AOP for the Domain of Runtime Monitoring:  
Breaking Out of the Code-Based Model

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

Jonathan Cook  
Department of Computer Science  
New Mexico State University  
Las Cruces, NM 88003 USA  
joncook@nmsu.edu

ABSTRACT
Runtime monitoring, even the canonical “logging” example of AOP, has long been one of the domains into which AOP has effectively been deployed. Yet to date AOP has not supported the full breadth of needs across the scope of widely varied runtime monitoring applications. In this paper we present the directions in which we believe AOP needs to be extended in order to better support the breadth of runtime monitoring needs.

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

General Terms
Languages, Measurement

Keywords
Runtime monitoring, Aspect oriented programming

1. INTRODUCTION
Runtime monitoring, observing some behavior of a program during its execution, is very useful in a myriad of ways for developers to produce better software. Runtime monitoring is used in a variety of lifecycle activities, from a coverage tool used during testing to ensure good code coverage, or a logger producing a trace that can offer post-mortem clues to program behavior, to various tools used in reverse engineering that extract information about the program so that a developer can understand a program they did not create. Aspect oriented programming, likewise, is an elegant framework for constructing and implementing program behavior that is orthogonal to the underlying program code base, and this makes it a natural fit for the domain of runtime monitoring, which must rely on inserted instrumentation to implement the monitoring. Indeed, AOP has long been used for a variety of runtime monitoring needs.

In spite of this fit, runtime monitoring has a wide breadth of needs that current AOP languages are not able to satisfy. Code coverage is one example where AOP falls short; code coverage determines which source code has been covered and which has not during execution, and is generally applied at the statement or basic-block level. Current AOP languages lack the ability to instrument or weave at a fine enough level of granularity to compute this coverage information. Efficient profiling is another example where current AOP frameworks fall short; profilers typically use timer-based sampling techniques to obtain point samples of where in the code the program is currently executing. This type of instrumentation is completely decoupled from the underlying code (source, byte, or machine code), but current AOP frameworks take a code-centric view of the weaving problem.

Thus, for the full domain of runtime monitoring to be supported, we believe that AOP needs extended to support both more detail in weaving, and to offer new dimensions of weaving that break away from the code-centric view and support other dimensions, such as time. Such extensions will probably also be useful in other domains, but we restrict our ideas here to the support of runtime monitoring in all of its varied forms. This position paper presents ideas that provide a starting point for addressing these extensions.

2. RUNTIME MONITORING AND AOP
Runtime monitoring is the act of observing an executing system in order to learn something about its dynamic behavior. Runtime monitoring generally refers only to the act of monitoring, which is in turn typically employed by some higher level analysis [4, 14]. Monitors span the gamut from “barely noticeable” to “extremely painful” in terms of their impact on application performance. Sampling-based code profilers are extremely efficient, gathering very simple data periodically while the program executes, while complex analyses such as the invariant inference of Daikon [11] require heavily instrumented programs to collect detailed variable and execution history.

Many applications of AOP have centered around monitoring and/or analyzing the behavior of a system, and we cannot adequately do a survey here; even the “hello world” of AOP, logging, is a monitoring example. There are several reasons that why AOP is a good choice to use in runtime monitoring: it naturally captures the idea of scattered instrumentation in a base program; it can be used on existing programs; it is formal and uses normal programming concepts that programmers can readily grasp; and generally ev-

1Being language-based, AOP frameworks typically have
every AOP language has the same terminology, so it is easy for a programmer to switch between different AOP languages.

Rajan and Sullivan were, as far as we can tell, the first to make a clear note that current AOP models are insufficient to support many monitoring tasks such as coverage and profiling [13], and Rajan continues work related to such ideas [10]. Others have extended AOP to support specific analyses; LoopsAJ [12] introduces a pointcut for counting-type loops in AspectJ, and ArrayPT adds a pointcut for array element access [8].

Coady et al. [9] noted that currently available pointcuts were both not abstract enough and sometimes not detailed enough for an aspect language interacting with the virtual machine, a level similar to that needed by many runtime monitors. Sousan et al. [15] built several interesting pointcuts on top of existing AspectC++ pointcuts, for the monitoring and analysis of embedded real-time systems. Much work has also been done on dynamic aspects (e.g., [1]).

Bodden and Havelund [5] created Racer, a race detection tool, using AOP and the abc system. In their work they also found that the existing set of pointcuts was insufficient for their monitoring needs, and implemented their own new pointcuts to specifically monitor locks and thus detect potential race conditions. Other monitoring-related work includes [2, 6].

Monitor-Oriented Programming was detailed in [7]. This work describes the monitoring task in high-level formal notations, and demonstrates how AOP can be used to provide a rigorous framework for building runtime verification instrumentation. It did not, however, attempt to go beyond instrumentation that was already supported by AspectJ.

3. EXTENDING THE WEAVING DETAIL

One main limitation in using AOP across the monitoring spectrum is that the level of weaving detail is restricted, generally to features like method/function calls/returns, exceptions, and object data member access if available. Many monitoring scenarios need instrumentation at the basic block level or even statement level, and may need to instrument local variable, parameter, or other types of data access.

Thus, our first position is that to more fully support runtime monitoring, AOP needs more detailed pointcuts that allow fine-grained instrumentation to be specified and inserted. We have implemented a basic block pointcut and a loop edge pointcut using abc [3], and have experimented with condition and statement pointcuts. Ultimately, one can imagine that a use could be found for pointcuts that expose all of the basic constructs in a given programming language. Our preliminary work and example, described later, will further describe the issue of weaving detail.

4. NEW WEAVING DIMENSIONS

Extending weaving detail is very important, but perhaps the biggest limitation to supporting the full breadth of runtime monitoring is that the general view of AOP weaving has been single dimensioned: that of the underlying program, and even more directly, that of the program’s source code (even if already translated into byte or machine code). That is, pointcuts specify patterns over the program code (typically involving program symbols) which match joinpoints that are typically reducible to statically identifiable points in the program (shadows)\(^2\). Even with pointcuts that could be independent of the source code, such as object data member access, implementation is usually translated to code-based instrumentation (e.g., finding all bytecode instructions that read and write the data member) rather than, say, a VM-based flag to catch the access of the data member.

We propose a more extensive multi-dimensioned view of AOP. Our second position is that aspect weaving should not be limited to be based only on code features. Below are three other dimensions of weaving that would be useful for runtime monitoring, and possibly other domains as well.

- **Time** : Pointcuts referring to time constraints for when advice is executed; the time dimension includes absolute “wall clock” time and relative program execution time.
- **Data Space** : Pointcuts referring to constraints over program data for deciding when to execute advice;
- **Probability or counting** : Pointcuts providing a probabilistic or count-based control of advice execution, extending the all-or-nothing model to allow a probability of advice execution.

These are each discussed further in the following subsections. Where we provide hypothetical new pointcuts, we are not attempting to provide fully complete lists for each domain, but simply give examples meant to illustrate the ideas.

4.1 Time

The time dimension will weave aspect code not based on location in code but on timers, either relative or absolute. An obvious example of the utility of this dimension is profiling, where a timer-based interruption of the program samples where the program is at that point in time, and constructs a statistical profile of the execution behavior of the program. Other time-based uses would be to periodically check data structure health or application progress. This might be done over relative time, such as every 10 minutes of program execution time, or it might be over absolute time, such as at 1:00am while the system is more likely to be idle. Hypothetical built-in pointcuts for this dimension are:

- `timer_abs(T,I)` execute aspect every T seconds of real time, beginning at time I.
- `timer_reI(T)` execute aspect every T seconds of application time, possibly per thread.
- `timer_class(C,T)` execute aspect every T seconds of time spent in class C.

4.2 Data Space

Weaving on data space could, similarly to time, be performed on absolute or relative scales. Advice might be executed based on total space used by the application, by a class, or by a module; it might be executed based on when some data is accessed (beyond just object data members); at a system level, advice execution could be connected to page access (page traps are a “traditional” efficient instrumentation mechanism for data-watching monitors). In this dimension, garbage collection algorithms that trigger on space usage could be seen as an aspect in the space dimension:

\(^2\)The static points may need slightly refined by dynamic residues.
when the application’s space usage reaches the next triggering level, the garbage collector is triggered to execute. Thus, conceptually, AOP over non-code dimensions can be a unifying theory for some current practices such as garbage collection. Hypothetical built-in pointcuts for this dimension are:

- \texttt{total\_space(M,N)} execute aspect if total space usage rises above \(M\) bytes, and every \(N\) bytes thereafter. (Must be edge-triggered, not level-triggered.)
- \texttt{class\_space(C,M,N)} execute aspect if space usage of class \(C\) rises above \(M\) bytes, and every \(N\) bytes thereafter. (Must be edge-triggered, not level-triggered.)
- \texttt{datastruct\_range(Root)} execute aspect if a particular portion of a data structure is accessed.

### 4.3 Probability or Counting

This dimension controls whether or not the advice is actually executed where woven, or not. Current AOP assumes that every time a joinpoint satisfying the pointcut expression is reached, the advice will execute\(^3\). However, research in runtime monitoring has shown the utility of \textit{sampling} based approaches, where instrumentation is executed probabilistically, either randomly or (more efficiently) with a countdown/reset approach. An AOP system could support such monitoring needs inherently, by allowing weaving to be done in this dimension.

Other types of counting are useful for certain runtime monitoring situations. For coverage analysis without regards to profiling, only a boolean “hit” flag is needed for each basic block. Once the flag is set, no more monitoring for that block is needed. Thus, being able to specify that an aspect executes on only the first instance (or the first \(N\) for other scenarios) would allow AOP to support this type of monitoring. Similarly, some applications need to be allowed to start up and get past an initialization or start up phase before it is useful to monitor them. In this case, skipping the first \(N\) executions of an aspect would be useful.

Hypothetical built-in pointcuts for this dimension are:

- \texttt{probability(P)} execute aspect only with probability \(P\).
- \texttt{every\_nth(N)} execute aspect only every \(N\)th time.
- \texttt{first\_num(N)} execute aspect only the first \(N\) times.
- \texttt{after\_num(N)} execute aspect only after skipping it \(N\) times.

### 5. PRELIMINARY WORK

We have experimented with some initial ideas, using \texttt{abc} [3] as our underlying framework. \textit{Abc} is an open and extendible implementation of AspectJ. Our goal is not just that AOP can support more runtime monitoring needs, but that it can do it efficiently. Thus, part of our goal is to extend AOP in a manner that allows for efficient implementation.

Our first extension extends the level of detail in the code-based dimension of weaving, and implements a new \texttt{basicblock} pointcut designator. A source code example showing the usage of the basic block joinpoint is shown in Figure 1. For each basic block in the instrumented code, this aspect will print out the event of entering the block and of exiting the block, with reflective information indicating the line number in the actual source code of the start of the block and a unique identification number for that block.

Our new extension could be used to support a variety of monitoring needs that require block-level monitoring, including code coverage. Other analyses that, say, use a control flow graph combined with dynamic block execution information from the runtime monitor are enabled with this pointcut. Such monitors are not possible in existing AOP frameworks. The basic block extension shown here is only one example on the extensions that we intend to implement. Although we have not added any new native, efficient pointcuts that support the time or probability dimensions, we have experimented with existing capabilities. For example, the \textit{sampling-based} approach to monitoring, can be accomplished by wrapping a \texttt{basicblock} aspect with another aspect that controls the execution, such as:

```java
void around(): adviceexecution() &
if (rand.nextInt(4)==0) && within(Wrapped) {
  proceed(); // execute basicblock aspect
}
```

In this example the \texttt{Wrapped} aspect will execute only 25\% of the time. This method, although not efficient nor modular, can allow us to experiment with other dimensions of weaving before we undertake their implementation.

### 6. EXAMPLE

As both an example of our \texttt{basicblock} extension and as a demonstration that other weaving dimensions are needed for \textit{efficient} monitoring instrumentation by AOP, we present a code coverage aspect example here. All experiments were run on a Dell rack server, with a quad-core Intel Xeon 64-bit CPU, 2.13GHz, 4GB RAM, running OpenSuse Linux 10.3, kernel version 2.6.22.18. All execution time values are the result of using the \textit{Unix time} command, averaging the execution of five equivalent runs, and subtracting out an average value for executing an empty Java program, in order to factor out the JVM startup cost.

Using the basic block pointcut designator, we developed an aspect program that simply dumps a trace of basic block executions (printing the method name and the block ID number) to a file. A postprocessing step calculates the coverage statistics from the trace file and the file produced during weaving that records how many blocks each instrumented method contains. This is certainly not the most efficient method for doing coverage analysis; still, our results demonstrate that our extensions can support this “canonical” monitoring example, and the output (a block trace) could be used for other analyses beyond just basic coverage.

\[^3\]Some dynamic weavers can essentially turn on or off, but still do not offer the notion, nor try to efficiently support, an active probability of execution.
Table 1: Coverage example results.

<table>
<thead>
<tr>
<th>Java Application</th>
<th>Full app # blocks</th>
<th>Key class # blocks</th>
<th>Time, no instr</th>
<th>Time, full instr</th>
<th>Time, key class instr</th>
<th>Coverage % full instr</th>
<th>Coverage % key class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Linpack</td>
<td>156</td>
<td>156</td>
<td>0.0675</td>
<td>4.603</td>
<td>4.603</td>
<td>69.23</td>
<td>69.23</td>
</tr>
<tr>
<td>JTetris</td>
<td>240</td>
<td>58</td>
<td>0.3275</td>
<td>0.9133</td>
<td>0.9133</td>
<td>65.48</td>
<td>64.79</td>
</tr>
<tr>
<td>Image2Html</td>
<td>409</td>
<td>267</td>
<td>0.6611</td>
<td>16.7019</td>
<td>15.7491</td>
<td>51.83</td>
<td>64.79</td>
</tr>
</tbody>
</table>

Table 1 shows the results of the coverage analysis. The applications we used are JTetris, an implementation of the popular Tetris game; Image2Html, a program that takes a JPEG image and translates it into a colored HTML "ascii art" picture; and Java Linpack, an implementation in Java of the FORTRAN Linpack routines (matrix manipulation).

We ran each program either fully instrumented or instrumented on a "key class", one class that is central to the program and would likely be the focus of intense scrutiny if one was evaluating the program. Note that for Java Linpack, the application consists only of one class, and for Image2Html, one class does most of the processing.

As can be seen, our simplistic block monitoring approach does incur significant overhead. The compute-bound programs run much slower, yet the overhead for JTetris is not obvious to the user, and is thus tolerable in practice. An aspect using a "native" counting pointcut that only executed on the first execution of a block could implement this basic coverage analysis much more efficiently.

7. CONCLUSION

This paper presented ideas for extending the normal AOP concepts to support the full range of runtime monitoring needs. We proposed that new pointcuts are needed that operate at a finer level of detail over the base program, and we proposed that pointcuts are needed in dimensions of weaving other than the program itself: time, probability and counting, and data space. We believe that these and other new ideas for aspect weaving will serve to enable AOP to be used for a large number of program monitoring tasks. This will move runtime monitoring from being dependent on highly technical instrumentation requirements to being generally available to developers who may need particular monitoring tasks for their unique project. Although not addressed in this paper, we also imagine that these new dimensions of weaving, and added detail of code-dimension weaving, will be useful for other domains. The ideas of different dimensions also open up new realms of thinking about dynamic weaving and runtime support, and other parts of “typical” AOP frameworks.

8. REFERENCES