A Special-Purpose AOP Framework for Runtime Monitoring

Amjad Nusayr
University of Houston - Victoria
anusayr@cs.nmsu.edu

Jonathan Cook
New Mexico State University
joncook@nmsu.edu

Gholamali Rahnavard
New Mexico State University
rah@nmsu.edu

ABSTRACT

We present a design and initial prototype for a Teams, a new aspect oriented programming framework specifically for usage as an abstract instrumentation capability for runtime monitoring and dynamic analysis. Our goals for this framework are simplicity, extensibility, portability, and monitoring concept coverage. If successful, Teams will provide us and other researchers an easy-to-use platform for building instrumentation that will support their monitoring and analysis research.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging

General Terms
Measurement, Languages

Keywords
Runtime Monitoring, Aspect Oriented Programming

1. INTRODUCTION

For several years we have investigated the application of AOP (aspect oriented programming) to runtime monitoring instrumentation. Indeed if one looks at the success stories of AOP, many of them are analysis-based applications that use AOP weaving to instrument and monitor the underlying program they analyze. Examples include tracematches [6], race detection [5], monitor-oriented programming [8], and many others.

The drawback that existing AOP systems have is that none of them support all the levels of detail that dynamic analysis can require, and this limits the granularity of what a researcher can do with them. For example, virtually all AOP frameworks can only instrument down to the method level (call or execution) when it comes to code detail, and data instrumentation is usually very sparse (e.g., object fields but nothing else) or non-existent. We pointed this out in 2009 [13, 14] as we began a detailed investigation of the application of AOP to runtime monitoring, and others have noticed this before us (Rajan and Sullivan [15]) and as recently as last year [2]. Thus investigating the application of AOP ideas and mechanisms to the full spectrum of runtime monitoring needs is still an important research area.

We pursued extensions to the ideas of AOP that attempted to extend the capabilities of AOP to a much finer grained capability in code, and to add other dimensions of weaving capability that were previously ignored, such as probability, sampling, and time. We used the open framework abc [1], the AspectBench Compiler, that was designed to be extended by other researchers, and we settled in to experimenting with our ideas.

Unfortunately we soon hit hard limitations. While trying to debug why our attempt at statement-level weaving was not working, we finally received a message back from one abc developer essentially saying, “well, we never intended for abc to be used at the statement level.” We hit other limitations as well, and especially found that the compilation focus of abc severely precluded integrating our rather radical ideas about weaving over the time dimension. We had some successes, but finished the project wondering if there was any sense in continuing to use abc.

We are now attempting to build Teams, our own AOP framework, not for general purpose AOP usage, but in particular for using AOP as a high-level abstraction for performing instrumentation that will be used for monitoring and analysis. Teams stands for The Extensible Aspect-based Monitoring System. This short paper presents our initial ideas for the framework and presents some early progress in prototyping the framework.

2. DESIGN

Figure 1 shows the high-level design of Teams. Teams will be a lightweight cross-platform system for exploring AOP mechanisms, to be used in particular for runtime monitoring. Focusing on monitoring in particular will mean that we will concentrate on features that are observational and informational, and will not be concerned with AOP features that modify the behavior of the system. This greatly simplifies some of the standard AOP concerns such as conflicting advice behavior.

A key notion in our architecture is that the advice portion of an aspect will be written in a plain old programming language, or POPL for short. In many AOP systems advice is packaged into the aspect, which then requires the AOP...
framework to either have a full language compiler built in, or at least a partial grammar of the language in order to translate extra-language features inside the advice into the underlying programming language. We will instead rely on well-defined interfaces and standard naming conventions to connect advice written separately in the POPL with the pointcut expressions and aspect definitions written in our lightweight aspect language.

We will initially focus on a fully dynamic runtime engine that will perform load-time or run-time weaving, but as Figure 1 shows with the dotted lines, we can later integrate static weaving if the performance payoff is large and needed.

We plan on making our framework cross-platform in the sense of allowing the runtime portion to be replicable on different platforms. The pointcut compiler will be implemented only once, and will be responsible for checking the correctness and consistency of the pointcuts and reducing them to a simpler intermediate representation. This representation can be used by multiple runtime engines on different platforms; the only other portion that would need rewritten for each platform would be the actual advice for the aspect. One could imagine, for example, an aspect for def-use coverage analysis, where the pointcut expression(s) would not need to be changed for different platforms, but the advice implementation would be.

Many instantiations of this framework can be imagined, not only for different platforms but also for different purposes. For example, in a test environment setting where execution slowdown might not be an issue, we could use a very intrusive tool such as Pin [11] to provide very low-level joinpoint types and enable a user to create fine-grained run-time monitoring instrumentation at the abstract AOP level without worrying about how to program Pin itself.

Figure 2 shows an example of how an aspect might be specified in a Java-based version of TEAMS. The left side shows the pointcut expressions in our own Teams specification language, while the right shows the advice in plain Java code. We rely on naming conventions to connect the two sides. An aspect class in the POPL will be required to be named, e.g., AspectName, and perhaps be in a teams package scope, and the aspect pointcut definition will be required to have the same Name. Each pointcut expression will have a name, and advice associated with that pointcut expression will be a method named with the same name and prefixed by the execution mechanism (e.g., before).

2.1 Providing Joinpoint Information

A limitation of the advice-as-POPL approach is that one cannot support any special extended syntax that might provide meta-level access to information about the joinpoint that the advice was executed on. We are limited only to providing joinpoint information through advice arguments.

Another consideration is when and how to generate joinpoint information; if our framework always generates as much information as possible, this would be wasteful if the advice does not actually use it.

Figure 2 shows one possible mechanism for specifying what information the advice might need. In this mechanism, the pointcut declaration includes a parameter list that names the information that the advice should be provided with; in the examples above each pointcut only has one parameter, but more could be imagined. Thus it would be the responsibility of the aspect creator to declare what information the
Pointcuts in Teams for Java

```java
aspect BasicBlock {
  pointcut allBB(BlockID): basicblock();
  pointcut appBB(LocationInfo): basicblock() &&
      within(myapp.*.*);
}
```

Advice in POPL (Java)

```java
public class AspectBasicBlock {
  public static void before_allBB(BlockID id)
  {
    System.err.println("Entered basic block " + id);
  }
  public static void before_appBB(LocationInfo info)
  {
    System.err.println("Entered basic block " +
          info.currentMethod +
          ":" + info.currentLine);
  }
}
```

Figure 2: Example Aspect Specification in Hypothetical TEAMS for Java.

advice should be provided with; this is similar to how the `args` designator works.

Each fundamental pointcut designator in the aspect language would have a set of information types that it could provide, and the pointcut expression compiler will verify that there is at least one pointcut designator in the expression that can generate the requested information type. For example, a method execution designator could produce the class and method name of the executing method, and a basic block designator might produce, in addition to the enclosing class and method names, a unique block ID and the source line number on which it begins. If more than one can generate the requested type then the designator that produces the most specific data of the type will be the one that generates the joinpoint data.

This information would then passed as a plain typed parameter in the advice POPL. The advice method must be declared to accept the parameters that the pointcut expression declares.

2.2 Extensibility

We envision a modular architecture of the runtime system where each pointcut designator in the pointcut language will have its own implementation of a common interface, acting as a plugin component to the base weaving engine. In this way new pointcut designators can be added by us or by others that implement unique monitoring-oriented joinpoint types.

The pointcut expression grammar will also need to be extended when introducing a new pointcut designator for a new joinpoint type. We envision two possibilities for this issue. One is to be careful in designing the grammar and its associated actions so as to make it cleanly extensible at the pointcut designator syntax. One issue is how to handle the syntax of the designator arguments, which can be widely varying (e.g., many existing designators takes a regular expression to match package/class/method names). A second approach that may better handle this issue is employing a grammar merging tool, such as `gDiff`, and to allow the writer of a new designator to write a grammar for their designator, following particular naming rules to avoid conflicts, and then merging that grammar with the base aspect grammar. We plan to experiment with this approach to evaluate its viability.

Because we are focusing on monitoring as the domain for our AOP framework, we are not concerned with the advice interference problem. We intend our advice methods to be observational only and not to affect the program state. However, since the advice is in the POPL of the system and the user can be in control of how it gets compiled, there is nothing immediately preventing a user from accessing and modifying parts of their system within their advice. We view such use as outside TEAMS’s intended scope and do not plan on providing any particular support for it. Possible indirect affects from e.g., timing modifications of concurrent behavior, are also beyond the scope of our concerns.

2.3 Support for Broad Runtime Monitoring Capabilities

When we began pursuing the use of AOP for broad runtime monitoring needs, we quickly encountered fundamental discrepancies in the assumptions behind typical AOP frameworks that the radically different ideas that some runtime monitoring approaches needed. We defined various dimensions that an AOP framework needed to support to be considered to fully support runtime monitoring; these included code and data which are typically handled in AOP frameworks, but also included probability and time, where advice might be executed at joinpoints based on some probabilistic sampling rate or where the joinpoint type itself might be a temporal designation rather than anything related to the program structure. These ended up being difficult issues to solve in existing AOP frameworks, and TEAMS must be designed to handle them from the beginning in order to be useful to us.

A typical approach to deciding where to weave advice for a compound pointcut expression is to allow each pointcut designator to produce a shadow that describes where it matches the program, and then to find the intersection (for the `&&` operation) of these shadows as the concrete description of the entire pointcut.

To handle the sampling ideas, two distinct alternative mechanisms are needed. One is for a pointcut designator to base its shadow on other shadows; for example, with a static joinpoint selection probability we would make a probabilistic decision at each shadowed code location whether to
weave advice or not. Two is to be able to insert dynamic residue computation that makes a runtime decision whether to execute advice or not; for example, with a dynamic joinpoint probability, each time a joinpoint occurs a dynamic probabilistic decision is made to execute the advice or not.

To handle the time domain pointcut designators, we need mechanisms where advice can be triggered completely separate from what the program is doing. We expect that with careful interface design and modular framework construction we will be able to allow new advice mechanisms to attach to the runtime framework and access the information and advice handles they need to accomplish these novel and different ideas that are crucial to supporting broad runtime monitoring needs. For example, a time-domain advice plugin will register itself, read some meta-information about the advice intervals it needs to obey for the particular pointcut expressions being used, and then it will spawn a thread to act as an alarm, sleeping the appropriate intervals and then waking up to execute the necessary advice method(s).

Data domain pointcut designators may also need mechanisms that are entirely separate from the program. For example, while in Java object field accesses are easily mapped to specific bytecode instructions, for a less memory safe language such as C++ there is no such easy reduction; in this case a mechanism such as using page protection faults to catch memory references, or even CPU watchpoint registers which can trap accesses to specific addresses, might be needed to provide an efficient data domain designator.

3. PROTOTYPE

Figure 3 shows the high-level design of TEAMS-Java, the initial beginnings of prototyping our framework for the Java language. In this prototype we are currently ignoring the possibility of static weaving, considering it an optimization which we can later revisit.

The pointcut compiler is created from an ANTLR grammar, and its job is to verify the consistency of the pointcut expressions, extract out the necessary information and identifiers, and match the joinpoint information needed to the proper pointcut designators in the pointcut expression. It then saves a strictly formatted reduction of this information in an intermediate representation.

The runtime engine is embodied as a class loader agent which triggers on each Java class being loaded. It uses the ASM bytecode manipulation library to perform the necessary weaving operations on the classes being loaded[7]. ASM has built-in functionality for constructing and accessing a control flow graph of a method, and code-level weaving is nicely doable using ASM.

We have created rudimentary pointcut designators for method execution, method call, and basic blocks. The runtime weaver loads the advice class which is named the same as the aspect name used in the pointcut definition, with a prefix "Aspect". Each advice is a static method named as the same name of the pointcut expression name, with a prefix indicating its weaving mode: "before" or "after". The runtime weaver is a JVM class loader extension that inspects each class being loaded and uses ASM to on-the-fly transform the bytecode to include invocations to the advice methods where necessary.

Our system is a prototype in the true sense of the word, in that we are still experimenting with various capabilities and expect that we will need a rigorous re-design to support the full ideas outlined in Section 2.

4. EVALUATION

Table 1 shows some very preliminary execution performance evaluations results for TEAMS. The programs used are taken from the DaCapo java benchmark suite (9.12-bach release) [4], run within the DaCapo framework but using the Unix time command to get user-space execution times. Where AspectJ has an equivalent pointcut designator, we try to reproduce as closely as possible the same results as in TeAMS (we do not have a full implementation yet so in TeAMS we, e.g., hardcoded a restriction not to weave in the DaCapo classes or in the Java library classes, but in AspectJ we use the within PCD to accomplish essentially the same thing.)

The results show the benchmark execution time without any instrumentation, and then the execution time and number of advice executions for each of TeAMS and AspectJ. In both the advice bodies were a simple increment of a counter variable, plus a check to print out the counter on the final execution of the advice (which we determined beforehand on a previous fully-traced execution).

If we look at the Xalan execution and call results, AspectJ
Table 1: Execution overhead for TEAMS.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>TEAMS</th>
<th>AspectJ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Instrumentation</td>
<td># JP Exec</td>
</tr>
<tr>
<td>Xalan, execution(<em>get</em>)</td>
<td>14.75</td>
<td>1.29M</td>
</tr>
<tr>
<td>Xalan, call(<em>get</em>)</td>
<td>14.75</td>
<td>1.25M</td>
</tr>
<tr>
<td>Xalan, basicblock</td>
<td>14.75</td>
<td>18.9M</td>
</tr>
<tr>
<td>H2, execution(<em>get</em>)</td>
<td>38.62</td>
<td>161K</td>
</tr>
<tr>
<td>H2, call(<em>get</em>)</td>
<td>38.62</td>
<td>768M</td>
</tr>
<tr>
<td>H2, basicblock</td>
<td>38.62</td>
<td>2.46B</td>
</tr>
</tbody>
</table>

Figure 4: Per-Advice Call Performance.

has much better performance, with an overhead of about 0.5% on the execution pointcut and about 2.3% for the call pointcut, while TEAMS has an overhead of 12.4% on execution and 59.5% on call. Recall, though, that TEAMS does runtime advice weaving while AspectJ weaves in a separate compilation step, so the TEAMS times include weaving overhead; we do not yet have an analysis of how much of the overhead is weaving and how much is advice execution, but weaving the more distributed calls would likely be more costly than weaving only in the bodies of the matching methods for the execution pointcut, so the jump makes sense.

Moreover, the overhead for the basic block pointcut is 62.5%, only a slight increase from the call overhead but with 15 times more advice executions. It is probably unlikely that most of this time is weaving time, but that the JVM is doing a better job at inlining the advice execution for the basic block than for the calls. This would also explain the increase in the AspectJ time from execution to call; execution advices are more localized and thus are more likely to be inlined while advices at each call site are more distributed and so many may not get executed enough to be inlined.

Figure 4 shows the per-advice-call overhead comparisons for the data from Table 1, where the corresponding TEAMS and AspectJ datapoints are vertically lined up, and the two basic block data points stand alone to the right. Note that the vertical scale is logarithmic. Two interesting observations arise: one, for the H2 execution and call pointcut designators, TEAMS is relatively close to AspectJ in performance, while for the Xalan execution and call designators the discrepancy is quite large; and two, that although TEAMS is always worse than AspectJ when comparing the same configuration, both have large ranges in overhead and TEAMS is not uniformly worse than all AspectJ configurations. Indeed the H2 basic block advice execution is comparable to the best AspectJ advice execution on a per-advice basis.

Thus, while the initial overheads are sometimes quite a bit larger than AspectJ’s overheads, our very initial effort in TEAMS seems to be within the range of AspectJ’s performance.

5. RELATED WORK

As mentioned earlier, recently Binder et al. [2] have reiterated the need for AOP to be expanded for supporting the many issues in runtime monitoring and dynamic analysis. Rajan and Sullivan were, as far as we can tell, the first to make a clear note that current AOP models are insufficient for supporting many monitoring tasks such as coverage and profiling [15].

So many of the successful applications of AOP are program instrumentation and analysis that they would be too numerous to try to list; our goal is to support more fundamental instrumentation capabilities for these applications to be pushed even further. For example, the Monitor-Oriented Programming [8, 12] framework is an elegant system for creating formal analyses that provide runtime verification of particular properties, but its implementation is limited by what AspectJ supports [10]. If we can offer MOP more extensive access to the program behavior then it becomes more useful and capable of verifying more properties of an
executing program.

Various framework approaches have been undertaken to explore new AOP ideas, including XAspects [16], CME [9], and @J [3] (which is especially targeted for dynamic analyses, though Java-only), but were generally targeted for particular uses and are in various stages of abandonment.

6. CONCLUSION

The TEAMs framework presented here is in its earliest prototype form, and yet it appears to have promise for supporting the directions we hope to go. Our vision is to use the elegant, formal, notations of AOP to support a broad variety of the instrumentation needs that occur in runtime monitoring. Along with some standard AOP pointcut designators, shown here was one sample pointcut designator that provided a more detailed code-based joinpoint than most AOP frameworks do, the basic block PCD. We intend to not only support further code-centric designators, but also support designators that move away from a code-centric view and incorporate such instrumentation needs as stochastic sampling and even time-domain sampling.

In the future we hope that TEAMs will become a useful tool for both researchers and practitioners who need to implement some custom instrumentation for their own purposes but do not desire to build it themselves from scratch.

7. REFERENCES


