Aspect-based Introspection and Change Analysis for Evolving Programs

Kevin Hoffman, Murali Krishna Ramanathan, Patrick Eugster, and Suresh Jagannathan
Computer Science Department, Purdue University
{kjhoffma,rmk,peugster,suresh}@cs.purdue.edu

***Abstract*— As new versions of software are developed bugs inevitably arise either due to regression or new functionality. Challenges arise in discovering, managing, and testing the impact of changes on software. These challenges are magnified in software systems that evolve, because the new functionality is piece-wise introduced into a live program with prior state produced by the prior component versions. If new functionality introduced into a live system induces bugs, it can be extremely difficult to analyze at run-time exactly which differences led to the incorrect behavior.**

In order to help programmers plan for evolution, understand the impact of specific evolutionary steps, and to diagnose evolution gone wrong, herein we propose combining the benefits of Aspect-Oriented Programming and reflection with impact analysis techniques from the OO and software engineering disciplines.

We contribute a tool that assists with the deployment of new code to evolving software that gives insight as to precisely the behavioral changes between the new code and the code it is replacing within the running system.

This tool is implemented using pure aspect-oriented and reflection techniques, and we discuss how to combine this tool with a load-time aspect weaver to allow precise determination of the cause of bugs introduced in live, evolving systems. We conclude by considering the challenges of implementing and deploying such a tool and outline our plans for future research and evaluation.

I. Introduction

Evolving software systems are becoming ever more prevalant and important, especially considering the rise of highly-available service-oriented architectures. The rate of change of software systems has been accelerated by deployment of web-based applications, which are expected to run without interruption and with full reliability, notwithstanding their high rate of change and evolution.

The challenges caused by combining high-availability and rapid evolution in the same system are daunting. Techniques and tools to mitigate and overcome such challenges is a topic of intense research.

One such challenge linked to rapidly evolving software is the problem of determining the impact of specific changes to source code – how will these changes propagate and influence the rest of the system? This question is well-known and well-studied in the literature [1]–[4]. Techniques have emerged for answering this question with meaningful precision for software that does not change at runtime.

One such class of techniques is known as dynamic impact analysis, which strives to improve accuracy over the static analysis of source code by employing code instrumentation and post-mortem analysis. Code is instrumented to generate trace data at runtime, both versions of the program are then run through one or more test cases, and finally these traces are compared and analyzed in order to infer the impact of the changes with as much precision as possible.

However, in their present state these techniques are difficult to apply to systems that evolve at runtime. While they can be used to understand how two versions differ across some common test cases, they cannot currently be used to discover the influence of new code on the live system where the new code is influenced by the behavior and state of the old code. Another challenge added by live evolution is that there is no beginning or end to the program where traces are typically endpointed, so it is difficult to define endpoints without scattering code or tags throughout the target program.

The contributions of this paper are as follows: First, we present how to combine aspect-oriented programming and reflection to transparently instrument programs while providing flexibility in how this instrumentation is applied and when it is active. Second, we enhance analysis techniques first introduced in [5] that allow the programmer to analyze the traces to achieve precise change analysis. Third, we evaluate the performance overhead of using our tool in long running (evolving) systems by measuring the performance impact both when tracing is and is not active.

II. Technique Overview

The general strategy of our approach is to first use aspects to instrument programs so that they generate ”interesting” trace data (using the AspectJ 5 load time weaver), and then to analyze these traces (collected from multiple versions (or evolutionary steps) of the program) in order to determine which changes in the source code were responsible for the exhibited changes in program behavior.

We build on the insights and techniques of the Sieve [5] dynamic impact analysis system, but adapt it for object-oriented programming, change the nature of the trace data, and add flexibility and power stemming from implementing the instrumentation via AOP.

A. Trace Generation

The traces generated by the instrumentation approximate a *complete program trace* – a complete, sequential log of every low-level instruction executed (kept on a per thread basis) – in the same way that static analyses approximate the run-time behavior of a program. Greater accuracy over static analysis can be achieved because the traces have access to actual program state and control flow as the program executes.

However, while capturing a complete program trace would be feasible (e.g. by modifying the JVM) and would preserve all information for the analysis, the size of these traces would quickly become challenging for long-running programs and would also cause analysis to be inefficient or even intractable.

To approximate a complete program trace, we use the following strategy: First, the program trace is endpointed so that only pieces of the execution trace are instrumented and recorded. For example, it could use the entrance and exit to a certain high level method as one set of endpoints. Each segment of the complete program trace captured between endpoints is called a trace segment.

For each trace segment several individual traces (representing an approximation of what happened over time) are generated over the lifetime of the trace segment. One individual trace, termed a trace point, is captured for each unique (method, truncated call stack) pair. The same trace point may be active many times over the course of the trace segment (as the same method with the same truncated call stack can be called many times).

As trace events (field accesses and method calls) are captured by the instrumentation, the code first determines the current trace point (creating the individual trace for that trace point if it is the first time entering that trace point) and then adds to the individual trace in the following way. First, information about the event is fed into a hashing function, and this hash value is used as a key into the individual trace (represented using an ordered hash table). If the individual trace does not contain the hash key, then the event is appended to the end of the trace along with a counter initialized to 0. It also includes relevant information about the event such as the source code location corresponding to that event. If the trace already contained the hash key, then the counter associated with that hash key is incremented.

In this way the trace approximates the sequential execution of events during all executions occurring in that trace point during the current trace segment. The use of event hashing and not storing more than one trace entry per hash allows traces to remain small, while the counter helps to model which branches were taken and how many loop iterations were executed.

B. Trace Analysis

The analysis of the trace data builds on the technique introduced by the Sieve [5] system.

To understand what has changed between different versions of the (running) program, the following procedure is used: First, one or more trace segments are captured on both the old and new versions of the running program. Next, the program identifies for each individual trace in the new version the corresponding individual trace in the old program. Currently the tool uses method name and signature to pair up traces, but more complex heuristics would be needed if method signatures or class names change between versions (as might be the case with an evolving system).

It then computes the minimal differences in execution between each new and old trace point by computing the longest common subsequence algorithm. Each individual trace is represented as a sequence of trace events, so the longest common subsequence between two traces represents the execution events that occurred in both versions of the program. The LCS can then be used to determine the minimal set of execution events that either did not occur in the new version or were new to the new version.

The result of the analysis can be used by developers to understand the precise effects new component versions have upon program execution. If the input traces were complete program traces, then the results of the analysis would present precisely the points in both time and code at which the behavior of the new version diverged from what the old version would have done. In the trace approximations described above all temporal notions of computation are not recorded (aside from capturing the sequence in which execution events are first encountered within a given trace point), so the results of the analysis represent the precise points in the source code where the execution differed, but do not provide insight into the time of the divergence or how differences observed in different trace points can be ordered with respect to each other.

The traces can be enhanced in several ways to increase the precision of the analysis. First, the definition of a trace point can be extended from (method, truncated call stack) to include additional context, such as actual values of parameters or the number of times the method was called (allowing the capture of different traces for both the first N and last M executions for any given (method, truncated call stack)). Due to our use of aspects and reflection to implement trace generation, all of the above enhancements could be parameterized and adjusted at run-time – an important factor for long running, evolving systems. This parameterization could be used to increase the accuracy of the generated traces where it is anticipated that the increased accuracy is needed, while still avoiding full generation of the complete program trace (unless this is desired as indicated by the parameters).

C. Example

We present a small example in this section to illustrate how the tracing process works. Consider the two versions of some program shown in Figure 1. The only difference between the two versions is on line 9. For this example a trace point is identified solely by the method name, a location solely identified by a line number, the trace event data includes the values of the parameters, and the hash of a trace event is the hash of the method name and parameter values. We define one trace endpoint using the pointcut `execution(void C.m1(..))`.

Figure 2 shows the individual traces that would be generated for the trace segment that became active as part of the `C.m1` method execution. Each line in a trace consists of the following format: `hash.counter: line: event data`. This example exhibits many illustrative points: First, note that the execution between the two versions would first begin to differ at line 9 – in version A the call to `m3` is made on line 10 and then again on line 12, whereas in version B this call

```
         (Version A)                        (Version B)
 1) class C {                       1) class C {
 2)  void m1(){                     2)  void m1(){
 3)    out.println("p1");           3)    out.println("p1");
 4)    m2(true);                    4)    m2(true);
 5)    out.println("p2");           5)    out.println("p2");
 6)  }                              6)  }
 7)  void m2(boolean b){            7)  void m2(boolean b){
 8)    out.println("p3");           8)    out.println("p3");
 9)    if (b)                       9)    if (!b)
10)      m3();                     10)      m3();
11)    out.println("p4");          11)    out.println("p4");
12)    m3();                       12)    m3();
13)    m4();                       13)    m4();
14)  }                             14)  }
15)  void m3(){                    15)  void m3(){
16)    out.println("p5");          16)    out.println("p5");
17)  }                             17)  }
18)  void m4(){}                   18)  void m4(){}
19) }                              19) }
```

Fig. 1. Two versions of a program to be traced

```
======== C.m1.txt ======        ======== C.m1.txt ======
81a9db.1:  3: println p1        81a9db.1:  3: println p1
fc3511.1:  4: m2 true           fc3511.1:  4: m2 true
d8247b.1:  5: println p2        d8247b.1:  5: println p2

======== C.m2.txt ======        ======== C.m2.txt ======
56b1b1.1:  8: println p3        56b1b1.1:  8: println p3
5b0e30.2: 10: m3                7f4bdb.1: 11: println p4
7f4bdb.1: 11: println p4        5b0e30.1: 12: m3
42eea0.1: 13: m4                42eea0.1: 13: m4

======== C.m3.txt ======        ======== C.m3.txt ======
5533c9.2: 16: println p5        5533c9.1: 16: println p5
```

Fig. 2. Individual traces generated for example program in Figure 1

is only made on line 12. This behavior is exhibited by the traces. The trace for `C.m2` for version A shows `m3` was called before `println("p4")` and that it was called one more time sometime after that. The trace for `C.m2` for version B shows `m3` was called after `println("p4")` and that it was not called again after that. Second, when the LCS is generated between the two versions it would show that the execution diverged after line 8 and then converged back at line 13.

Adjusting the definition of trace point identity and the inputs used in the hash for events, and also allowing timestamps to be used allows programmers to explore changes with varying levels of accuracy and efficiency.

III. Implementation

This section describes the role of aspect-oriented programming and reflection in implementing our techniques.

A. Tool Overview

The tool is composed of a collection of instrumentation aspects that generate trace data and a Java analysis program to extract meaning from the trace data. To run a program using the instrumentation framework a script is used in the place of the `java` command that invokes the AspectJ 5 load-time weaver and configures the instrumentation aspects according to parameters passed on the command line. Parameters can fine tune exactly which parts of the program are instrumented and are also used to adjust the accuracy of trace data (allowing tradeoffs to be made between performance and accuracy).

B. Aspectized Instrumentation

The generation of trace data is composed of four components: call stack tracking, trace event tracking, trace generation, and trace persistence.

For each thread an object representing the current threads call stack is maintained by an aspect. This aspect intercepts all method calls within the scope of interest using before and after advice and adds to or removes from the thread's call stack object. Each call stack entry records the information required to identify the current trace point (method name, signature, and possibly argument values). An aspect is used instead of a Java exception object to obtain the current call stack both because our technique is faster (the call stack is updated only as it is changed, instead of rebuilding it every time it is needed) and it allows us to gather more information about the call stack (such as the values of parameters to methods).

Events that are of interest to the trace (field accesses and method calls) are modeled by using a pointcut. This pointcut uses an `if` primitive pointcut so that it does not match when there is no active trace segment. A `before` advice uses the above pointcut to execute trace generation logic whenever an event of interest is about to be executed.

The logic to generate the traces works in the following fashion. When an event of interest is about to execute it generates the current trace point using the current call stack for the current thread and the individual trace associated with that trace point is looked up (or created if it does not exist). The data for the event is collected through reflection, and then a portion of this data is put through a hash function to generate a key. If the key already exists in the hash table that trace entry is 'revisited' to update its counter and timestamps. If the key does not exist, a new trace entry is added to the hash table (the hash table also records the order of the insertions to capture an approximate sequence of execution within the trace point).

When the end of the trace segment is reached the trace persistence component writes all trace data generated during the trace segment into a series of files and then deactivates tracing. One file is generated for each individual trace with the trace entries written in the order in which they were inserted into the hash table. Each trace entry written includes information relevant to the analysis, including the original source location responsible for that execution event. This persistence logic can easily be offloaded onto a background thread so as not to block the progress of the application thread.

C. Use of AspectJ Reflection

Reflection is used in the implementation primarily in two places. First, the call stack builder uses reflection features of `thisJoinPoint` to extract the source location of method calls. The call stack builder also uses the new reflection features in AspectJ 5 to determine whether or not the method is actually part of an aspect, which can be useful in the analysis stage to analyze the change in how aspects advised the base code between versions of a program.

Second, reflection is used in the advice that advises trace events to capture dynamic information about the event (e.g. values of method parameters) and also for supporting more complex definitions of trace points.

D. Trace Endpointing

Traditionally, the endpoints of a program trace are defined to be the start and end of a program. However, this is not useful for long running, evolving systems, so a means of defining endpoints with finer granularity is needed. Because aspects are used to perform the instrumentation, the full power of the AspectJ quantization model is available to the programmer in describing where trace segments should begin and end.

When the program is started, abstract pointcuts indicating the (possible) places where trace segments begin and end can be instantiated (by using an XML file which is used by the load-time weaver). The `cflow` and `cflowbelow` primitive pointcuts are then used with the above pointcuts to capture precisely the places where a trace segment begins and ends. The programmer can use dynamic primitive pointcuts in the endpoint pointcuts (e.g. `if`) so that tracing is only activated if certain boolean variables are set. More complex activation logic could be added if required (similar to how logging frameworks allow classes of log messages to be turned on or off) at the cost of higher performance overhead.

Using pointcuts to implement endpointing allows for great flexibility in deciding the scope and precision of tracing operations. The tool could also be enhanced to allow for more than one trace segment per thread to be active at one time, although this would affect performance (whether or not the impact would be significant is yet to be determined).

E. Analysis of Trace Data

The analysis is implemented as a Java program that accepts as input the location of two directories, where each directory contains the results from one trace segment (for one particular version of the program). The analysis program also accepts a mapping that tells it how to correlate trace points in one version with the trace points in the other version (by default, this mapping is the identity function).

For each trace point it computes the longest common subsequence of trace events, using the hash value as a sequence element. The output is similar to the common `diff` tool in that it shows which trace events were exhibited in the old version but not the not version and vice versa. The tool assists the user by parsing the rest of the trace data associated with each trace event (source code information) and displays this decoded information to the user. In this way the user can see the source locations where the executions differed. If trace points were defined using a larger context and trace data contained temporal information, as discussed in section II, then the resulting output would also give insight into *when* the new version first diverged from the old version in addition to information about *where* it did so.

IV. Discussion

In this section we discuss how the techniques described in this paper relate to software evolution and also discuss the performance impact of using these tools in deployed systems.

A. Benefits for Software Evolution

The techniques presented in this paper are beneficial for evolving software systems in the following ways:

1) Java software can be transparently instrumented and the behavioral differences between different versions of objects (or similar objects) can be precisely observed.
2) The use of pointcuts to describe trace endpoints gives a great deal of flexibility in dynamically deciding when and what to trace through the use of dynamic pointcuts, while still reaping optimizations made by the AspectJ weaver. For example, before upgrading the evolving system tracing could be turned on for some built in test cases, and then afterwards the traces generated using the new classes could be compared to the old traces to understand exactly what went wrong (or right).
3) AspectJ pointcuts allow developers to selectively specify what should be instrumented (improving efficiency) while still being robust to enhancements and changes to the type system in the future.

We anticipate collaboration with others in this field to further learn how these ideas can be applied.

B. Performance Implications

In order for these techinques to be most useful for evolving software they must be deployed on production systems. Production systems are much more sensitive to effects on performance, so it is important to understand the performance costs associated with the instrumentation and tracing.

Even when the instrumentation and tracing logic is woven into a program, there are different levels of tracing activity with different corresponding performance costs. The first level is where the code has been instrumented, but there is no active trace segment in *any* thread. In this level the performance cost is due to the dynamic `if` pointcut designator that checks before every method call the value of a boolean flag (in this level, the flag is false, and further logic is not executed). The next level is where there are active trace segments on other threads, but not in the current thread. In this situation the global tracing flag is true (meaning tracing is active somewhere), but the dynamic `if` pointcut designator has to also then check a hash table to see whether or not there is an active trace segment for the current thread (in this case, there is not). Finally, the highest level is where there is an active trace segment for the current thread, so trace data is being generated for every method call (or field access if desired).

We have designed and run two benchmarks to quantitatively understand the performance costs associated with each level of activity. The first one is a microbenchmark that is a program consisting almost exclusively of nested method calls (there are a few loops and addition operations, but the primary computation of the program is calling methods). This microbenchmark

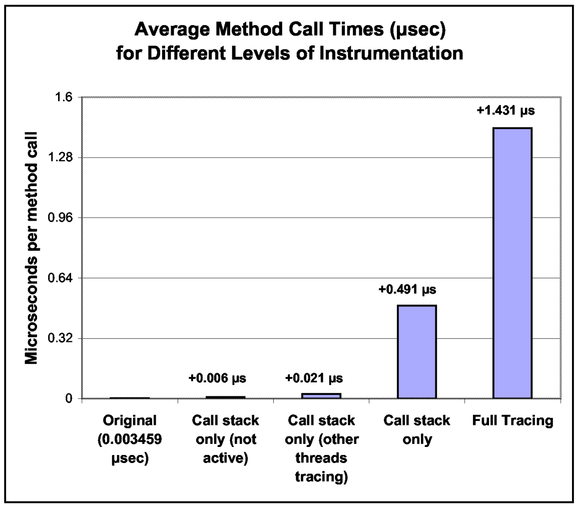

Fig. 3. Microbenchmark Results

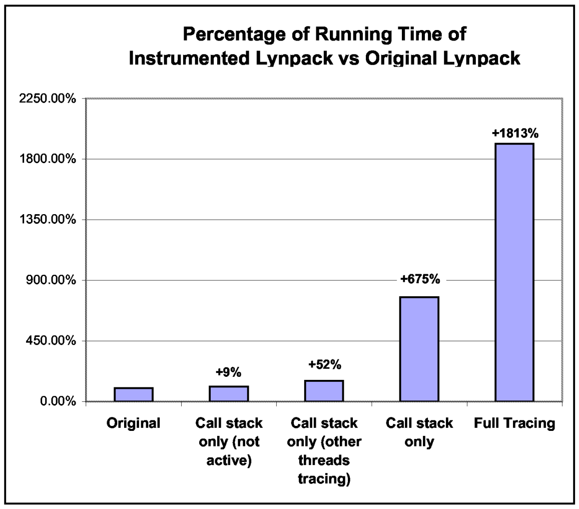

Fig. 4. Lynpack Benchmark Results

was designed to exhibit the actual timing overheads associated with the differing levels of instrumentation. The second benchmark is the Java Linpack benchmark [6], which heavily exercises the floating point processor while also involving a large number of method calls (over 100 million).The goal of this benchmark is to see how the overall performance of a computationally intensive program (involving a large number of method calls) is affected after it has been instrumented. Both of these benchmarks represent worst-case or near worst case scenarios for the instrumentation; applications in which the actual number of method calls is relatively small (e.g. applications built on middleware) will be affected much less.

All benchmark tests were performed on a machine with a 2.33 Ghz Intel core 2 Duo procesor, 2 GB RAM, running OS X. The programs were compiled and run using Sun's Java 1.5.0.07 (Server Hotspot VM) and AspectJ 1.5.3. Before the tests were run all applications were closed and the system left until it entered an idle state. All timings computed do not include the time to load the Java VM or perform weaving. All numbers presented are averages over three runs.

The results of the method calls microbenchmark are presented in Figure 3 showing the average method call times. In this test over seven million method calls were made under each of the different scenarios and then the average time to execute a method call was computed by dividing the wall clock time by the exact number of calls made. (For some of the faster tests hundreds of millions of method calls were made to ensure a stable result.) The results for the Lynpack benchmark as presented in Figure 4 show how much slower instrumented programs ran compared to the original programs.

Applications in production would be in the *call stack only (not active)* level most of the time. At this level the overhead due to code instrumention is 6.1 nanoseconds per method call. For almost all applications this overhead should be negligible. Some computationally intensive programs that make heavy use of method calls in their innermost loops may experience a slowdown similar to Lynpack (9%).

Given that the duration of trace segments should be relatively small (being inactive most of the time – becoming active before and after components change), especially compared to the duration during which there are no active trace segments, the higher overhead present during tracing should not pose a significant obstacle to adoption. Additionally, if performance is an issue, load time parameters can cause the instrumentation to not be applied to certain sections of code such that there is no overhead for these regions of code.

V. Conclusion

We have presented new techniques for understanding the changes in behavior of programs as they evolve (while running) over time. These techniques show promise to assist in planning, monitoring, and diagnosing evolving systems. There is certainly more to understand relating to the performance of the technique and how to improve it, but our results so far indicate that the overhead is an acceptable burden for most applications. Future work involves further developing and refining the methods for evolving software, applying the technique to one or more case studies, and developing tool and IDE support for easy integration into the development process.

References

[1] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *ICSE 03*, 2003, pp. 308–318.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *ICSE 05*, 2005, pp. 432–441.

[3] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *ESEC/FSE-11*, 2003, pp. 128–137.

[4] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *OOPSLA 04*, 2004, pp. 432–448.

[5] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A tool for automatically detecting variations across program versions," in *ASE 2006*, September 2006.

[6] J. Dongarra, R. Wade, and P. McMahan, "Linpack benchmark – Java version," 2007, http://www.netlib.org/benchmark/linpackjava/.