo` inputs a string and returns that string in lower case and without leading and trailing white spaces. To this end, `foo` leverages `bar`'s functionality. In version n of the components, `foo` calls `bar` to eliminate white spaces and then performs the transformation to lower case, whereas in version $n+1$, `bar` performs both the transformation and the trimming (e.g., for efficiency reasons) and `foo` simply acts as a proxy. In such a case, monitoring all changed functions (i.e., `foo` and `bar`) independently would erroneously report problems in `bar`'s behavior for every call that involves a string with at least a capital letter (e.g., $\texttt{bar}_n$(" StRing") would return "StRing", whereas $\texttt{bar}_{n+1}$(" StRing") would return "string").

Second, monitoring every changed component independently may impose unnecessary overhead on the users' executions. Consider again the example of functions `foo` and `bar` discussed above. If `foo` is the only function calling **bar**, then there is no need for monitoring `bar` independently because its changed behavior would be exercised through interactions of the rest of the application with `foo`.

The approach that we propose to alleviate these two problems is based on an analysis of the applications that use the modified components. At this initial stage of the research, we are considering an analysis of the calling relationships within the application. The analysis would be performed on the users' site and would consist of four main steps. For ease of presentation, we illustrate the analysis for the case of a single application using the modified components. Also, we discuss the analysis at the function level, but it could be adapted to operate at different levels of abstraction (e.g., the class level, for Java applications).

1. Construct a conservative call graph[1] for the application that uses the modified component(s).
2. Mark, in the call graph, the nodes that correspond to changed components.
3. Identify, in the call graph, all hammocks[2] that contain only changed components. (Note that such hammocks may contain one or more components.)
4. For each changed component, select the largest hammock that contains it, that is, the hammock with the largest number of nodes.

After these four steps are performed, each selected hammock identifies the boundaries of a part of the deployed system that our technique must monitor. Because the largest hammock is selected, intermediate unchanged functions, if their use is totally enclosed, would not cause inefficient splitting of hammocks. Figure 2 shows an example application of the technique for a system that consists of seven functions, three of which are modified. In the figure, the changed functions are colored in yellow and the selected hammocks are depicted using dashed ovals. For this example, the analysis would identify two subsets of the application for our technique to monitor—our technique would need to monitor all interactions between function `chgd2` and the rest of the system and all interactions between function `chgd1` and the rest of the system. In the next two sections, we discuss how our technique captures such interactions for Java and C programs.

[1]A *call graph* is a directed graph in which nodes represent functions and edges represent call relationships between functions.

[2]A *hammock* is a single-entry, single-exit subgraph.

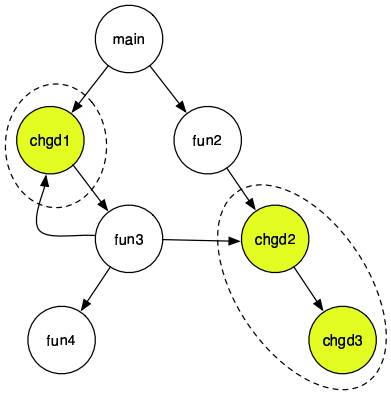

Figure 2: Example of call graph with hammocks.

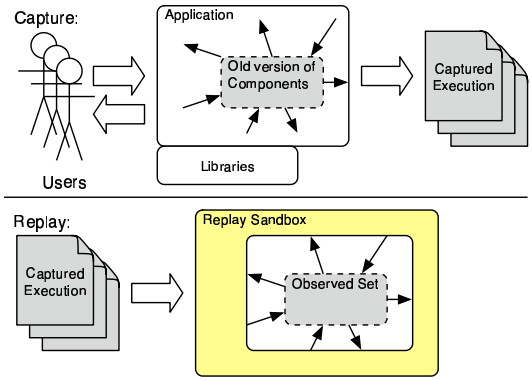

Figure 3: Capture and replay of deployed components.

3. JAVA CAPTURE AND REPLAY

One possible way of running the new version of the component in the users' environment is to use off-line techniques based on capture-replay mechanisms. We present this approach in the context of Java programs. to implement the approach, we leverage a technique and a tool previously developed by one of the authors [7]. The tool is called SCARPE and performs Selective CApture and Replay of Program Executions for Java software.

Given an application, SCARPE lets one (1) select a subsystem of interest, (2) capture at runtime all the interactions between such subsystem and the rest of the application, and (3) replay the recorded interactions on the subsystem in isolation. SCARPE is designed to be efficient: for each execution, it only captures information that is relevant to that execution. To this end, it disregards all data that, although flowing through the boundary of the subsystem of interest, do not affect its execution. Intuitively, our technique captures only the minimal subset of the application's state and environment required to replay the execution considered on the selected subsystem.

In the context of the MonDe approach, the subsystem of interest would consist of one or more components (classes) identified as discussed in Section 2.1. We would use SCARPE to capture users' executions of such subsystem that would then be replayed on the new version, as illustrated in Figure 3, when free cycles are available.

In the replay phase, the technique automatically provides a *replay sandbox*. The replay sandbox inputs the captured executions, in the form of event logs,[3] and replays each event in the log by acting as both a driver and a stub. Replaying an event corresponds to either performing an action on the observed set (e.g., writing an observed field) or consuming an action from the observed set (e.g., receiving a method invocation originally targeted to external code). Based on the event log, the replay scaffolding is able to generate and consume appropriate actions, so that during replay the right classes are created and the interactions among these classes are reproduced.

Because the events captured also include events generated in the system of interest, such as return of values, SCARPE can be used as an oracle when replaying the captured executions. If the behavior of the new component(s) is observably different from the behavior of the current component(s), the problem can be reported to the developers. It may also be possible to report the (sanitized) captured execution that generated the problem. In this way, developers not only are notified, but can also reproduce and investigate the problem.

One open issue that we will have to address is how to handle situations in which the subset of the application state accessed by the updated components is different from the subset that was captured. (As stated above, we capture only a subset of the application's state to improve performance.) To address this issue, we will investigate ways to extend the amount of information captured and to balance the resulting trade-offs between efficiency and effectiveness. For example, we are already capturing complete objects of some specific classes, such as `String`.

4. SHARED LIBRARY PARALLEL EXECUTION

A second method for running the new component version is to run it on-line alongside the existing component version. This technique will work well if the performance overhead does not severely impact the application, if the amount of interaction data to capture would be too much to store efficiently, or if the environment is not easily amenable to full capture yet the component interactions can be intercepted. Compiled applications (e.g., C programs) deployed using shared (dynamic link) libraries are able to utilize this on-line approach.

One of the authors has built DDL [9], an extended dynamic linker that opens up the dynamic linking process and provides programmatic control over it. DDL provides the basic capability needed to build program monitoring and manipulation tools. The two basic capabilities that DDL supports are an informative one, where DDL informs a tool of the bindings that are taking place, and a manipulative one, where DDL allows a tool to redirect a binding to a different symbol.

In previous work, we have already used DDL to perform software updates in which multiple component versions are run in parallel and allowed to immediately affect the application [1]. In this work, we reuse that basic infrastructure to support the MonDE approach by running the new compo-

[3]Events correspond to the various forms of interactions between the subsystem of interest and rest of the application, such as method calls, access to field, and exceptions.

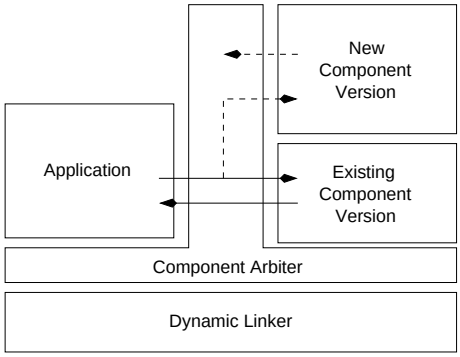

Figure 4: Shared library-based parallel version execution.

nent version in an "evaluation mode" that prevents it from affecting the overall application.

Figure 4 shows the overall architecture of this approach. A new component version is loaded into the application space, but DDL-controlled linking isolates it within control of the component arbiter. The arbiter intercepts interactions with the existing version of the component and duplicates them for the new version. Results and external effects of the new version are controlled by the arbiter, and either logged for later delivery to the remote development site or compared with the existing version at run time for aggregation of results that will be sent to the development site.

In this approach, a component is a shared object. Our previous work restricts the component to be a collection of C functions or a C++ class (and its objects) that obeys the ideas and rules of component-based design: external data is not directly modified but is only done so through accessor methods or functions external to the shared object.

Our approach is different than the capture and replay approach taken with Java in that the new version is executed in parallel with the application and the old version. Under the assumptions above, this approach is easier than capture-replay because the compiled-code environment is more opaque. By running the new version in parallel with the application, it can have read-only access to application data as it needs, without having to record those accesses.

For outgoing calls that modify state, we intercept these and can do one of two things. Firstly, we can return the call without actually invoking the state-modifying code, and thus disable external state modification. This approach will work as long as the state that is maintained by the old version and the application is all that the new version needs. If the new version needs new state maintained by another component, then the *hammock* idea presented earlier can be extended to encapsulate that component with the other, so that they are updated together. Special consideration will also be needed for large stateful components, such as databases. In this case, if the new version needs unique state modifications, then a special stub wrapper would be used to isolate its state changes and yet keep them and make them available back to the new version when needed.

5. CONCLUSION

We have proposed an approach for verifying a new version of one or more components using field data and at the field site. The approach, that we call Monitored Deployment (MonDE), works by deploying the component version and running it either on-line, in parallel with the application, or later, in a replay environment using previously captured interaction data. In either case, the new version is executed on actual field data but is prevented from affecting the system. Performance data is then sent back to the development site for evaluation by the engineers. Such an approach can assuage the concern of allowing real user data to be captured and delivered off-site, while still enabling field testing of the new version.

Related work has looked at executing component versions or variants within a system, yet not for pre-functional deployment evaluation in end-user environments [1, 2, 4, 5]. Similar execution encapsulation mechanisms have been used for execution of mobile and untrusted code, intrusion detection, and other security issues (e.g., [3]). Other approaches in verifying new component versions have also been investigated (e.g., [8]).

Acknowledgments

This work was supported in part by NSF under grants CCR-0306457, EIA-9810732, and EIA-0220590 to New Mexico State University and grants CCR-0205422, CCR-0306372, and CCR-0209322 to Georgia Tech.

6. REFERENCES

[1] N. Abbas, J. Cook, and S. Tambe. Reliable Runtime Upgrading of Binary C++ Classes. Technical report, New Mexico State University.

[2] V. Kharchenko, P. Popov, and A. Romanovsky. On Dependability of Composite Web Services with Components Upgraded Online. In *Proc. 2004 DSN Workshop on Architecting Dependable Systems*, June 2004.

[3] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, FL, 1994. IEEE Computer Society Press.

[4] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proc. European Conference on Object-Oriented Programming*, pages 337–361, 2000.

[5] A. Mos and J. Murphy. COMPAS: Adaptive Performance Monitoring of Component-Based Systems. In *Proc. 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems*, May 2004.

[6] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of the 9th European Software Engineering Conference and 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–137, Helsinki, Finland, September 2003.

[7] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Online Proc. of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, St. Louis, MO, USA, May 2005. http://www.csd.uwo.ca/woda2005/proceedings.html.

[8] A. Podgurski and E. J. Weyuker. Re-estimation of Software Reliability after Maintenance. In *Proc. 19th International Conference on Software Engineering*, pages 79–85, New York, NY, USA, 1997. ACM Press.

[9] S. Tambe, N. Vedagiri, N. Abbas, and J. Cook. DDL: Extending Dynamic Linking for Program Customization, Analysis, and Evolution. In *Proc. International Conference on Software Maintenance*, Budapest, Hungary, September 2005. to appear.