Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs

Raffi Khatchadourian*
Computer Sc. and Eng.
Ohio State University
Columbus, OH, USA
khatchad@cse.ohio-state.edu

Johan Dovland
Department of Informatics
University of Oslo
Oslo, Norway
johand@ifi.uio.no

Neelam Soundarajan
Computer Sc. and Eng.
Ohio State University
Columbus, OH, USA
neelam@cse.ohio-state.edu

ABSTRACT

Reasoning, specification, and verification of Aspect-Oriented (AO) programs presents unique challenges especially as such programs evolve over time. Components, base-code and aspects alike, may be easily added, removed, interchanged, or presently unavailable at unpredictable frequencies. Consequently, modular reasoning of such programs is highly attractive as it enables tractable evolution, otherwise necessitating that the entire program be reexamined each time a component is changed. It is well known, however, that modular reasoning about AO programs is difficult. In this paper, we present our ongoing work in constructing a rely-guarantee style reasoning system for the Aspect-Oriented Programming (AOP) paradigm, adopting a trace-based approach to deal with the *plug-n-play* nature inherent to these programs, thus easing AOP evolution.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*; D.2.4 [**Software Engineering**]: Software Verification—*Formal methods*

General Terms

Languages, theory

Keywords

Aspect-oriented programming, modular reasoning, rely-guarantee

*A portion of this work was administered during this author's visit to the Computing Department, Lancaster University, United Kingdom.

Seventh International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008), April 1, 2008, Brussels, Belgium.

1. INTRODUCTION

Aspect-oriented programming (AOP) [21] allows for modular implementations of crosscutting concerns. Since its inception, many authors [8, 24, 26, 33] have shown how aspects may be used to write localized implementations of important crosscutting concerns such as process synchronization, event logging, data persistence, exceptional situation handling, etc. The *separation of concerns* that AOP enables helps produce programs whose components are increasingly decoupled from one another as a direct consequence of the reduction of *scattered* and *tangled* code. As such, AO programs tend to enjoy *plug-n-play*-type capabilities where base and/or aspect components may be introduced, removed, and interchanged easily [24]. This inherent nature of AOP is beneficial in the sense that AO programs may evolve in a non-invasive fashion simply by "switching" features on and off. However, program evolution is bound to occur at unpredictable frequencies; therefore, programmers are often required to make key decisions and come to conclusions about a software components, base-code and aspect alike, utilizing either incomplete or highly volatile information at hand. Ergo, the ability to reason about individual AO program components modularly and to then compose these reasoning efforts, just as we would compose the components themselves, to obtain the actual behavior of the overall program becomes extremely desirable. This ability would permit AO programmers to avoid the unfortunate situation where the entire program must be reexamined upon each component change, thereby facilitating tractable evolution.

Despite its benefits, modular reasoning about AO programs indeed presents significant challenges [1, 3, 13, 5, 22, 23, 37, 29, 9, 38, 31, 12, 7, 34, 36, 30]. The problem is that, by circumscribing core concerns into classes and crosscutting concerns into aspects we are essentially creating *two* different systems, a *baseline* system (base-code) and an *augmented* system which is the result of applying aspects that alter the behavior of the baseline. Indeed, the ability of an aspect to change the behavior of the base-code that it advises, which is the very reason for much of the power of AOP, is also what causes difficulties for reasoning about the behavior of such software. In fact, as aspects "weave" in and out of (or "plugged" then "played") a software system, we may be forced to reason about the entire system, accounting for the interleaved execution of various pieces of advice with the base-code.

What we would aspire instead is to draw meaningful and useful conclusions about component code, e.g., base-code

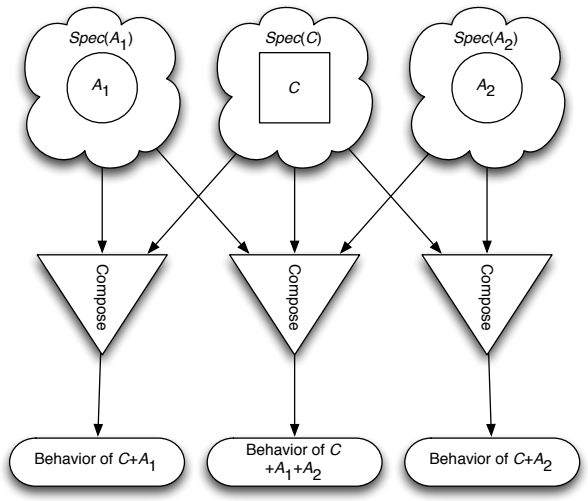

Figure 1: Schematic of behavior derivation using parameterized specifications of a component C under the influence of advice of A_1 and/or A_2.

which may reside in a method or an advice body that is itself subject to advice, *without* considering the actual advice code. Ideally, we would like to specify the behavior of AO components without any particular advice in mind such that in order to arrive at the behavior of the augmented system, just as aspects are *plugged-in* to enhance or *enrich* the behavior of the advised components, the specifications of applicable advice would be "plugged" into the matching behavioral specifications of the base-code. Furthermore, in order to arrive at useful conclusions that remain valid despite the addition of advice, it may be necessary to constrain possible advice behavior in order to preserve the intended semantics of the advised component. In other words, for a component to function correctly, assumptions may need to be made about potentially applicable advice such that these assumptions hold during evolution, with aspects entering, leaving, and *re*-entering the software.

Adopting such an idealized approach would allow developers as AO programs evolve to deduce the behavior of the augmented software without reexamining the *internals* of each component. In essence, the specified behavior of a component would be *parameterized* over the behaviors of all possibly applicable advice. Figure 1 helps to illustrate this notion, portraying schematically at a bird's-eye-view how these parameterized specifications can be hypothetically combined with the specifications of advice in order to obtain the overall system behavior, where $Spec(X)$ refers to the behavioral specifications of component X.

Although the above outlined approach may seem desirable, there are several key obstacles that must be overcome in its achievement:

Usefulness. As previously mentioned, we would like to draw useful conclusions about component code that is subject to the application of advice without considering the actual advice code. As we are focused on *evolving* AO software, the advice code may not yet exist; it may be added a later time. It is not clear, however, exactly how useful these conclusions can be considering that they should hold upon the application of any advice. What sorts of conclusions could we draw in this case?

Complexity. Since component specifications would be written in terms of any applicable advice, there is a strong possibility of these specifications becoming unwieldy. As such, any changes made to the *internals* (i.e., the implementation) of the advised component code would require a rather involved effort to rebuild the component's parameterized specifications. Also, situations may arise where a component `C` may not be under the influence of advice, yet *Spec*(`C`) would still be specified over all possibly applicable advice. Therefore, the complexity of the specification may be unnecessarily complicated. This situation is pictorially represented in 2 and further discussed later in this section. Making suitable restrictions on the behavior of potential advice via the use of language constructs, minimizing the join point model, providing behavioral constraining assertions by adapting a rely/guarantee [39, 18] methodology, which is the focus of our previous work [19, 35], and using predicates and/or functions on specifications themselves as in [16], which is a focus of our future work, may help alleviate several of these obstacles.

Obliviousness. Annotating the component code with parameterized specifications by very nature compromises the traditional *oblivious* [10] property intrinsic to AOP, in particular languages such as AspectJ [20]. Thus, by allowing AOP authors to construct their specifications in a parameterized fashion, and to further constrain the behavior of intangible advice (which would constitute the *actual* parameters), we are indeed forcing them to be at least cognitive of crosscutting concerns (CCCs). Nevertheless, it has been shown in [22] that even in (non-AOP) ordinary software, one must still be aware of CCCs, and [37] suggests that designing components subject to advice also requires the cognition of CCCs.

Modelling. How do we model specifications that abstract enough information from the internal details of components while simultaneously constraining the effects of potentially applicable advice that manipulates the internal implementation of these components? We will see later in this paper how our proposed approach, with the help of *traces*, may allow us to write such specifications for AOP.

Composition. Given the specifications of a component and its applicable aspects, how do we decide if the constituent advice is applicable especially considering that advice may be bound to lexical and dynamic pointcuts? A reasoning formalism should account for such situations if it intends to deal with lexical pointcut designators (LPCDs), e.g., `within()`, and/or dynamic pointcut designators (DPCDs), e.g., `cflow()`, `if()`. Then, given that advice is applicable to component code, how do we utilize the specifications of the advice and that of the parameterized component specifications to arrive at the overall behavior of the system, thus verifying that the software behaves as intended? Conversely, how do we derive the behavior of a component in which no advice is applicable given the nature of the components specifications? In fact, the schematic in Figure 1 is somewhat misleading as it fails to mention the situation where no advice is applicable. That is, in order to derive the behavior of a component that is not under the influence of advice we must either (i) obtain the behavior of a component `C` taking its parameterized specification and then composing it with a "empty" aspect specification $\mathtt{A}_{null}$ (portrayed in

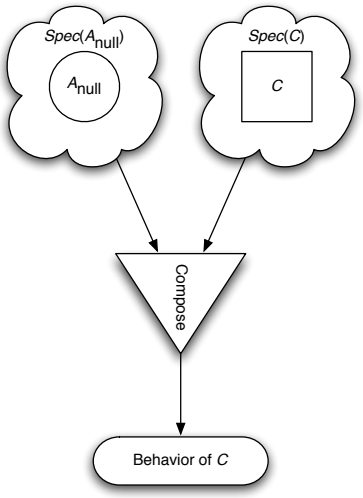

Figure 2: Schematic of behavior derivation using parameterized specifications of a component `C` not under the influence of advice.

Figure 2), or (ii) supply a second specification for each component which would correspond to the situation where no advice is applicable.

The focus of this paper is to (i) present new ideas in our ongoing work in this area, (ii) discuss our proposals to combat several of the above mentioned obstacles, and (iii) facilitate interesting discussion in respects to some of the open issues that faces our approach, in which we highlight throughout the paper.

2. CONSTRAINING AND ENRICHING THE BEHAVIOR OF PROGRAMS

Several approaches [1, 12, 13, 7] have been developed to restrict the behavior of AO components in order to reduce the efforts required in reasoning –both formally and informally– about such systems. In this paper, however, as mentioned in section 1, we are interested in specifying the intended runtime behavior of components under the possible influence of advice that accounts for the unique evolutionary nature of AOP; hence, assertions of these components should reflect this notion. Such a solution seems to call for a technique that would to need to be more flexible than existing approaches in the way that constraints on acting advice are expressed. In this section, we will briefly discuss how behavior of processes within concurrent programs are suitably constrained to achieve *interference freedom* using the rely/guarantee approach. This section will draw a parallel with concurrent programs and that of AOP, outlining our previous work in adapting the rely/guarantee approach for AOP. In addition, we will overview how assertions for evolving AO programs can be written, and how specifications can be composed to arrive at the *effective* behavior of the overall system, that is, the behavior of the components augmented with the behavior of applicable aspects.

Constraining behavior of concurrent programs. Consider a concurrent program with two processes P_1 and P_2 that share some variables that either of them may read and write. Standard modular reasoning would require us to reason about each process independently of the other and then combine the results of the two reasoning tasks in an appropriate manner to arrive at the behavior of the whole program, $[P_1//P_2]$. But since the two processes will be interleaved during execution, whatever conclusions we may have drawn about each of them when reasoning about them independently may not, in fact, be valid. In effect, the actions of each process may *interfere* with the other process thereby invalidating whatever results we may have established by reasoning about that other process.

The *rely-guarantee* approach [18, 39] addresses the problem of interference in concurrent programs as follows. Let σ be the state, i.e., the set of all program variables of the program consisting of two processes P_1 and P_2 running in parallel. When reasoning about P_1, we recognize that the actions of P_2 may modify the state. Hence, we write our assertions in the proof outline of P_1 in such a manner that they continue to be satisfied even in the presence of such actions. To enable this, we identify a relation $rely_1()$ that is a predicate over two states, σ_a and σ_b. This relation means the following: suppose at some point in the execution of P_1 the current state is σ_a and that some part of P_2 is now interleaved in the execution; suppose that the state when P_1 gets control back is σ_b; then $rely_1(\sigma_a, \sigma_b)$ must be satisfied. In other words, when reasoning about the behavior of P_1, we assume that any interleaved action that P_2 (or any other process in the case of programs with more than two processes) may change the state but *only within the constraints* specified by $rely_1()$. If this is satisfied, the correctness of the proof outline of P_1 will not be affected by the actions of P_2. Conversely, when reasoning about P_2, we introduce a relation $rely_2()$ that imposes constraints on the changes in the state that may be caused by P_1's actions.

Next, we must verify that P_2 and P_1 meet the requirements contained respectively in $rely_1()$ and $rely_2()$. To make this possible, when reasoning about each process, we establish a *guarantee* clause. This clause, denoted $guar_1()$ in the case of P_1, is again a relation over two states; it is a guarantee provided by P_1 that any change it makes in the state when executing any instruction in it will obey the constraints specified in $guar_1()$. The specification of P_1 is of the form $(pre_1, rely_1, guar_1, post_1)$ which denotes: if P_1 starts in a state that satisfies pre_1 and if all transitions, i.e., state changes, made by P_2 satisfy the constraints specified in $rely_1()$; then each transition made by P_1 will satisfy the constraints specified in $guar_1()$, and the state, when P_1 finishes execution, will satisfy $post_1()$. The *parallel composition* rule requires us to check, using $guar_1()$ and $guar_2()$, that the *rely* clauses of both processes are satisfied.

Constraining behavior of AO programs. As we noted earlier, reasoning in concurrent programs seems to have some resemblance to reasoning about AOP. Suppose, for example, that the code of a component, say a class `C`, subject to advice contains an assignment statement assigning a value v returned from a method call `m()` on an object `obj` to a particular instance variable `x` of `C`. When reasoning about the code of `C`, we might have established an assertion following the assignment that states that the value of `C.x` would be equal to v. Suppose now that an aspect is added that encompasses a piece of *after*-advice that applies at the *call*-join point associated with the invocation of `obj.m()`. Immediately following the execution of the assignment of the returned value v to

C.x, the *after*-advice would execute and, possibly, invoke a mutator method of C that assigns a new value to C.x. When control returns to C, at this point, the assertion we previously established may no longer satisfied. In other words, the aspect has *interfered* with the component code.

While this seems highly analogous to the case of the concurrent program, there are notable differences between the situation in AOP and that of concurrent programs. Firstly, AO programs are intrinsically sequential, making the interleaving of their constituent statements more predictable. Nevertheless, when reasoning about components independently we must recognize that advice may be weaved in, out, or around each join point, thereby detracting this otherwise innate predictability. Secondly, the severity of possible interference is governed by the join point model of the underlying AO language. Possible interference is thus dictated by the types of join points, the control structures, and the mutable contexts that are exposed and available to advice to manipulate. Thirdly, there is an asymmetry between components that does not possess the ability to advise other components, e.g., a method within a class in AspectJ (sans annotation-based mechanisms such as @AspectJ), and components, e.g., aspects, that do possess this ability. In particular, while advice can *intercept* the execution of a class, a class does not intercept the advice. As such, control is solely at the mercy of aspects as opposed to other paradigms like concurrent programming, and coroutines [6] where control is explicitly released and suppressed at various points. Lastly, in the case of parallel programs, concurrent processes are typically designed hand-in-hand, while in AO programs aspects may be added, removed, and/or changed to a system at unpredictable intervals as the software evolves.

A key observation underlying our approach is that assertions contained within component subject to advice should be in the form of a relation over two states σ_a and σ_b. Here, σ_a refers to the state of the advised component prior to control transferring to advice, and σ_b refers to the state of the advised component immediately following the point where it reacquires control. Therefore, components subject to advice can effectively detail the sorts of constraints on advice behavior required for it to behave properly regardless if advice is applicable at the moment. Principally, when reasoning about a component C we recognize that its behavior may be modified as a result of aspect(s) being applied to it. As C executes, if control were to reach a *join point* that matches a *pointcut* at which a particular advice is applicable, control will transfer to the advice before, after, or around (potentially bypassing) the statement at that point. The advice would then execute, possibly changing the values of some of the instance variables of C and/or other accessible parameters. Finally, control would then return to C which would continue execution.

The approach discussed in this paper is based on augmenting leverages a existing technique made for improving modularity in AO programs. We extend the notion of *pointcut interfaces* [13] by annotating pointcuts with associated specifications that must be met during the execution of the matching join points by both the component (through a *guar* clause) and applicable advice (through a *rely* clause). We will discuss related work in more detail in section 4, even so, it is worth mentioning here that the contractual obligations between advice code and *advised* code is similar in spirit to Crosscutting Interfaces (XPIs) [12], however, our interest lies in establishing run time *behavioral* properties exhibited at compile time, i.e., through use of an axiomatic proof method.

Deriving effective behavior. Unlike concurrent programs where a prime concern is preserving process interference freedom [32], the of addition of aspects typically corresponds to *enriching* existing program behavior. Indeed, it is the possibility of such enrichment that is the source of much of AOP's power. For this purpose, we introduce the concept of *join point traces* (*JPT*s). A JPT is used when reasoning about a component C under the potential influence of advice, to record the flow-of-control through various join points contained within C. These join points are the ones "exposed" by the pointcut interface of C, where items of advice may be applied to enrich or otherwise affect its behavior. We will delve into the details of the structure of JPTs in section 3, but the central idea is to specify the behavior of C in terms of assertions involving not just the variables of C but also *abstractly* in terms of the state changes caused by various items of advice that could possibly be applied at the various join points recorded in C's JPT, *without* referencing the actual advice. When reasoning about C, we will not, of course, know what these state changes will be since the aspect(s) in question may not yet have been constructed (or even if they have been, we have not yet reasoned about them). Hence, in our reasoning, we have to allow for a range of possible state changes –subject to the constraints of the appropriate *rely*() clauses– that these items of advice may carry out; essentially the assertions characterizing the behavior of C will allow for various such changes and, corresponding to each, specify how C will behave. In effect, the behavior of C will be *parameterized* with respect to the possible behaviors that each item of advice code may engage in at the various join points, with the JPT being used to record the "parameter values" representing these behaviors. The next step, given a particular set of advice specifications, is to *compose* our JPT-based specification of C with the specified behaviors of the aspects to arrive at the resulting enriched behavior of the composed system as illustrated earlier in Figures 1 and 2. Formally, this will be carried out by appealing to our rule of composition of aspect and the components they advise.

3. SPECIFICATION AND VERIFICATION

In this section, we will explore possible ways to specify and curtail the behavior of AO programs in order to improve reasoning in these systems that is natural to the way they evolve, intuitively similar to what is portrayed in Figure 1. We will then examine several inference rules using a highly distilled version of an AspectJ-like AO language that will allow us to show that the *composition* of AO components meets a certain specification. Our goal in this paper is not to provide a complete formal set of rules but rather to indicate the types of considerations involved in them. In future work we intend to define the syntax for a complete but simplified version of AspectJ, present its operational semantics, extend our set of proof rules to apply to this language, define a formal operational model based on the notion of JPTs, and address questions about soundness and completeness of the rules with respect to the model.

Specifications and pointcuts. To demonstrate the crux of our proposal, we will only consider *call*-join points and *after* advice. *Before* advice could be theoretically handled in a symmetric manor; *around* advice, however, poses some in-

teresting complications as advice could alter both the calling and callee objects and avoid the execution at the join point entirely by opting not to call `proceed()`. We leave *around* advice as a problem open to discussion with the possibility of leveraging existing work from [4]. For further simplicity of the presentation, we will also not consider such constructs as lexical pointcut designators (LPCDs) but will consider them in future work.

The flexibility and expressiveness that we desire with our specifications may lead to undesired complexity since many join points may be traversed as a result of a given method invocation. This complexity, however, depends heavily on the strength of the associated *rely* clauses as any applicable advice must respect it. Thus, although our specifications will be over the behavior any applicable advice, we do not need to consider potential behavior that does not abide the *rely* assertion. Another possibility would be to follow the conventions in [1], allowing only external calls to methods within a component `C` listed on the interface of `C` to be subject to advice, that is, `C` is "sealed." Pointcuts appear on `C`'s interface in order to export important internal events within `C` that the author of the component feels aspects may be interested in advising. Note that importantly the author does not examine existing aspects to come to this decision, instead, she solely examines `C` to determine which internal events should be exported on its interface. In much the same way, in the context of our proposal, the author of the component determines the necessary constraints to place on possible advice that would apply to the it based on the internals of that component alone, deciding what essential constraints are necessary to place on the behavior of advice either currently in existence or to be developed in the future. Moreover, as another possible extension to our approach, it may be worthwhile to break the sealing of a component for *observer* aspects [3]. That way, less intrusive aspects, e.g., logging aspects, would be allowed to advise the execution of the entire program. We leave both of these issues open for discussion.

A common challenge with providing a reasoning scheme for software that contains objects (and aspects) is aliasing, which tends to cloud the vision as to what an object's state precisely consists of. A component `C`, say a class for instance, will in general define a number of (instance) variables. Some of these will be of primitive types (`int`, `boolean`, etc.), others will be of reference types. Consider an instance `obj` of `C`. The state of `obj` at any time will consist of the values of the variables of primitive types plus the values of references to objects. Generally we will not consider the states of objects that `obj` contains references to as part of the state of `obj`. Only changes to the values of its primitive variables resulting from execution of methods invoked on `obj` will be reflected as changes in the state of `obj`. As these methods execute, they will in general invoke methods on objects that `obj` has references to, resulting in changes in the states of those objects; these latter changes are not part of the changes in the state of `obj`. In effect, we are assuming that there is a *heap* in the background that holds all the objects, retains their current states, and makes them available to us as needed. These considerations are, of course, common to reasoning frameworks for all object-oriented languages. Hence we will not consider them in any detail when presenting our formalism.

Join point traces. From this point forward we will typically consider the situation where we wish to specify and verify a method in a class that may potentially be under the influence of advice. It is not inconceivable to conversely apply our proposal to reasoning about a piece of advice in an aspect which itself may be open to advice (possibly even itself), however, as noted earlier, *around* advice does pose several interesting problems (e.g., constraining the behavior of calls to `proceed()`). For now, however, consider, in general, a method `m()` of a class `C`. The JPT for this method will record the flow-of-control through the various join points in the body of `m()` where items of advice defined in various (perhaps yet-to-be-developed) aspects may apply. Each join point is a call to a method either of the same class `C` or of a different class. Note that we do not have to worry about *every* call that appears in the body of `m()`. Suppose there is a call to a method `n()` of a class `D`. If `n()` appears in an exported pointcut on the pointcut interface of `D`, then this call, in our current approach, will be recorded on the JPT of `m()`. If not, however, it will not be record since this call in this case would have no advice applicable to it. The designers of `D` are responsible for deciding whether or not calls to `D.n()` should be included in one of the pointcuts of `D`. Our specifications, using JPTs, will reflect these decisions.

Traces of various kinds have been widely used for specifying the behaviors of different types of systems ranging from ADTs [16, 17] to processes in a distributed system [15, 28]. In each case, elements in the traces are used to record information about important events in the system; the ordering of the elements in the trace represents the order in which the corresponding events took place. Specifications of the systems are written in terms of conditions that must be satisfied by the structure of the trace and by the information recorded in the individual elements. In the case of JPTs, the events of interest are the arrival of control at various join points. Since the only kind of join point we are considering is the call-join point and since the only type of advice we are considering is *after* advice, each element of the JPT will correspond to the completion of a call. In order to deal with DPCDs such as `cflow` ad `cflowbelow`, it is also convenient, from the point of view of specifying our methods, to record on the JPT the events corresponding to the *start* of method execution as well as its end.

Let us now consider the structure of the elements that appear in a JPT. Consider the completion of a call of a method `m()` of a class `C` invoked on object `obj`. Suppose we have a pointcut `pc` defined in the pointcut interface of `C` that includes calls to `m()`. Suppose `A` is an aspect that includes an (*after*) advice that applies to this pointcut. We will assume that any variables defined in `A` will be of primitive types. The code of the advice may update the values of these variables and also update the state of `obj` by updating values of the primitive variables in that state. For simplicity in the presentation, in this paper, we will not consider the possibility of `A` invoking additional methods. Trace models are, in general, powerful enough to handle such complications but the resulting specifications tend to be rather complex. The exclusion of such calls means we do not have to worry about additional items of advice associated with calls to such methods being triggered. Nevertheless, we encourage open discussion on how some of these restrictions may be relaxed.

During actual execution of the system, if an aspect such as `A` considered above had been defined, control will transfer to the corresponding advice code. That code will execute,

possibly resulting in changes in the state of the aspect as well as in the state of `obj` (e.g., through exposing context at the join point using the `target()` PCD), or even the caller of `obj.m()` (e.g., through exposing context at the join point using the `this()` PCD). Hence, in this element of the JPT, we will record the state of `obj` at the time that control reached the completion of the call to `obj.m()` and its state when control returns from the advice, and likewise for the calling object of `obj.m()`. If there is no applicable advice, either because calls to `obj.m()` are not included in any pointcut, no aspect such as `A` had been defined to apply at the pointcut, or because the conditions for the advice to be applied are not satisfied, these two states will be identical, since the state will not, in this case, change between the time that the call completes and the code of the calling method continues execution.

However, the completion of the call to `obj.m()` is not the only point at which advice may apply. That is, calls made to additional methods within the body of `C.m()` may themselves be subject to advice. As such, the specifications of these method calls will also be parameterized over applicable aspects. Thus, although until now we have been considering an individual method `m()` of a class `C` and described the JPT as if it corresponded to just (a single call to) this method, there is, in fact, a single JPT for the *entire system*. This JPT is initialized, at the start of the system's execution, to the empty sequence. We will use the symbol $\gamma\tau$ to denote this (global) trace. Each time a method is called, an element is added to the JPT to record the start of this invocation. And when the method call completes, an element corresponding to the completion is added to the JPT, in fact, this element corresponds to the join point at that location. The effect of any (*after*) advice that applies at this call-join point is recorded in this latter element. The completion elements of the JPT will have the structure $(oid, mid, aid, args, res, \sigma, \sigma')$ where *mid* is the identity of the method whose call just completed; *oid* is the identity of the object on which the method *mid* was invoked; *aid* is the identity of an applicable aspect instance, *args* and *res* are the arguments and result (if any) of this method. σ and σ' are state vectors as follows. $\sigma[oid]$ is the state of the callee object at the time the method call completed (immediately prior to transferring control to the advice if any) and $\sigma'[oid]$ the state at the time immediately following the completion of the advice code. Likewise, $\sigma[\texttt{this}]$ is the state of the calling object at the time the method call completed and $\sigma'[\texttt{this}]$ the state at the time immediately following the execution of advice. If there is no applicable advice at this join point, then $\sigma[\texttt{this}] = \sigma'[\texttt{this}]$. These state vectors also contain the state of aspects in a similar fashion. Specifically, $\sigma[aid]$ refers to the the *aspect state* immediately following the completion of the execution of the method, whereas $\sigma'[aid]$ is the aspect state immediately following the completion of the advice code. If there is no applicable advice at this join point, these elements will not be included. We should note again that the above description is an *operational* picture of the JPT.

Let us now consider the structure of the elements of the JPT that record method call invocations. Consider again the call of a method `m()` invoked on object `obj`. If we wanted to account for *before* advice that might apply at this point, the element of the JPT recording this invocation would have to include information very similar to the above. In this paper though, we are only considering `after` advice, hence we can omit much of this information. The only information that does need to be included are the identities of `m()` and `obj` since these may be used to control the applicability of some advice, especially those point to pointcut expressions containing DPCDs.

Inference rules. We will consider three rules corresponding respectively to accounting for the advice that may apply when a method call completes; for the call a method body makes to another method; and for combining the specification of a method `C.m()` with that of the aspects that apply, including those that apply to the various methods that `m()` calls during its executions to arrive at the resulting "enriched" behavior of `m()`. Let us first consider the rule depicted in Figure 3 corresponding to completion of a method call to `C.m()`. The specification of such a method will be a 4-tuple, $(pre, post, guar, rely)$. Here, *guar* is obtained from the conjunction of each *guar* clause of each exported pointcut which corresponds to calls to `C.m()`. *rely* is obtained in a similar manor except that it is derived from the disjunction of each *rely* clause. This specification annotation means that if *pre* is satisfied when `C.m()` is called; and if the methods called in the body of `C.m()` satisfy their respective specifications –these calls may be to methods of `C` or other classes or both–; and if any advice applicable to calls to `C.m()` satisfy the *rely* clause; then when body of `C.m()` finishes execution, it will satisfy *post* as well as the requirements specified in the *guar* clause. Note that the post-condition will, in general, involve the JPT since that is what will allow us to later enrich this specification, accounting for the action of advice defined in any aspect that may be developed to apply to calls to `C.m()`. But this JPT is not the global JPT, $\gamma\tau$; rather, it corresponds to a single execution of this method and we will use the symbol $\lambda\tau$ to refer to it.

$$\frac{\begin{array}{c} pre \wedge [\lambda\tau = \langle (inv, \texttt{C.m}) \rangle] \Rightarrow p \\ \{p\} S \{q\} \\ q \Rightarrow guar(\sigma[\texttt{this}]) \\ \big[q \wedge rely(\sigma[\texttt{this}], \sigma'[\texttt{this}])\big] \Rightarrow \\ post[\lambda\tau \leftarrow \lambda\tau^\frown(\texttt{this}, \texttt{C.m}, ?, args, res, \sigma, \sigma'),\ \sigma \leftarrow \sigma'] \end{array}}{\texttt{C.m} :: \langle pre,\ post,\ guar,\ rely \rangle}$$

Figure 3: Rule for method specification.

S is the body of the method, with pre-condition p. This differs from *pre* since *pre* does not give us any information about $\lambda\tau$; thus the first line essentially tells us that when the method body starts execution, $\lambda\tau$ has been initialized to contain the element representing the start of the invocation (*inv*) of this method (`C.m()`. The post-condition of S is as denoted q. The second line of the rule requires us to show that when S completes, q will indeed hold. Moreover, `C.m()` will, in general, provide a guarantee to any advice that may apply when the call to `C.m()` completes. This guarantee is represented by *guar* and this has to be satisfied when S finishes; i.e., the post-condition q of S must imply this assertion.

The next requirement, split over the next two lines of the rule, essentially allow us to go from the post-condition q of the body of the method to *post*, the post-condition of the method. The difference between these two assertions arises

because the JPT has an extra element added to it to represent the completion of this method; and the state might be modified from σ to σ' as a result of an advice that has been defined to apply at the completion of a call to this method. Any such advice is required, as indicated in the third line, to satisfy the *rely* clause. The element being appended to $\lambda\tau$ at this point corresponds to the completion of this method call. The first element denotes the fact that the object in question is simply the *this* object. The *aid* element represents the identity of the aspect, if any, that may apply at this point. Since we are only considering the method at this point, we have no requirements with respect to an aspect state, hence the question mark. But the state of the current object may itself change; this change is represented by the elements denoted $\sigma[\texttt{this}]$ and $\sigma'[\texttt{this}]$; these elements may, as just noted, be assumed to satisfy the *rely* clause of this method (since if not, the aspect is considered unacceptable). Once we have added this method completion element to the JPT, we change the state of the object to whatever the aspect assigns to it. The rule requires us to show that, following this assignment, *post*, the specified post-condition of the method is satisfied.

The rule looks rather involved but much of it is *notational* complexity. Intuitively, the rule may be summarized by saying, it requires us to show that the body S of `C.m()` behaves according to its specification; and that when S finishes, the state satisfies the requirements specified by the *guar* clause so any *after* advice that is defined can legitimately assume this clause. The rule also requires us to take account of the fact that when the method is invoked, before the body starts execution, the trace must be appropriately initialized. And when the method finishes, the trace must be finalized by adding a completion element that also records the effect of any advice that may be applied corresponding to calls to this method.

Next let us consider the rule portrayed in Figure 4 for dealing with a method call. Suppose the method invocation is of `obj.m(args)` where `m()` is a method of the class `C`. Let us assume that the specification of `C.m()` is $\langle pre, post, guar, rely\rangle$. Let us further assume that the (local) JPTs for the calling and called methods are named $\lambda\tau_1$ and $\lambda\tau_2$, respectively. At the start of the method, we have to account for the fact that the object `obj` of the calling method will play the role of the `this` object of the called method and substitute the actual arguments for the formal parameters of `C.m()`. We let $p[\overline{x}/\overline{e}]$ denote p where all free occurrences of a variable in the list $\overline{x}$ are replaced by its respective expression in $\overline{e}$. When the method finishes, we have to substitute in the reverse direction *and* append $\lambda\tau_2$ of `C.m()` to $\lambda\tau_1$ of the caller.

$$\frac{p \Rightarrow \texttt{C.m}.pre[pars/args] \qquad \texttt{C.m}.post \Rightarrow q[\lambda\tau 1/\lambda\tau 1\hat{}\lambda\tau 2,\ args/pars]}{\{\ p\ \}\ \texttt{obj.m(args)}\ \{\ q\ \}}$$

Figure 4: Rule for method call

Essentially, the JPT records appropriate information about the various (potential) join points through which control flows. The value of $\lambda\tau_1$ at the time of the method call, i.e., just prior to control being transferred to `C.m()`, represents all join points we have encountered thus far in the calling method. Now control continues in the body of `C.m()`. As that body is executed, additional methods may be called and information about the corresponding call-join points should be accumulated in $\lambda\tau_2$, the local JPT of `C.m()`. But this is, in fact, simply a record of the additional join points that we are encountering as execution continues. Hence, as control returns from `C.m()` to its caller, we need to append this record to the JPT that we already had immediately prior to the call, i.e., to $\lambda\tau_1$. This will ensure that, in $\lambda\tau_1$, we will have a *complete* record of the control-flow along join points that occurs during the entire execution of the caller of `C.m()`, *including* the flow that occurs during the execution of the methods that are called.

The final rule (Figure 5) we consider is for *applying* an aspect to a class, in particular to a class method, to arrive at the resulting enriched behavior of the method. More precisely, this aspect has been defined to apply at a pointcut that includes the call join point to `C.m()` and we want to arrive at the enriched behavior of this method as a result of this aspect. Let us first consider a simpler form of the rule ignoring the possibility of DPCDs. `A` refers to the aspect being applied and $\texttt{A}_{adv}$ is the applied advice defined in the aspect.

$$\frac{\{\ guar(\sigma) \wedge ap\ \}\ \texttt{A}_{adv}\ \{rely[\sigma/\sigma@pre, \sigma'/\sigma] \wedge\ aq\ \} \qquad \texttt{C.m} :: \langle pre,\ post,\ guar,\ rely\ \rangle}{\{pre \wedge ap\ \}\ \texttt{C.m}() + \texttt{A}\ \{post \wedge aq\ \}}$$

Figure 5: Rule for aspect application (simple version)

In this rule, the *rely* and *guar* clauses are part of the specification of `C.m()`. The $\sigma@pre$ notation denotes the state at the start of the execution of this code. The first line thus requires us to check that this code meets the *rely* and *guar* clauses. In particular, the post-condition of $\texttt{A}_{adv}$ ensures that the *rely* clause with $\sigma@pre$ playing the role of the starting state of the clause and σ playing the role of ending state, is satisfied. The *ap* and *aq* are assertions over the state of the *aspect*. The second line simply states that we have already established the required result about `C.m`(). The post-condition in the conclusion of the rule shows us the effect of the enrichment resulting from the application of the aspect.

There are some problems with this rule. First, the pre-condition requires not only the expected pre-condition of the *method* is satisfied but also, *ap*, which is a condition over the aspect state. The value of this state would not be affected by the execution of the body of `C.m`(), so if it is satisfied at the start of the execution of this body, it will also be satisfied when the code $\texttt{A}_{adv}$ starts execution. But how will we ensure that this condition is, in fact, satisfied at the start of `C.m`()? This state is going to be modified only as a result of execution of various pieces of advice code of this aspect, not by the body of `C.m`(). Nevertheless, there is nothing in the pre-condition of `C.m`(), as we have it so far, that will ensure that the aspect state, as it existed at the start of `C.m`(), satisfies this assertion. Hence we need to add this as an additional part of the pre-condition of `C.m`(). Therefore, this information needs to be provided as part of the *invocation* element that is added to the JPT at the start

of C.m().

However, although the execution of the body of C.m() will not directly modify the state of the aspect, there may be calls in this body to other methods, and those methods might be subject to this same aspect as well. Thus the state of the aspect when the execution of the body of C.m() completes and control is transferred to the advice code may not be the same as it was when C.m() started execution. Instead, it will be whatever it was when the most recent such call finished execution. These calls will, of course, be recorded on the JPT as per our method call rule. Further, the effect of the advice acting on the called methods will result in the aspect state that exists at the end of each such call to be recorded on the JPT. We can address these considerations by making two changes to the above rule. First, we modify the pre-condition of A so that the assertion *ap* applies to the state of the aspect as recorded in the invocation element of the JPT. Second, we need to modify the post-condition so that the assertion *aq* applies to the (final) aspect state recorded in the JPT when this method returns to its caller.

The third problem with the rule has to do with bound pointcut expressions containing DPCDs. The rule above assumes that the advice code $\mathtt{A}_{adv}$ will apply to the execution join point of C.m(). But it may not. Or, rather, we may have a condition that depends on the call stack (as is the case with such DPCDs as `cflow`) that will determine whether or not it is applicable. And this, of course, cannot depend on anything that is contained in the body of C.m() nor can it be specified as part of the *pre-condition* of C.m(). Instead, it will depend on the state of the call stack for each call to C.m() that we have to deal with in reasoning about the behavior of the overall system. In order to handle this, when reasoning about C.m(), if the pointcut associated with this advice is dynamic, we will allow for both possibilities – when the associated condition *is* satisfied and when it is *not* satisfied. For this purpose, we will have *two* (possibly) distinct post-conditions with the method, corresponding respectively to the cases when the method is called with the state of the call stack satisfying the condition of the dynamic pointcut and when this condition is not satisfied. These are marked, in the rule depicted in Figure 6, with the labels d and $\neg d$ respectively, d being the condition specified in the dynamic pointcut for deciding whether or not the advice should apply when the execution of C.m() finishes.

$$\frac{\{\ guar(\sigma) \wedge ap\ \}\ \mathtt{A}_{adv}\ \{rely[\sigma/\sigma@pre, \sigma'/\sigma] \wedge\ aq\ \} \quad \mathtt{C.m} :: \langle pre,\ post,\ guar,\ rely\ \rangle}{\{pre \wedge ap\ \}\ \mathtt{C.m}() + \mathtt{A}\ \{\langle d : post \wedge aq\ ,\ \neg d : post \wedge ap \rangle\ \}}$$

Figure 6: Rule for aspect application (revised version)

There is one final complication – the combination of the two problems identified above. That is, the items of advice applicable to methods called within the body of C.m() may *also* have dynamic pointcuts associated with them! This means the effect of dynamic pointcuts is not going to result in just *two* possibilities in the post-condition of C.m() but rather all possible combinations of the conditions corresponding to these various dynamic pointcuts being or not being satisfied! In the worst case, this would give rise to 2^n possibilities, n being the number of calls in the body of C.m(). What this tells us is that while dynamic pointcuts are undoubtedly powerful, using them too liberally can lead to systems that are extremely difficult to specify or reason about. In fact, it is precisely this problem that forced Krishnamurthi *et al.* [23], in their model checking approach, to introduce a *depth* parameter that is used as threshold to combat this explosion. This problem is indeed more general as illustrated above as it would also be encountered when a join point resides either in a loop or is traversed multiple times as a result of recursion. We will not present a formal version of the rule that accounts for this problem as a solution is currently being investigated.

4. RELATED WORK

Several authors have proposed restrictions to AOP in order to address the complexity of the associated reasoning. Clifton and Leavens [5] present MAO, a language that extends AspectJ [20] with concern domains and control-limited advice. MAO, via static analysis, allows developers to restrict the behavior of advice, e.g., to allow accesses to only certain parts of the heap belonging to a particular *concern domain*. MAO also allows for restricting the manipulation of control-flow by advice thereby forbidding it to perturb the control-flow of the base-code in inappropriate ways. Such restrictions are expected to help simplify reasoning about AOP since developers can examine the *signatures* of each advice declaration to reason about its potential effects. Similar restrictions are also conceivable using our proposed *rely* clauses; however, our advice restrictions are more flexible and fine-grained in that *rely* clauses take an arbitrary assertion over two states σ and σ', the state at the point in which advice obtained control and the state corresponding to the point when it released it, respectively. Furthermore, MAO does not provide the proper facilities to *combine* the specifications of the base-code and the advice to arrive at the *overall* behavior exhibited by the augmented system. Nevertheless, it should be possible to borrow some of MAO's ideas to help simplify our formalism.

Dantas and Walker [7] propose "Harmless Advice," a restricted form of advice that has minimal effects on the base-code, and develop a type system that enforces such behavior statically. Harmless advice cannot alter state (with the exception of I/O) and control-flow that is visible to the base-code. What is interesting about such advice is that, although highly constrained, it is shown to be quite useful especially in the domain of security. With the use of *rely* clauses, our approach could conceivably be adopted to relax some of the constraints on harmless advice in order to make it more "helpful" while maintaining effective local reasoning. This would allow the base-code developer to explicitly state on a fine-grained level what kinds of advice behavior he or she considers "harmless" by means of a less restrictive *rely* assertion, and then use JPTs to reason about the overall effects of the advice applied to the base-code.

Krishnamurthi *et al.* [23] propose a verification technique which can, using model-checking [2], modularly verify advice independent of the base-code. The proposal, given the base-code represented as a finite-state model, a set of properties that the augmented system (i.e., the base-code combined with the aspects) must satisfy, and a set of pointcuts where potential advice may be applicable, automatically generates enhanced interfaces which can be used for verifying the ad-

vice when it becomes available. Essentially, the interface captures the state of the model checking process prior to advice being added to the system. Goldman and Katz [11] present a related technique using their MAVEN tool. While these approaches, as well as the approach presented in this paper, all employ techniques that do not require repeated analysis of the entire augmented system each time a developer adds, removes, or changes advice, there are several key differences. Firstly, our proposed proof technique relies on deductive logical reasoning while model-checking entails a fundamentally different approach in which an abstract model is exhaustively examined for violations of a certain property. Furthermore, our approach is centered on combining the specifications of the base-code and that of the aspects using JPTs in order to assist developers in obtaining the overall behavior of the system. As such, our proposal does not require a specific property that neither the base-code nor the augmented system must exhibit.

Devereux [9] also attempts to exploit the similarities between AOP and concurrent programs. The approach translates an aspect-oriented program into to an equivalent, low-level concurrent program in an alternating-time logic formalism. The reasoning then is performed on this concurrent program using an assume-guarantee paradigm [14]. The modus operandi is focused on preserving particular properties of the base-code despite the addition of advice. Our approach, through the use of JPTs, on the other hand, allows a developer to reason about the behavior of the base-code *parameterized* over any applicable aspect; therefore, reasoning about the base-code does not need to be reconstructed for *each* property being verified nor a specific property that the base-code must evince. We are interesting in obtaining the enriched behavior of the combined system as opposed to solely verifying the existence of interference freedom [32]. Moreover, transformation from an aspect-oriented program to a concurrent program may cause the task of reasoning about the original program to be more difficult. That is, a change to either the base-code or an aspect could possibly result in previous reasoning efforts being invalidated. Also, assume-guarantee reasoning in concurrent programs are normally leveraged with the acknowledgement that other processes may exist in the system. In AOP, nevertheless, aspects may not even have been developed yet or may be interchanged between different systems. As such, the proposal presented in this paper is designed more towards how AOP is used, especially to the *plug-n-play* capability inherent to aspects.

Several approaches [1, 12, 13, 25] attempt to augment traditional interfaces with various degrees of information regarding crosscutting concerns in order to improve reasoning. In particular, Kiczales and Mezini [22] argue that in the presence of crosscutting concerns we cannot expect to work with the standard interfaces provided by a class' methods and their behaviors. Instead, we must define a more detailed interface for the class that includes information pertaining to how the system is intended to be deployed. These *aspect-aware* interfaces, which include the various join points at which advices defined in the aspects are applicable, accompany traditional interfaces, thus adding to their usefulness.

5. FUTURE WORK AND CONCLUSION

Reasoning, specification, and verification of AO programs indeed presents unique challenges especially as such programs evolve over time. Constructing an approach general enough to reason about components subject to the unpredictable frequency of advice applicability poses many obstacles including but not limited to usefulness, complexity, obliviousness, abstraction, and composition. In this paper, we have presented our ongoing work in developing such a technique that attempts to overcome these obstacles in an effort to enable tractable evolution of AOP. We propose an approach that is aimed at tailoring specifications of these systems to their evolutionary *plug-n-play* nature and enhancing the expressiveness of constraints made on their constituent components.

In future work we intend to extend our set of proof rules to account for many additional AOP mechanisms. We also intend to define a formal operational model based on the notion of JPTs and address questions about soundness and completeness of the rules with respect to that model. One interesting direction for further work is to investigate multiple aspect instances as provided by the *association* facilities (e.g., `perInstance`) of AspectJ. The close relation between an aspect instance and an object should be reflected in reasoning mechanisms expressing the tight connection between the object state and the state of the aspect instance. We also plan to address mechanisms for member introduction and class hierarchy modifications, possibly utilizing techniques employed in [27]. Another possible avenue to explore is the notion of *specification weaving* as it may help in prevailing over some of the aforementioned hurdles. Additionally, we have listed several unresolved issues below in hopes of provoking interested and related discussion.

Execution-join points vs. call-join points: Suppose we have two classes `C` and `D` and there is a call in the body of `C.m`() to the method `D.m`$'$(). Further suppose there is an aspect that contains advice that applies (on calls to) `D.m`$'$(). When reasoning about what the advice code does, we are allowed to assume the *guar* clause given to us as described in section 3, but how it is exactly derived is not yet clear. It seems it should solely be from `D.m`$'$(). But we discuss how, at various points in the *body* of a method, the *rely*/*guar* clause can be assumed (for *rely*) or must be shown hold (for *guar*). That would mean we are referring to `C.m`(). But the advice that applies to the call-join point associated with the call to `D.m`$'$() is not concerned with `C.m`(); it is concerned with `D.m`$'$(). Furthermore, it is not concerned with what happens *inside* the body of `D.m`$'$() because it is associated with the call-join point, meaning that the question is only about the state at the *end* of `D.m`$'$(). Thus, is there a need to consider the *rely*/*guar* clauses in the middle of various methods?

Classes vs. objects. We are supposed to be specifying classes but we often treat it as if we are dealing with a specific object with a specific history (of method calls, etc.). Such an approach makes sense when dealing with processes in a concurrent language because each process is an actual run time entity and there is only one instance of a given process; but there can be any number of instances of a given class and the approach presented in this paper does not currently deal with this appropriately.

Heap access. We assume that we can access the states of all the relevant objects in the system. However, our formalism does not have any provision to ensure that the heap is properly updated, etc. That is, we are assuming that there is an operational system "running alongside" that keeps track of the heap and gives us the states of all the objects when-

ever we need them. This notion conflicts, however, with our goal of developing an axiomatic reasoning approach. Additionally, we assume access to the state of the aspects, e.g., *ap* and *aq* in the rules depicted in Figures 5 and 6. However, how is the state of the aspect maintained and how can we access it? This is especially problematic if there are multiple instances of the class that an aspect applies to, or if an aspect applies to multiple classes, because each method applied to each instance of a given class `C` will, potentially, trigger the aspects associated with `C` to execute and have that state modified. Somehow, we will have to keep track of this state; essentially, we are treating the aspect state as if it was part of the "static state" of the class `C` without having, in the formalism, any way to deal with such state. And the situation is, of course, worse for aspects that apply to multiple classes since then this state becomes part of the static state of each of these classes.

Acknowledgments

Many thanks to Gary Leavens of the University of Central Florida for valuable discussion and feedback. We would also like to thank the anonymous referees for their extremely useful comments and suggestions.

6. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *Eur. Conf. Object-Oriented Programming*, 2005.

[2] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[3] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect-Oriented Languages*, 2002.

[4] C. Clifton and G. Leavens. MiniMAO: Investigating the semantics of proceed. In *Foundations of Aspect-Oriented Languages*, 2005.

[5] C. Clifton, G. Leavens, and J. Noble. Mao: Ownership and effects for more effective reasoning about aspects. In *Eur. Conf. Object-Oriented Programming*, 2007.

[6] M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 1963.

[7] D. Dantas and D. Walker. Harmless advice. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[8] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. SyncGen: An AOP framework for synchronization. In *Int. Conf. on Tools and Alg. for Construction and Analysis of Sys.*, 2004.

[9] B. Devereux. Compositional reasoning about aspects using alternating-time logic. In *Foundations of Aspect-Oriented Languages*, 2003.

[10] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Advanced Separation of Concerns*, 2000.

[11] M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects. In *Foundations of Aspect-Oriented Languages*, 2006.

[12] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 2006.

[13] S. Gudmundson and G. Kiczales. Addressing practical software development issues in AspectJ with a pointcut interface. In *Advanced Separation of Concerns*, 2001.

[14] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *International Conference on Computer Aided Verification*, 1998.

[15] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[16] D. Hoffman and R. Snodgrass. Trace specifications: Methodology and models. *IEEE Transactions on Software Engineering*, 1988.

[17] R. Janicki and E. Sekerinski. Foundations of the trace assertion method of module interface specification. *IEEE Transactions on Software Engineering*, 2001.

[18] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 1983.

[19] R. Khatchadourian and N. Soundarajan. Rely-guarantee approach to reasoning about aspect-oriented programs. In *Software Engineering Properties of Languages and Aspect Technologies*, 2007.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Eur. Conf. Object-Oriented Programming*, 2001.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Eur. Conf. Object-Oriented Programming*, 1997.

[22] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *International Conference on Software Engineering*, 2005.

[23] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2004.

[24] R. Laddad. *AspectJ in Action*. Manning, 2003.

[25] K. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 2003.

[26] M. Lippert and C. Lopes. A study on exception detection and handling using AOP. In *International Conference on Software Engineering*, 2002.

[27] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, 2006.

[28] J. Misra and K. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 1981.

[29] K. Ostermann. Aspects and modular reasoning in nonmonotonic logic. In *Foundations of Aspect-Oriented Languages*, 2007.

[30] K. Ostermann. Reasoning about aspects with common sense. In *Int. Conf. Aspect-Oriented Software Development*, 2008.

[31] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Eur. Conf. Object-Oriented Programming*, 2005.

[32] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 1976.

[33] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Int. Conf. Aspect-Oriented Software Development*, 2003.

[34] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 2003.

[35] N. Soundarajan, R. Khatchadourian, and J. Dovland. Reasoning about the behavior of aspect-oriented programs. In *IASTED Int. Conf. Softw. Eng. and Apps.*, 2007.

[36] F. Steimann. The paradoxical success of aspect-oriented programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

[37] K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[38] J. Xu, H. Rajan, and K. Sullivan. Aspect reasoning by reduction to implicit invocation. In *Foundations of Aspect-Oriented Languages*, 2004.

[39] Q. Xu, W. de Roever, and J. He. Rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 1997.