Extending AOP to Support Broad Runtime Monitoring Needs

Amjad Nusayr
New Mexico State University
anusayr@cs.nmsu.edu

Jonathan Cook
New Mexico State University
joncook@nmsu.edu

Abstract—Runtime monitoring, where some part of a program’s behavior and/or data is observed during execution, is a very useful technique that software developers to use for understanding, analyzing, debugging, and improving their software. Aspect oriented programming is a natural fit for supporting the wide ranging instrumentation needs of runtime monitoring, but so far the limitations of AOP frameworks, namely supporting only code-based weaving and only a limited set of code feature joinpoint types, have hindered the application of AOP to many runtime monitoring needs. In this paper we present ideas for extending AOP to support weaving on dimensions other than source code, and for extending code-based weaving to finer-grained language constructs.

I. INTRODUCTION

Aspect oriented programming is an elegant framework for constructing and implementing program behavior that is orthogonal to the underlying program code base. This type of crosscutting concern historically meant that the code implementing the behavior would be scattered around in the program and not localized nor modularized. From the beginning of AOP, it has been observed that runtime monitoring is a natural domain that benefits from AOP, with the “logging” monitor being the “Hello World” example for AOP. Instrumentation for runtime monitoring perfectly fits the idea of a crosscutting concern.

Despite this natural fit and several examples of such use ([5], [7], [10], [15], [21]), runtime monitoring has not seen a massive shift towards AOP systems. One could pessimistically say that this is because AOP is not “standard” in the industrial development toolkit; but even if the shipping product does not use AOP, AOP could still be used in the development environment to perform monitoring tasks needed during development.

Rather, we believe this stall is because of two reasons. One, existing popular AOP frameworks have not offered a great enough level of weaving detail to be more generally useful for runtime monitoring (e.g., statement coverage analysis needs more than just method call and field access weaving, and indeed the citation [5] above needed to extend the aspect language). Two, existing AOP ideas and frameworks have limited the axis of weaving to just the source code, excluding many monitoring needs that might not be easily translatable to code-centric instrumentation (e.g., sampling-based profiling may need weaving based on execution time intervals rather than places in the code). We are not the first to note this; similar limitations were noted earlier in [21].

This paper presents ideas for extending the notions of AOP to include more detailed code-based weaving, and to include weaving over other dimensions of a program rather than just the source code. Others have noted and offered various unique additions and extensions to current AOP models (see Sections II and IV); our contribution here is a presentation of the scope and structure of extensions needed for runtime monitoring in particular.

II. BACKGROUND

A. Runtime Monitoring

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 [3], [12], [22].

Runtime monitoring requires some sort of instrumentation, which probes the executing system and reports back something about the data or execution of that system. Instrumentation is often inserted in-line into a program, so that each time the program hits a certain point of execution, the instrumentation is executed. Sometimes, such as with debugger breakpoints, the instrumentation is hardware-assisted to make it much more efficient. Some monitoring approaches may use timer-based instrumentation, such as sampling-based code profiling (e.g., [17]), while others may use memory page protections to signal an event (e.g., [23]), and still others use probabilistic or counter-based sampling (e.g., [1], [16]).

If a monitoring or higher-level analysis computation is not negligible, the instrumentation may simply record some minimal information about the event and store it for later offline processing, or deliver it through a queue to another thread or process that is performing the more intensive monitoring computation. Effectively creating the necessary instrumentation, and making it efficient, are complex and technologically difficult tasks, often involving creating specialized instrumentation tools by hand.

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, and performing offline analysis to construct the execution profile [17]. Debugger expression watchpoints, on
the other hand, can cause 4-6 orders of magnitude slowdown if they have no hardware support and thus need to evaluate an expression after each instruction [19]. Complex analyses such as the invariant inference of Daikon [9] require heavily instrumented programs to collect data variable history, and also use offline analyses to run the inference algorithms.

B. Aspect Oriented Programming

Aspect Oriented Programming frameworks also instrument an underlying base program, but in AOP this purpose is more generic, to weave in any crosscutting functionality that should be factored out of the base program and not be replicated in the many locations in the program source where it is needed.

A basic AOP model defines some specific fundamental pointcut designators (PCD’s),1 which are features in the program execution where the advice of an aspect can be weaved in. A composition language allows a pointcut expression to combine and constrain these to define a pointcut, which is a set of program joinpoints that satisfy the expression, and where the advice will be woven in.

In existing AOP frameworks, the fundamental pointcut designators are chosen somewhat pragmatically: they must be actually useful to an aspect programmer, but they must also be relatively practical to implement in the AOP system. Thus, in existing AOP systems, pointcut designators are typically points in the program where inserting instrumentation is “not too hard”; for example, method calls are very often used as one of the fundamental pointcut designators.

The most popular AOP system, AspectJ, implements AOP for Java programs. Its pointcut designators include method calls, method executions, object field accesses, exceptions, and a few others [14].

III. EXTENDING AOP IN DIMENSION AND DETAIL

There is a clear synergy between the ideas of AOP and the needs of runtime monitoring, and the only barrier to allowing AOP to be the unifying framework for almost all of runtime monitoring is the fact that current AOP frameworks are far too limited in the level of detail they support weaving at, and the fundamental style of weaving they support. The heart of our contribution here is to detail the extensions to current AOP ideas that will enable AOP to support the vast majority of monitoring needs. This paper is presenting initial ideas for these extensions, and thus the tone of this section is speculative; much more work is needed to experiment with and refine the specific forms and implementations of the various proposed extensions. In [18], a short workshop position paper, we very briefly formulated the idea of extending AOP into heretofore overlooked dimensions in order to support runtime monitoring. Here we elaborate and refine those initial ideas.

The general view of AOP thus far has been a one, or at most two, dimensioned view of aspects: they get woven into a base program based on joinpoints that are code based or data based. That is, the built-in set of pointcut designators that can be part of a pointcut specification are based on programming language features: method call, field access, etc. By far most pointcut designators get mapped to code features, and even the data-access pointcut designators often get mapped to code-based concepts (i.e., all expressions that access an object field).

We propose a more extensive multi-dimensional view of AOP. Aspect weaving should not be limited to only be done based on code features, and this is especially important for AOP to support the broad needs of runtime monitoring. We identify four dimensions of weaving that are needed.

- **Code**: The traditional weaving over code features.
- **Data**: Weaving over concepts in data space.
- **Time**: Weaving based on time constraints.
- **Sampling**: Weaving that supports sampling-based instrumentation.

Each of these is detailed below. In describing the concepts, we are not concerned at all with whether or not they are difficult to implement. We only mean to offer the idea, and justify its potential utility.

1) **Code**: Code dimension weaving is well understood already, and most AOP frameworks support a variety of code-based pointcut designators. We only mention here that, for monitoring purposes, existing AOP frameworks are still quite limited in the features that they support; statement level coverage analysis and many other analyses depend on being able to instrument down to the statement (or basic block) level, yet current AOP systems do not support this. Analyses integrated with various testing methods may need to capture expression and sub-expression evaluation results, and other analyses need similar detail in what they can capture; for AOP to support such monitoring, the level of detail in terms of program concepts that is exposed to advice weaving must be much greater than current practice allows.

2) **Data**: Data dimension weaving has some, but very limited, support in existing AOP frameworks. For example, AspectJ has pointcut designators for object field accesses, but not for accesses to local variables, array elements, or arguments. Above the basic concepts of individual variables, the data dimension could support weaving on higher level data-oriented concepts, such as when a node in a data structure is accessed, when references to objects have changed, or how much space an application has allocated. In the allocation example, garbage collection algorithms that trigger on space usage could be seen as an aspect in the data dimension: 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 and other services, which others have already noted [6].

3) **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

---

1Various literature refers to the basic built-in weaving-point designators of an aspect language as joinpoint types, primitive pointcuts, and pointcut designators. We use the latter term in this paper.
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 for every 10 minutes of program execution time, or it might be over absolute time, such as at 1:00am every Sunday night while the system is more likely to be idle.

Further refinement of this idea could support specific features of the particular programming languages and frameworks it is deployed on. For example, in Java distinct timers per thread might be useful, or per class or per object (an aspect might be “quiescent” until an object has had X amount of computational time on it, then is triggered and is allowed to perform a check on the object’s state or deep state).

4) Sampling: The sampling dimension controls whether or not the aspect is actually executed where woven, or not. Current AOP assumes that every time a joinpoint satisfying the pointcut expression is reached, the aspect will execute. However, research in runtime monitoring has shown the utility of sampling based approaches, where instrumentation is executed probabilistically, either randomly or (more efficiently) with a countdown/reset approach. 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 execute 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.

Many monitoring needs can benefit from “partial” monitoring, from single sites that just need to control the overhead of monitoring, to remote monitoring of user sites where small portions of each user site can be monitored efficiently, and results accumulated to understand the application as a whole. The sample dimension would support this.

5) Integrating the dimensions: Each of the four dimensions above have been shown by the examples given to be useful axes on which to extend AOP for runtime monitoring. We believe these dimensions and extensions will also be useful for other applications of AOP, but we only focus on runtime monitoring here. The question remains, though, on how they integrate and interact with each other. As with existing AOP pointcut designators, not all compositions of new PCD’s in these dimensions would make sense, and the semantics of some compositions need to be carefully specified.

The time dimension pointcut designators are usable by themselves; in this case, the aspect would be executed at a particular time, regardless of where the program is. This means that the aspect could not assume some specific program state or scope, and thus the reflective information available to the aspect would be limited, or at the very least difficult to access. Composing the time pointcut designators with others, such as method invocations, brings in some design decisions because it is virtually guaranteed that the two pointcut designators, each designating an “instantaneous” event, would never be simultaneously satisfied. More work is needed to design useful semantics, but one approach would be to make the time domain designator mean “at least this much time” when composed with other designators.

Data dimension pointcut designators that track space usage are similar to existing ideas in triggering garbage collection, heap compaction, and other runtime efficiency services. Notice that pointcut designators that deal with space usage reaching a particular level must be edge-triggered rather than level-triggered. That is, the aspect executes when the condition first becomes true, and not continuously thereafter.

The sampling dimension is not usable by itself, but would depend on being composed with other pointcut designators that provide an underlying “real” joinpoint set over which to perform the count or the probability. It could naturally be composed with existing pointcut designators, and with both the time and data proposed pointcut designators, although some combinations may not be obviously useful.

IV. RELATED WORK

There is much recent activity and novel ideas for extending AOP in manners similar to our work here, but we do not find previous work that specifically laid out the ideas of various weaving dimensions and types of detailed code weaving that are particularly useful for runtime monitoring. Rajan [20] and then Dyer and Rajan [8] are investigating very similar ideas, explicitly working on arguing for more extensive join point models (thus allowing more pointcut designators) and embodying those in an intermediate language and virtual machine support for weaving. 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 [21].

Harbulot and Gurd [11] used abc to extend AspectJ with a pointcut designator for loops, specifically focusing on numerical loops in scientific code. Bodden and Havelund [5] created a race detection tool, also finding that the existing set of pointcut designators was insufficient for their monitoring needs, and implemented their own new pointcut designators to specifically monitor locks and thus detect potential race conditions. Khaled et al. [13] used AspectJ for program monitoring, specifically for supporting program visualization. Hamlen and Jones [10] used AOP for the security monitoring of references, where security policies are checked in-line with the reference access. Bockisch et al. [4] describe VM support for dynamic join points, and this work may be able to provide the underlying capabilities needed for supporting time and data space dimension pointcut designators, and perhaps the probability/counting dimension as well.

A very nice formal framework for 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 analyses. However, from the perspective of monitoring instrumentation, the MOP work in some sense skipped the hard part, by simply implementing their ideas on top of AspectJ. This means that all of the ideas are limited in practice to be usable only at the level of detail AspectJ provides: method call/return, field access, and a few other program events. Our contribution in this paper is relatively orthogonal to the MOP work; enabling MOP to use our extensions would produce a very powerful monitoring and verification framework.

V. CONCLUSION

This paper presented ideas for extending the normal AOP concepts to support the full range of runtime monitoring needs. We showed that new pointcut designators that operate at a finer level detail over the base program’s code are needed, and we proposed that AOP support dimensions of weaving other than the source code: data, time, and sampling. 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 need particular monitoring tasks. 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 purposes that AOP supports. 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.

Many of our ideas are still preliminary and need not implement, but also experimentation and refinement. We are continuing to work on implementing various pointcut designators, in particular more of the code-based designators. We are also beginning to experiment with using the designators to support common runtime monitoring tasks, and using this experience to refine the designators and to figure out what types of information are needed in the advice used for monitoring. Another direction that needs investigation is improving the performance of monitoring advice, since controlling the overhead cost is generally a real concern in program monitoring. Ongoing work (e.g., [2]) that finds new ways to optimize advice execution may help make detailed runtime monitoring even more efficient and usable in the future. New mechanisms for making reflective information easier and faster to obtain in the advice code will also be needed, perhaps generating more static data during compilation so that advice can quickly access the data without costly runtime searches.

REFERENCES