# Realizing Aspects by Transforming for Events

Robert E. Filman[1] and Klaus Havelund[2]

[1] Research Institute for Advanced Computer Science
NASA Ames Research Center, MS/269-2
Moffett Field, CA 94035
rfilman@mail.arc.nasa.gov
[2] Kestrel Technology
NASA Ames Research Center, MS/269-2
Moffett Field, CA 94035
havelund@email.arc.nasa.gov

**Abstract.** We explore the extent to which concerns can be separated in programs by program transformation with respect to the events required by these concerns. We describe our early work on developing a system to perform event-driven transformation and discuss possible applications of this approach.

## 1 Aspect-Oriented Programming

Programming is about realizing a set of requirements in an operational software system. One has a (perhaps changing) set of properties desired of a system, and builds and evolves that system to achieve those properties. Software engineering is the accumulated set of processes, methodologies, and tools to ease that evolutionary process, including techniques for figuring out what it is that we want to build and mechanisms for producing higher-quality systems.

A recurrent theme of software engineering is that of "separation and localization of concerns." That is, we have "concerns" in building a software system. These concerns range from high-level ilities like reliability and security to low-level issues like caching and synchronization. Our environment should provide us with the linguistic structures to group together just the elements for a particular concern, while nevertheless yielding an efficient operational system. This allows us to concentrate expertise on that concern in one particular place, easing system development and maintenance.

Conventional programming languages (e.g., procedural, imperative and functional languages) provide only a few facilities for separating concerns. The guiding theme of these languages is functional decomposition. One determines what the elements of the domain are and what behaviors they need to have, and writes code to implement these methods. The writer of this code must take account any other requirements beyond pure functionality in the actual code that isn't somehow otherwise supported in the environment. The insight of Object-Oriented Programming was recognizing the leverage of localizing concerns centered on functionality of the elements of the domain (in objects), indexing behavior with

respect to these objects (methods on objects) and providing mechanisms for acquiring default behavior and values (inheritance).

Object-oriented decomposition is often good for the "dominant" functional concern of the system, but leaves other concerns unsupported. This is especially true for those concerns that require coherent behavior across many different functionalities. The best conventional programming can offer is to concentrate the code of other concerns in another function or object, and demand programmers explicitly invoke that code when appropriate. But spreading out the responsibility for invoking the code for multiple concerns to all programmers produces a more brittle system. Each programmer who has to do something right is one more place that a mistake can be made; each spot where something needs to be done is a potential maintenance mishap. Additionally, there may be execution costs in control transfer. Some separate concerns may require so much local context that they may not even be expressible as separate subprograms.

Object-Orientation hasn't given us is any leverage on the problem of cross-cutting requirements and behaviors—elements that require matching code in many places in a system, but which are not neatly packagable in the standard object decompositions. Aspect-Oriented Programming (AOP) (and, more generally, Aspect-Oriented Software Development (AOSD)) is an emerging technology for creating programming systems which provide a "single locus" for expressing such cross-cutting behavior while nevertheless creating systems that actually execute efficiently. The general theme of AOSD is to let programmers express the behavior for each concerns in its own element. Such a system must also include some directions for how the different concerns are to be knitted together into a working system (for example, which each separate concern applies) and a mechanism for actually producing a working system from these elements. For example, most AOP systems given one a way of saying, "High security is achieved by doing $X$. Reliability is achieved by doing $Y$. I want high security in the following places in the code, and reliability on these operations." The AOP system then produces an object that invokes the high security and reliability codes appropriately.

## 2 AOP and Events

Elsewhere, we have argued that the programmatic essence of Aspect-Oriented Programming is making quantified programmatic assertions over programs that otherwise are not annotated to receive these assertions [1–3]. That is, in an AOP system, one wants to be able to say things of the form, "In this program, when the following is true, do the following," without having to go around marking the places that need the desired modification. Most naturally, most of the kinds of quantified statements that programmers want to make are about behavior that is to take place when certain conditions are realized in the executing program. Consider some AOP applications:

**Synchronization in distributed systems [4, 5]** Code to check the synchronization condition ought to run before and after synchronization operations.

This code needs internal state (for example, lock state and a perhaps a queue of waiting processes.)

**Buffer manipulation in operating systems kernels [6]** In operating systems code, prefetching of pages is to be executed when a page fault event occurs within the run-time execution context of a prefetch advisory. Entering and leaving such contexts are also events.

**Distributed middleware [7–9]** Aspects can run on inter-object communication in distributed systems to check for consistent configurations, automate testing, and provide a great variety of other dynamically configurable behavior.

**Distributed quality-of-service [10, 11]** By intercepting service invocation events, aspects can be used to regulate quality of service in concurrent systems.

**Collaboration and design [12, 13]** By interceding at service invocation and repository entry, aspects can be used to enforce access control, synchronization, persistence and resource management in collaborative systems.

**E-commerce [14, 15]** Douence et. al provide an example of using complex histories of client events to determine e-commerce prices. Truyen et. al argue that appropriate aspect behavior is controlled by a complex context set up by a sequence of user actions.

**Replication [16–18]** Replication through aspects is accomplished by taking each action that changes state and propagating it to the replicants.

**Debugging [3]** AOP techniques can be used to create the trace of events to track program execution or to insert the switching commands to force (concurrent) programs to explore multiple program paths.

**Program instrumentation [19]** AOP techniques can be used to insert logging information on system performance at interesting junctures in program execution.

Most conventional AOP systems rely primarily on wrapping function calls with aspect behavior. However, these examples illustrate the need to be able to respond to sequences of events, actions at the individual statement level, and properties of the state of the modeled system.

So what is an event? Ultimately, we want to be able to quantify over anything that changes the data or program counter state of the abstract machine executing a given program. Unfortunately, the abstract interpreter is not completely accessible at the programming level—it is neither fixed by the language definition nor do all its activities (for example, thread switching and garbage collection) have any visible realization in the program text. Similarly, an optimizing compiler may rearrange or elide an "obvious" sequence of expected events. And finally, the data state of the abstract interpreter (including, as it does, all of memory) can be a grand and awkward thing to manipulate.

Nevertheless, much of a program is accessible—we do, after all, have the program text (or the byte code), and can manipulate that code to our heart's content. We may not be able to capture everything that goes on in a particular interpretive environment, we can get close enough for many practical purposes. The strategy we adopt is to argue that most dynamic events, while not necessarily local to a particular spot in the source code, are nevertheless tied to places in

the source code. Table 1 illustrates some primitive events and their associated code loci.

Users are likely to want to express more than just primitive events. The language of events will also want to describe relationships among events, such as that one event occurred before another, that a set of events match some particular predicate, that an event occurred within a particular timeframe, or that no event matching a particular predicate occurred. This suggests that the event language will need (1) abstract temporal relationships, such as "before" and "after," (2) abstract temporal quantifiers, such as "always" and "never", (3) concrete temporal relationships referring to clock time, (4) cardinality relationships on the number times some event has occurred, and (5) aggregation relationships for describing sets of events.

| Event | Syntactic locus |
|---|---|
| Accessing the value of a variable or field | References to that variable |
| Modifying the value of a variable or field | Assignments to that variable |
| Invoking a subprogram | Subprogram calls |
| Cycling through a loop | Loop statements |
| Branching on a conditional | The conditional statement |
| Initializing an instance | The constructors for that object |
| Throwing an exception | Throw statements |
| Catching an exception | Catch statements |
| Waiting on a lock | Wait and synchronize statements |
| Resuming after a lock wait | Other's notify and end of synchronizations |
| Testing a predicate on several fields | Every modification of any of those fields |
| Changing a value on the path to another | Control and data flow analysis over statements (slices) |
| Swapping the running thread | Not reliably accessible, but atomization may be possible |
| Being below on the stack | Subprogram calls |
| Freeing storage | Not reliably accessible, but can try using built-in primitives |
| Throwing an error | Not reliably accessible; could happen anywhere |

**Table 1.** Table 1: Events and event loci

We are currently working on a system where a set of event-action pairs, along with a program, would be presented to a compiler. Each event action pair would include a sentence describing the interesting event in the event language and an action to be executed when that event is realized. Said actions would be programs, and would be parameterized with respect to the elements of the matching events. Examples of such assertions are:

- On every call to method foo in a class that implements the interface B, replace the second parameter of the call to foo with the result of applying method f to that parameter.
- Whenever the value of x+y in any object of class A ever exceeds 5, print a message to the log and reset x to 0.

- If a call to method foo occurs within (some level down on the stack) method baz but without an intervening call to method mumble, omit the call to method gorp in the body of foo.
- Every call to foo must be followed by a call to baz without an intervening call to mumble.

These examples are in natural language. Of course, any actual system will employ something formal.

Clearly, a sufficiently "meta" interpretation mechanism would give us access to many interesting events in the interpreter, enabling a more direct implementation of these ideas. It has often been observed that meta-interpretative and reflective systems can be used to build AOP systems [20]. However, meta-interpreters have traditionally exhibited poor performance. We are looking for implementation strategies where the cost of event recognition is only paid when event recognition is used. This suggests a compiler that would transform programs on the basis of event-action assertions. Such a compiler would work with an extended abstract syntax tree representation of a program. It would map each predicate of the event language into the program locations that could affect the semantics of that event. Such a mapping requires not only abstract syntax tree generation (parsing) and symbol resolution, but also developing primitives with respect to the control and data flow of the program, determining the visibility and lifetimes of symbols, and analyzing the atomicity of actions with respect to multiple threads.

Java compiles into an intermediate form (Java byte codes). In dealing with Java, there is also the choice as to whether to process with respect to the source code or the byte code. Each has its advantages and disadvantages. Byte codes are more real: many of the issues of interest (actual access to variables, even the power consumption of instructions) are revealed precisely at the byte code level. Working with byte codes allows one to modify classes for which one hasn't the source code, including the Java language packages themselves. (JOIE [21] and Jmangler [22] are examples of an AOP systems that perform transformations at the byte code level.) On the other hand, source code is more naturally understandable, allows writing transformations at the human level, and eliminates the need for understanding the JVM and the actions of the compiler. (De Volder's Prolog-based meta-programming system is an example of source-level transformation for AOP [23, 24].) We find the complexity arguments appealing. Thus, our implementation plan is to work at the source code level.

## 3 Related Work

De Volder and his co-workers [23, 24] have argued for doing AOP by program transformation, using a Prolog-based system working on the text of Java programs. We want to extend those ideas to program semantics, combining both the textual locus of dynamic events and transformations requiring complex analysis of the source code.

At the 1998 ECOOP AOP workshop, Fradet and Südholt [25] argued that certain classes of aspects could be expressed as static program transformations. They expanded this argument at the 1999 ECOOP AOP workshop to one of checking for robustness—non-localized, dynamic properties of a system's state [26]. Colcombet and Fradet realized an implementation of these ideas in [27], applying both syntactic and semantic transformations to enforce desired properties on programs. In that system, the user can specify a desired property of a program as a regular expression on syntactically identified points in the program, and the program is transformed into one that raises an exception when the property is violated. Other transformational systems include, Ku a notational attempt at formalizing transformation [28], and Schonger et al's proposal to express abstract syntax trees in XML and use XML transformation tools for tree manipulation [29].

Nelson et al. identify three concern-level foundational composition operators: correspondence, behavioral semantics and binding [30]. Correspondence involves identifying names in different entities that are "the same"—for data items, things that should share storage; for functions, functional fragments that need to be assembled into a whole. Behavioral semantics describe how the functional fragments are assembled. Binding is the usual issue of the statics and dynamics of system construction and change. They discuss alternative formal techniques for establishing properties of composed systems within this basis.

Masuhara et. al present a semantics-based approach to compiling AOP systems. They introduce the notion of "join point shadows"—the places in the text where the a particular aspect needs to be woven [31].

Walker and Murphy argue for events as appropriate "join points" for AOP, and that the events exposed by AspectJ are inadequate [32].

## 4 Concluding remarks

We have suggested that an interesting way to implement AOP systems is by describing the events that are to trigger aspect behavior, and transforming an existing program with respect to these events. We note that we've been considering implementation environments, not software engineering. An underlying implementation does not imply anything about the "right" organization of "separate concerns" to present to a user. In particular, we have been completely agnostic about the appropriate structure for the actions of action-event pairs. It may be the case that unqualified use of an event language with raw action code snippets is a software engineering wonder, but we doubt it. On the other hand, we believe that such transformational system would be an excellent environment for experimenting with and building systems for AOP. In some sense, these ideas can be viewed as a domain-specific language for developing aspect-oriented languages.

## References

1. Filman, R.: What is aspect-oriented programming, revisited. [33]

2. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns (OOPSLA 2000). (2000)
3. Filman, R.E., Havelund, K.: Source-code instrumentation and quantification of events. [34] 45–49
4. Holmes, D., Noble, J., Potter, J.: Aspects of synchronization. In: Workshop on Aspect Oriented Programming (ECOOP 1997). (1997)
5. Netinant, P., Elrad, T., Fayad, M.E.: A layered approach to building open aspect-oriented systems: A framework for the design of on-demand system demodularization. Comm. ACM **44** (2001) 83–85
6. Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., Ong, J.S.: Structuring operating system aspects: Using AOP to improve OS structure modularity. Comm. ACM **44** (2001) 79–82
7. Filman, R.E., Barrett, S., Lee, D.D., Linden, T.: Inserting ilities by controlling communications. Comm. ACM **45** (2002) 116–122
8. Hunleth, F., Cytron, R., Gill, C.: Building customizable middleware using aspect oriented programming. [35]
9. Jørgensen, B.N., Truyen, E., Matthijs, F., Joosen, W.: Customization of object request brokers by application specific policies. In: Proc. Middleware'2000. (2000)
10. Becker, C.: Quality of service and O.O. oriented middleware multiple concerns and their separation. [36] 117–126
11. Zinky, J., Shapiro, R., Loyall, J., Pal, P., Atighetchi, M.: Separation of concerns for reuse of systemic adaptation in quo 3.0. [33]
12. Filman, R.E.: A software architecture for intelligent synthesis environments. In: Proc. 2001 IEEE Aerospace Conference. (2001) 2879–2888
13. Pinto, M., Amor, M., Fuentes, L., Troya, J.: Collaborative virtual environment development: An aspect-oriented approach. [36] 97–102
14. Douence, R., Motelet, O., Südholt, M.: Sophisticated crosscuts for e-commerce. [33]
15. Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Jørgensen, B.N.: Customization of on-line services with simultaneous client-specific views. [33]
16. Antunes, M., Miranda, H., Silva, A.R., Rodrigues, L., Martins, J.: Separating replication from distributed communication: Problems and solutions. [36] 103–110
17. Filman, R.E., Lee, D.D.: Redirecting by injector. [36] 141–146
18. Herrero, J.L., Sánchez, F., Toro, M.: Fault tolerance AOP approach. In: Workshop on Aspect-Oriented Programming and Separation of Concerns (Lancaster). (2001)
19. Deters, M., Cytron, R.K.: Introduction of program instrumentation using aspects. [35]
20. Sullivan, G.T.: Aspect-oriented programming using reflection and meta-object protocols. Comm. ACM **44** (2001) 95–97
21. Cohen, G.A.: Recombing concerns: Experience with transformation. In: Workshop on Multi-Dimensional Separation of Concerns (OOPSLA 1999). (1999)
22. Kniesel, G., Costanza, P., Austermann, M.: JMangler—a framework for load-time transformation of Java class files. In: First IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM 2001). (2001)
23. De Volder, K., Brichau, J., Mens, K., D'Hondt, T.: Logic meta-programming, a framework for domain-specific aspect programming languages. (http://www.cs.ubc.ca/~kdvolder/binaries/cacm-aop-paper.pdf)
24. Volder, K.D., D'Hondt, T.: Aspect-oriented logic meta programming. In Cointe, P., ed.: Meta-Level Architectures and Reflection, 2nd Int'l Conf. Reflection. Volume 1616 of LNCS., Springer Verlag (1999) 250–272

25. Fradet, P., Südholt, M.: AOP: Towards a generic framework using program transformation and analysis. In: Workshop on Aspect Oriented Programming (ECOOP 1998). (1998)
26. Fradet, P., Südholt, M.: An aspect language for robust programming. In: Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999). (1999)
27. Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: Proc. 27th ACM Symp. on Principles of Programming Languages. (2000) 54–66
28. Skipper, M.: A model of composition oriented programming. In: Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000). (2000)
29. Schonger, S., Pulvermueller, E., Sarstedt, S.: Aspect oriented programming and component weaving: Using XML representations of abstract syntax trees. In: Workshop Aspektorientierte Softwareentwicklung (Bonn), Institut für Informatik III, Universität Bonn (2002)
30. Nelson, T., Cowan, D., Alencar, P.: Supporting formal verification of crosscutting concerns. In Yonezawa, A., Matsuoka, S., eds.: Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192, Springer-Verlag (2001) 153–169
31. Masuhara, H., Kiczales, G., Dutchyn, C.: Compilation semantics of aspect-oriented programs. [34] 17–26
32. Walker, R.J., Murphy, G.C.: Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. In: Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001). (2001)
33. Workshop on Advanced Separation of Concerns (ECOOP 2001). In: Workshop on Advanced Separation of Concerns (ECOOP 2001). (2001)
34. FOAL 2002: Foundations of Aspect-Oriented Langauges (AOSD-2002). In: FOAL 2002: Foundations of Aspect-Oriented Langauges (AOSD-2002). (2002)
35. Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001). In: Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001). (2001)
36. Proc. Int'l Workshop on Distributed Dynamic Multiservice Architectures (ICDCS-2001), Vol. 2. In: Proc. Int'l Workshop on Distributed Dynamic Multiservice Architectures (ICDCS-2001), Vol. 2. (2001)