Towards More Generic Aspect-Oriented Programming:
Rethinking the AOP Joinpoint Concept

Jonathan Cook
New Mexico State University
Las Cruces, NM 88003 USA
E-mail: joncook@nmsu.edu

Amjad Nusayr
University of Houston – Victoria
Victoria, TX 77901 USA

Abstract

In aspect oriented programming, the concepts of joinpoint and pointcut have always been central and have, as their names imply, been centered around the idea of a point in a program’s execution. Furthermore, in practical terms this has often been synonymous with points in the static representation of the program where invocation of advice can be inserted. We present here ideas for rethinking both of these, most significantly presenting the idea of redefining a joinpoint to be an interval of program execution. This redefinition cleans up some concepts in AOP and opens the door to new ideas and mechanisms for future AOP research to consider.

1. Introduction

In aspect-oriented programming (AOP), the joinpoint model and the advice mechanisms are important concepts in order to understand the capabilities for a particular AOP framework. These two essentially capture the weaving capability of the AOP framework, determining where in a computation advice can be applied (joinpoints), how advice will be applied to the underlying program (advice mechanisms), and how abstract the specification of this weaving can be (in the pointcut expression language).

Existing notions of joinpoints and their respective joinpoint models have suffered from a restrictive foundational definition that has hampered the expansion of AOP ideas into the full range of capabilities that it could encompass. This restrictive foundational definition is:

A joinpoint is a well-defined point in the execution of a program where advice can be applied.

This basic definition can be found in essentially equivalent wording in many references e.g., [11], although other researchers have also noted its insufficiency [3, 5, 12].

The problem with this definition is with the focus on the word point (which reifies in the term pointcut). We claim that this word has stymied the development of ideas that can expand where and how AOP ideas can be applied. In fact, joinpoints should not be thought of as points in the program execution but should be a much more abstract concept, which then impacts how they should be specified.

Another issue is that the joinpoint model has not traditionally been well-separated from the advice execution model. The joinpoint model is typically reified into the pointcut expression language, but the way a pointcut expression is specified usually intertwines it with specifying how the advice will be executed, with the traditional mechanisms being before, after, or around the joinpoint.

Just as AOP embraces the notion of separation of concerns, in turn the concept of the joinpoint model should be separated from the concept of the advice execution model. In this paper we present a rethinking of the notions of the joinpoint model and advice execution model, and show how this new thinking can expand and enhance the future application of AOP. Our own interests are focused on using AOP to support the broad needs of runtime monitoring instrumentation, and the ideas presented in this paper will help fulfill those needs.

2. A New Approach to Joinpoints

We believe that the foundational ideas and definitions of joinpoints early on in the development of AOP have hindered the potential application of AOP ideas into new realms. In viewing the body of AOP work we see that for the most part joinpoints have largely corresponded to locations in the source code where a weaving compiler could insert the advice code so that program and advice execution was very efficient. Even joinpoint designators such as object field access, which appear data related and not code related, are translated into advice weaving at code locations where the field is accessed. Dynamic weaving, where run-
time conditions could influence the execution of advice, still often uses underlying mechanisms that involve identifying a set of code locations which might execute the advice and then either performing static weaving augmented with dynamic checks (residues), or applying dynamic weaving on these locations as needed.

While Java supports efficient object field access joinpoints because it is easy to identify the bytecode instructions that access them, in less memory-safe languages such as C++ an object field access joinpoint designator could not be reduced to a small set of locations in the source code, but would rather likely need mapped to many pointer accesses, even with the application of sophisticated points-to analyses. Because of this, AOP frameworks targeting such languages have not attempted to provide these non-code-centric joinpoint designators.

In moving away from a code-centric view in our application of AOP to runtime monitoring, we identified other dimensions of joinpoint designators that cannot be mapped to the code-centric weaving model but nevertheless are useful. One is the data dimension that the above example demonstrates; two other dimensions are time and probability. We defined temporal joinpoint designators that define a joinpoint at an absolute wall-clock time or relative interval-based times. For example, a joinpoint could be at 1:00am each night to run advice that would check consistency in server data structures, or might be every hour of runtime to each night to run advice that would check consistency in based times. For example, a joinpoint could be at 1:00am each night to run advice that would check consistency in server data structures, or might be every hour of runtime to each night to run advice that would dump out usage statistics. These joinpoints are equally “points in the execution of a program” but cannot be related to specific locations in the code. The probability dimension enables advice execution sampling, not selecting points in program executions directly but rather influencing the advice execution model using criteria that are not related to the program itself, another diversion away from thinking of joinpoints as program-centric entities.

Our work thus brought new ways of thinking about how “a well-defined point in the program execution” might be defined, but as we thought of these and how to execute advice at these points, we started to formulate another notion that expanded the idea of a joinpoint.

This second and more important idea that changes the definition of joinpoint is to abandon the thinking of joinpoints as points in the execution. Rather, they should be thought of as abstract intervals of program execution whose endpoints are defined as abstract execution conditions. An interval might be reducible to a point but does not necessarily need to be. Indeed we think this extension actually captures the existing notions better, and the main reason for this extension is to separate the understanding of the advice execution model from the joinpoint model, which currently is intertwined together in the pointcut language.

Pointcut expressions are supposed to be abstract specifications of sets of joinpoints, but they actually also define the advice execution mechanism as well—typically using the well-known before, after, and around mechanisms. If joinpoints are points in a program execution, the keywords representing these mechanisms actually make no sense. A point in Euclidean space is a dimensionless, infinitely small location. Mapping this idea to program execution, a point is a location between two execution steps that change the program state. In this sense, the ideas of before and after collapse into the same thing, both being between the two execution steps surrounding the point, and around is meaningless since there is nothing there to envelope.

Only in thinking of a joinpoint as an interval of execution steps do these advice execution mechanisms make sense, and take on their traditional meanings. Before is before the interval begins, after is after it ends, and around is around the entire execution interval. Thus applying advice execution around a method call defines the joinpoint as the interval from the invocation of the method to the return of the method to its caller.

It is true that one can view the combination of the advice execution mechanism and the pointcut expression as signifying a point in the execution (e.g., before and method-call signify the point before the call), but this we think is confounding two separate “concerns” that should indeed be separate, and it means that the pointcut expression does not by itself determine points in the program execution.

Another violation of the view of a joinpoint as a point is in the treatment of threads by most AOP frameworks. In typical AOP frameworks like AspectJ, advice execution in AspectJ will occur in one thread, and other program threads will continue to execute; this means that the advice does not execute at a point in the program but rather during an interval of program execution, where other threads are advancing the program execution and state while the advice is executing. In reality, current joinpoint models define a joinpoint as a point in a thread’s execution, not a point in a program’s execution, despite the general usage of the latter.

We think our notion of a joinpoint being a program interval solves these issues. It separates the idea of a joinpoint from the advice execution mechanisms, and it can cleanly handle issues of parallelism. We think it ought to be reified into the joinpoint model, and consequently in the pointcut expression language, and will be pursuing that end in our research. Rather than always implicitly defining intervals as current pointcut languages do, the ability to abstractly specify the execution conditions for the start of the interval and separately the execution conditions for the end of the interval, will allow new and novel applications of AOP for software engineering.

Masuhara et al. [14] raised a similar issue but concluded that joinpoints should be points, and devised an event-based understanding of joinpoints and weaving. Although seemingly opposite of our emphasis on intervals, in reality inter-
vals must be denoted by some notion of event at their endpoints, and so our approaches have some similarity. Recognizing the interval explicitly, however, will better support some of the issues in concurrency such as relaxing the point in time when advice might actually execute.

3. Extending the Advice Execution Model

With a new understanding of what a joinpoint is (an interval of program execution), we will pursue an expansion of the notions of what it means to weave and execute advice. Existing notions are obviously successful and useful, and map to this new understanding easily: before means to execute advice before the interval, after means to execute the advice after the interval, and around means to execute some part of the advice before the interval, some part after, and controlling the execution of the interval.

However, we can also imagine and define new ideas for advice execution, in particular dealing concretely with notions of concurrency. At least two notions of concurrency can be dealt with, both of which have been so far only been treated sporadically, without devising an encompassing abstraction [2, 9].

The first is with program concurrency within a multi-threaded or distributed program. Advice execution has traditionally been applied to a single thread—whichever thread reached the (thread-specific) joinpoint. However, advice needs to inspect and reason over one particular program state, all threads should be stopped during advice execution. We encountered this issue when creating time domain joinpoint designators (e.g., invoke advice at a particular wall-clock time). When the joinpoint occurs it is not particular to any single thread, so in our initial experimentation we chose to suspend all application threads. But the general issue applies even to existing joinpoint designators that are “triggered” by one thread but may need to inspect shared data before any other threads change the data.

This idea is extensible to particular thread groups, the generality being that advice execution should be defined not in relation to simply one thread but to a set of threads. This leads the joinpoint model to need joinpoint definitions over sets of threads rather than a single thread. For example, defining a joinpoint condition as when all threads reach a synchronization point is a condition over a set of threads rather than just over one. In our research we will be devising new specification methods to deal with these issues.

The second notion in concurrency is that of how advice is executed. The traditional “weaving” idea is that advice is executed in the program thread that reached the joinpoint, but there is no inherent reason this needs to be the case. Advice could be executed in its own thread while the application thread(s) continue to execute. This will allow computationally intensive advice to execute without stopping the application, provided that the advice does not need a single application state snapshot. Even here, with separate conditions specifying the endpoint for the joinpoint interval, novel combinations could be made—for example, advice could execute in its own thread and application threads could continue until any application thread attempts to make a change in some important data structure the advice is inspecting; here the endpoint of the joinpoint interval would be a modification to the data structure.

Concurrent advice execution will enable weaving ideas such as during, meaning advice is executed somewhere within the joinpoint interval; if any threads reached a condition that cause them to leave the interval, they would suspend until the advice is finished. In the minimal case where the interval is defined to be a point, this captures the idea of “stop all threads” while the advice executes. Other ideas could capture soft- or hard-realtime conditions, such as executing the advice “as soon as possible” after the beginning condition of the joinpoint, or “within 10 milliseconds” of the beginning condition.

4. Related Work

There is much recent activity and novel ideas for extending AOP in a variety of manners, and several approaches to understanding the fundamental ideas in AOP that relate to the ideas we present here.

Masuhara et al. [14] raised the similar issue of whether joinpoints should be intervals or points, and they and others have continued work relating to their view of joinpoints as events [3, 13]. Others have modeled and investigated a variety of views on how AOP should work [1, 4]. Kojarski and Lorenz [12] define an elegant model to understand the pieces of AOP and how different understandings of AOP can be formally modeled; we can build on such approaches for our work. Very recently Binder et al. [5] have reiterated the need for AOP to be expanded for supporting the many issues in runtime monitoring and dynamic analysis. Dyer and Rajan [10] investigated new AOP infrastructure 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.

A very nice formal framework for Monitor-Oriented Programming was detailed in [8]. 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. The language used to describe the monitoring task may provide a good foundation for thinking about more general AOP pointcut expressions.

Much of the notion of joinpoints being intervals comes from thinking about concurrency. Even though the notion of thread and advice interaction has been touched on in
some previous work [2, 7], this work did not propose a concrete model for advice execution in threads. Most popular AOP languages effectively ignore concurrency; in AspectJ the thread that reaches a joinpoint will execute the advice, and nothing is specified beyond that. Ansaloni and Binder [2] introduced a framework that enables asynchronous advice execution by collecting advice invocations in a buffer and executing the advices in a separate thread. Douence et al. [9] created Concurrent Event-based AOP (CEAOP) which defines concurrent AOP using labeled transition systems, and overcoming some of the limitations of synchronization and thread communications. CEAOP supports concurrency in the underlying program, and concurrent execution of advice with the base program by allowing the advice body executions in parallel.

5. Conclusion

Fundamental to AOP is the notion of a joinpoint, which has traditionally been thought of as a point in the program execution that can practically be mapped to points in the program itself. We argued here that the notion of a joinpoint really should be a defined interval of program execution whose defining conditions may or may not be related to loci in the program itself. This change better captures existing usage of AOP, in particular with the around advice application and the treatment of threads, and it allows thinking about new and novel directions for AOP. We also argue here that this change in joinpoint thinking also helps to better separate advice execution from pointcut definition, and allows the imagination of new types of advice execution that have wide applicability in the software engineering field.

In this paper we are only presenting ideas of how the abstractions in AOP might be generalized and expanded. We are not attempting to claim that it is straightforward to put these ideas into practice. Indeed the existing ideas within AOP are arguably there because it was somewhat obvious how to implement them efficiently. Other ideas have followed similar paths. The idea of tracematches as joinpoint designators was introduced first, then followed by work in optimizing their detection [6].

Working out other details as well, such as how to create a usable advice language that embodies these ideas without overwhelming the user, may be problematic. Certainly some of the success of AOP, as with other technologies, is found in the combination of power and simplicity. Yet the continuing research in AOP should consider breaking away from ideas that limit what can be imagined and implemented, and should explore radical new and novel approaches to AOP.

References