

Rely-Guarantee Approach to Reasoning about Aspect-Oriented Programs

Raffi Khatchadourian

Computer Science & Eng., Ohio State Univ., Columbus,
OH 43210, US
khatchad@cse.ohio-state.edu

Neelam Soundarajan

Computer Science & Eng., Ohio State Univ., Columbus,
OH 43210, US
neelam@cse.ohio-state.edu

Abstract

Over the last few years, the question of reasoning about aspect-oriented programs has been addressed by a number of authors. In this paper, we present a *rely-guarantee* approach to such reasoning. The rely-guarantee approach has proven extremely successful in reasoning about concurrent and distributed programs. We show that some of the key problems encountered in reasoning about aspect-oriented programs are similar to those encountered in reasoning about concurrent programs; and that the rely-guarantee approach, appropriately modified, helps address these problems. We illustrate our approach with a simple example.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification

General Terms Languages, Verification

Keywords Rely-guarantee, modular verification, aspect-oriented software

1. Introduction

Kiczales *et al.* proposed *aspect-oriented programming* (AOP) [10] as an approach to enable modular implementation of *cross-cutting concerns*. Since its introduction, various authors [4, 13, 14] have shown how AOP may be used to write modular code for such concerns as synchronization, logging, etc. At the same time, several authors [1, 2, 11, 12] have noted that *reasoning* about aspect-oriented programs presents some key challenges. 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 serious difficulties for reasoning about the behavior of AO programs. The main point is that, since the addition of an aspect can potentially change the behavior of the base code, whatever reasoning we may have done about the base code may no longer be valid; and we may be forced to reason about the entire system accounting for the interleaving of the various pieces of advice with the base-code. Various authors have considered ways to address this problem and we will briefly discuss some of these approaches later in the paper. In this paper, we develop an approach to reasoning about the base-code and the aspects that seems to offer important advantages over the existing approaches.

The problem of reasoning about AOP programs has some resemblance to reasoning about *concurrent* programs. Consider a simple concurrent program $[P_1 // P_2]$, where P_1 and P_2 are two parallel processes that share some variables that both of them may read or write. Standard modular reasoning would require us to reason about, say, P_1 and P_2 independently of each other and then put the results of the two reasoning tasks in an appropriate manner to arrive at the behavior of $[P_1 // P_2]$. But since these 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 can *interfere* with the other process thereby nullifying whatever conclusions we may have arrived at by reasoning about that other process. This is quite similar to the situation in AOP. Suppose, for example, that the base-code contains an assignment statement, assigning a specific value vv to a particular instance variable xx . When reasoning about this base-code, we might have established an assertion following the assignment that states that the value of xx would, in fact, be equal to vv . Suppose now we add an aspect that includes a piece of *after* advice that applies at a *set* joinpoint of the base-code and that the variable xx is one of the affected variables. Now, immediately following the execution of the assignment of vv to xx , the *after* advice would execute and, possibly, assign a new value to xx before returning control to the base-code. At this point, the assertion we previously established in the base-code is no longer satisfied. In other words, the aspect has *interfered* with the base-code in a similar manner as processes of a concurrent program interfering with each other.

The *rely-guarantee* approach [16] addresses the problem of interference between processes of a parallel program as follows. Before we consider the relevant details of the approach, let us introduce some notation. Let σ denote the state, i.e., the set of all program variables of the program. Particular states in which each variable has a particular value will be denoted $\sigma_a, \sigma_b, \dots$ etc. When reasoning about an individual process, say, P_1 , we recognize that the actions of the other process¹ may interfere with the state. Hence, we need to 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 ; suppose 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

¹The rely-guarantee approach allows us to deal with concurrent programs with any number of processes, not just two. For our purposes though, it is enough to consider how the approach works for the case of two processes.

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 $rely_1()$. Next, we ensure that any assertion p that we use in the proof outline of P_1 is *stable* with respect to $rely_1()$ in the following sense:

$$[p(\sigma_a) \wedge rely_1(\sigma_a, \sigma_b)] \Rightarrow p(\sigma_b)$$

This ensures that if, following some actions of P_1 , when control first reaches the point where p appears in the proof outline of P_1 the current state satisfies the assertion p , any changes that may result in the state as a result of parts of P_2 being executed will be such that the resulting state still satisfies p . Hence 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 will introduce a relation $rely_2()$ that imposes constraints on the changes in the state that may be caused by interleaving of P_1 's actions.

How do we 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 must *establish a guarantee* clause. This clause, in the case of P_1 we will denote as $guar_1()$, 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 this clause. Thus the specification of P_1 will be of the form $(pre_1, rely_1, guar_1, post_1)$ which denotes the following: 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 $post_1$. Given such specifications for P_1 and P_2 , the *parallel composition* rule requires us to check that $guar_1()$ *implies* $rely_2()$ and that $guar_2()$ *implies* $rely_1()$. Once we do that, we can conclude the result:

$$\{pre_1 \wedge pre_2\} [P_1 // P_2] \{post_1 \wedge post_2\}$$

The key observation underlying our proposed approach to reasoning about AOP programs is as follows. Suppose $m()$ is a method of a class C in the *base-code*. When reasoning about $m()$, we recognize that its behavior *may* be modified as a result of aspect(s) being applied to it. More precisely, as $m()$ executes, if control were to reach a joinpoint that matches a pointcut at which a particular advice specified in an aspect is applicable, control will transfer to the advice which will execute, possibly changing the values of some of the instance variables of C , and then control returns to $m()$ which then continues execution². To handle this, we will introduce a $rely()$ condition that will specify constraints on the state changes that the advice may perform. This seems analogous to the situation in the case of the two parallel processes, but there is a key difference. When reasoning about $m()$, we do not know exactly what aspect may be applied; indeed, the aspect may not even have been designed yet! Hence, the $rely()$ we use will not correspond to the *actual* behavior of any advice that such an aspect may contain but, rather, specify what kinds of behavior, on the part of such advice, are *acceptable* to $m()$.

In other words, when reasoning about $m()$, we recognize that advice contained in some aspect may apply to $m()$ at some point during its execution. If this advice were to make random changes in the relevant state, i.e., the instance variables of C that $m()$ deals with, clearly the behavior of $m()$ will be seriously affected. To prevent this, the designer of $m()$ considers what kinds of changes might be acceptable to $m()$ and what kinds of changes will not be acceptable. When reasoning about $m()$, we formalize this in its $rely()$ clause. Essentially, the $rely()$ clause is a set of conditions that is being imposed on any aspect that may be developed to apply to

$m()$. If a developer were to introduce an aspect that contains advice applicable at some joinpoint in $m()$ and the changes that this advice makes (to the instance variables of C) do not satisfy the conditions specified in the $rely()$ clause of $m()$, then the specification of $m()$ is no longer applicable. To put it differently, if the advice *does* satisfy the conditions specified in the $rely()$ clause, and if the pre-condition in the specification of $m()$ is satisfied when $m()$ is invoked, we can still be sure that the post-condition listed in the specification will be satisfied when $m()$ finishes execution despite the addition of the aspect in question.

If, when reasoning about the behavior of $m()$, we were extremely lazy, we could simply define $rely()$ as follows:

$$rely(\sigma, \sigma') \equiv (\sigma = \sigma')$$

where σ is the entire state of C , i.e., contains all the instance variables of the class³. This $rely()$ clause essentially *forbids* any applicable advice from making *any* changes in the state! While this may seem drastic, the notion of *harmless advice* proposed by Dantas and Walker [3], not considering I/O manipulation, is precisely of this form. Of course, we do not have to use such a restricted form of $rely()$; for example, later in the paper, we will see what kind of $rely()$ clause would be appropriate for a class of aspects that Clifton and Leavens [2] call *observers*. But there are a number of other issues that we need to consider. First, requiring that *all* of the items of advice that may be applicable at various join points of $m()$ satisfy the same set of conditions, contained in a single $rely()$ clause, may be inappropriate. Hence we need to allow for a more general characterization of these conditions. Second, while the $rely()$ clauses specify conditions that will be required to be satisfied by the behaviors of the various items of advice, in general, those advices may also require certain conditions to be satisfied when they take control. To allow for this, we will use a mechanism that is somewhat similar to the *guarantee* mechanism of the earlier *rely-guarantee* approaches; but we will see that there are some important differences as well. We will consider these and other questions in the next section. Rather than considering the formal details of a full proof system and its axioms and rules, we will, in the next section, present the essence of our approach by means of an illustrative example which is a simplified version of one that has been used in the literature in discussions of reasoning about AOP. In Section 3, we discuss related work. Section 4 concludes with a summary of our approach and presents a number of open issues that remain to be addressed in our approach.

2. Behavior of Base-Code and Aspect-Code

A widely used example in discussions of reasoning about aspect-oriented programs is derived [11] from JHotDraw [7]. This example consists of a base class, `Shape`, two derived classes, `Point` and `Line of Shape`. A cross-cutting concern involves *updating* the `Display` when a `line` or `point` object is moved. The crux is to make certain that when a `line` is moved, the `Display` should be updated only once even though the implementation of `Line.move()` invokes `Point.move()` twice, once for each end-point of the `line`. This is accomplished by using an appropriate aspect.

We will use a simplified version of this example using only a `Point` class as shown in Fig. 1. The field variables `x` and `y` denote the coordinates of the particular point object. `getX()` and `getY()` are used to access the values of the coordinates. The values that these methods return are, however, *scaled* by the scale-factor `s`. The idea is that, in a future version of this class, it is expected that points that are particularly close to the origin may be more easily displayed by adjusting the scale factor suitably. Such a

²That description, of course, assumes the advice in question is a *before* advice. We will consider the general situation later in the paper.

³In this paper, we ignore such complications as *static* variables etc.

scaling concern can be considered *cross-cutting* as various kinds of figures may need to be scaled prior to drawing.

```

1 class Point {
2   int x, y;
3   int s;
4
5   public Point(int xi, int yi)
6     { x=xi; y=yi; s=1; }
7   public int getX() { return (x*s); }
8   public int getY() { return (y*s); }
9
10  public void move(int nx, int ny)
11    { x=nx; y=ny; }
12 }
13
14 aspect adjustScale {
15   pointcut m(Point p):
16     execution(void Point.move(int,int))
17     && target( p );
18
19   after(Point p) : m(p) {
20     if ((p.x < 5) && (p.y < 5)) { p.s=10; }
21   }
22 }

```

Figure 1. Point Class and Aspect

That is, indeed, precisely what the `adjustScale` aspect does. The *pointcut* `m()` corresponds to an execution of the `move()` method. The *after* advice specified states that if the point `p` is sufficiently close to the origin, then the scale factor is set equal to ten⁴. Thus, if we consider just the class `Point`, we see that the scale factor will remain at 1; and the values returned by `getX()` and `getY()` will be equal to the actual `x` and `y` coordinates. But this behavior of the base-code is modified as a result of the aspect so that, in those situations where the point in question is “close” to the origin, these methods return a value that is ten times the actual `x/y`-coordinate.

When reasoning about this example, the questions we want to address are as follows. How do we reason about base-code, i.e., the behaviors of the methods `getX()`, `getY()`, and `move()` of the `Point` class when considering the class by itself so that the reasoning remains valid when the effect of the `adjustScale` is also considered? In particular, what *rely()* condition that will be applicable to any aspect that may act on the methods of this class should we assume when doing this reasoning? Second, how do we show that the behavior of the advice defined in the `adjustScale` aspect is consistent with the *rely()* condition imposed by the `Point` class? Third, does the correct functioning of the advice require us to impose any conditions—the *guar()* clause—on the behaviors of the methods of `Point` and, if so, how do we specify those conditions; and how do we check that the actual behavior of the `Point` class, in fact satisfies the *guar()* clause? And, finally, how do we arrive at the *resulting* behavior that the combined system (of `Point` class and `adjustScale` aspect) will exhibit?

Consider first the *rely()* clause. This example is so simple that what we should include in the *rely()* clause is almost immediately clear: Any advice that may be applied to `Point` should be such that it doesn’t modify the value of `x` or `y` since the only time when a point’s coordinates change should be when we apply the `move()` operation on the point. Thus the *rely()* clause here may

be written as:

$$rely(\sigma, \sigma') \equiv [(\sigma.x = \sigma'.x) \wedge (\sigma.y = \sigma'.y)] \quad (1)$$

But the situation could be more complex. It may be that the condition that needs to be imposed on the advice depends on the particular type of advice in question. For example, if the class in question has two methods, `m1()` and `m2()`, the conditions that might have to be imposed on an advice that applies to `m1()` may have to be more stringent than those that have to be imposed on an advice that applies to `m2()`. Indeed, different conditions may have to be imposed on pieces of advice that might act at different joinpoints in the *same* method. The most natural way of expressing these multiple *rely()* clauses would be to use the *pointcut* notation to specify which particular joinpoints a given *rely()* clause corresponds to⁵. The default is that a *rely()* clause such as (1) applies to all joinpoints in all methods of the class.

How do we verify that the *rely()* conditions required by the class are indeed satisfied? For each item of advice in the aspect that applies to this class, we have to consider the behavior of the advice and show that this behavior is such that for each initial state in which we might start execution of the advice, the final state that we will reach when the advice finishes, and the starting initial state will, together satisfy the corresponding *rely()* condition. There is a potential problem here. When reasoning about the behavior of the class, we may have assumed too *strong* a *rely()* condition for one or more joinpoints. For example, if we had not considered the possibility of an aspect that might change the value of the scale factor `s`, we might have added another clause to the definition in (1) requiring that $(\sigma.s = \sigma'.s)$. The `adjustScale` aspect will, of course, not satisfy this *rely()* clause. Therefore, if we had indeed assumed this *rely()* clause when reasoning about the behavior of the `Point` class, we would be forced to go back and reexamine that reasoning to see if this clause was *really* necessary. This is not a fault of the reasoning approach; rather, it says that when reasoning about a class, we must be sure not to impose stronger requirements than necessary on aspects that might be developed later; otherwise, we might be forced to redo the task of reasoning about the class.

Let us next consider the *guar()* clause. There is one key difference between the kind of *guarantee* clause used in reasoning about concurrent program behavior [16] and the kind of guarantee clause needed here. In the case of concurrent programs, the two processes act *symmetrically* with respect to each other. That is, portions of each are interleaved with portions of the other. In our case though, once an advice starts execution, it decides when—or even whether, in the case of *around* advice—to give control back to the base-code. There is no possibility that the base-code can somehow intercept the advice and resume its own execution. Therefore, the *guar()* clauses that items of advice need are very much like *pre-conditions* of methods. That is, they are simply assertions over the state that exists at the time that the advice starts execution. For our example, the *guar()* clause is particularly simple. The advice does not really depend on any particular condition being satisfied by the state when it starts. Hence, this clause is simply the assertion *true*.

In general, of course, the *guar()* will not be so simple. Moreover, it may also depend on the particular advice in the same way that the pre-condition of a method of a class may well vary from method to method. There is, however, an important difference. When deciding the pre-condition of a method, we consider the internal details of the method and decide what conditions need to hold in order for it to behave in the desired manner. Here, when writing down the *rely()* conditions, we do not yet have the aspect available. Instead, when reasoning about the class in question we determine

⁴ It would be more interesting to use a scale factor that is a floating point value that might be set to be *less than* 1 to handle situations where the point might be *far* from the origin.

⁵ Of course, when reasoning about a given class, we will not be concerned with *other* classes. Hence, in specifying the *rely()* clauses, we will never have to use a wildcard that ranges over multiple classes in the system.

what assertions the various methods of the class can guarantee will necessarily be satisfied at various possible joinpoints. It is entirely possible that in doing so, we specify a *rely()* that is not sufficiently strong for a given advice to function properly. In that case, we will be obliged to re-reason about the class to see if a stronger set of *rely()* conditions can be established. Of course, in the current example, the advice does not depend on the class state satisfying any particular condition so we do not have to face this difficulty.

There is one important point that we have not considered until now and that we need to consider in order to arrive at the total behavior of the combined system consisting the `Point` class and the `adjustScale` aspect. When reasoning about any of the methods of `Point`, we do not really have any information about what the value of `s` would be since (1) allows any aspect that executes during the execution of any of these methods to change the value of `s`. Now the combined system will, of course, exhibit a behavior in which the value of `s` will be either 1 or 10 depending on whether or not the values of both `x` and `y` are less than 5.

To arrive at this, we need to borrow another idea that has been used in dealing with concurrent systems, this time one that allows us to handle communicating processes [9]. In these systems, a *history* variable is used to record information about all the interactions between the different processes of the system. Then, when the specifications of the different processes are combined, the information recorded in the different histories are merged and required to be *consistent* with each other. This process provides the additional information needed to establish the behavior of the combined system.

In our case, there are, of course, no communications between processes or even any processes. Nevertheless, there is a notion of history, this being a record of the transfers of control between the base-code and the various items of advice in an aspect, and back. The history should also record the states that existed during these transfers. Consider the behavior of `Point.move()`. We noted earlier, since *rely()* clause in (1) does not impose any restrictions on what any applicable advice may do to the value of `s`, that when `move()` finishes, we cannot state anything about this value. That is not entirely accurate. We know from the body of `move()`, for example, that *it* does not change the value of `s`. This means that if the value of `s` changes during the execution of `move()`, it must be because of the actions of the advices that must have executed during that period. Hence the post-condition of `move()` will state that not only will the values `x` and `y` be equal to the values for the corresponding parameters received in the call to this method, the final value of `s` will be equal to whatever it was when the final advice⁶ to execute during the execution of `move()` – which may be, and indeed is in our example, an *after* advice–finished execution. When we reason about `adjustScale`, we can easily establish that only one advice, the *after* advice specified in the aspect, is applicable to the execution of `move()` and that this will either set `s` to 10 or leave it unchanged depending on whether or not both `x` and `y` are less than 5 when the body of `move()` finished. Then when we combine this with the behavior of `Point.move()`, in particular combine it with the clause that says that the final value of `s` will be whatever is left in it by the last advice that executes, we can conclude that `s` in fact, will be 10 or be whatever it was at the start of the method depending on the values of `x` and `y`.

⁶ By looking at the `adjustScale` aspect, we can see that only one advice, i.e., the specified *after* advice, will be executed immediately following the completion of the body of `move()`. But we do not have this information when reasoning about the behavior of the `Point` class since the aspect may not even have been created at that point. This is why the post-condition of `move()` can only assert that the final value of `s` when it returns to the caller will be what we have stated.

Interestingly, as we were going through the reasoning task that we have summarized above, we discovered that the system does not behave in the manner we had assumed it behaved! Specifically, we had expected that, for the combined system, for any point `p`, the value of `p.s` would be 10 if `p.x` and `p.y` were both less than 5 and 1 otherwise. While the first part of that expectation was indeed satisfied but the second part was *not*! The reason is that while the *after* advice does set the value of `s` to 10 if the `x` and `y` values are both less than 5, it does not *reset* `s` back to 1 if this condition is not satisfied! Hence, once the value of `s` becomes 10, it never goes back to 1. This was certainly not the behavior we had intended to implement when we designed the class and the aspect in Fig. 1 but this is what we discovered was the actual behavior during the reasoning process that we outlined above. Thus even apparently simple examples such as this might exhibit surprising behavior; thus reasoning techniques such as the one we have outlined above are essential.

3. Related Work

There have been several key efforts and approaches proposed to aid in reasoning about AOP. Clifton and Leavens [2] propose a categorization of aspects that help make AOP more compatible with traditional reasoning approaches. Aspects are separated into two categories, *Observers* and *Assistants*. *Observers* are aspects that are allowed to make certain kinds of changes to the base system, specifically, ones that do not alter the base module’s effective specification. *Observers* can change the base system state so as long as the changes lead to a new state that still satisfies the post-condition of the advised method. The claim is that these aspects are safe to ignore in reasoning about the base-code as they do not influence its specification. Conversely, aspects that can potential alter the effective specification of the base-code are categorized as *Assistants*. Aspects included in this category can not be safely ignored during reasoning as they are allowed to make arbitrary changes to the state of the base-code, thereby potential altering the effective specification of the module. In order to facilitate local reasoning in the traditional sense, the assistant can only influence the base-code if a special assistance clause is specified above the class declaration. This clause proclaims that the specified aspect is allowed to potential alter the base-code’s state and coincidentally alter its effective specification. This work differs from ours in a fundamental way. For instance, the base-code, in *accepting* assistance from an *assistant* aspect, must have *a priori* knowledge of which *assistants* may be applicable to it. The approach presented in this paper, however, assumes no such knowledge thus making it more flexible under evolution. The main reason for this difference is that our approach does not require the author of the base-code to explicitly reference any particular aspect. As such, using our approach, the author has the ability to make decision about the base code in isolation of advice and validate those decisions upon composition of the two subsystems (base-code and aspect). Furthermore, this allows for complex, base-code state altering aspects to be interchanged in a *plug-n-play* fashion while still allowing authors to reason about possible aspect influence.

Kiczales and Mezini [11] define a notion of *Aspect-Aware* interfaces claiming that traditional reasoning techniques are not applicable to programs (AOP and non-AOP alike) that contain cross-cutting concerns. They claim that interfaces must then be augmented with additional information that accounts for the presence of cross-cutting concerns. Namely, this information added to base-code module interfaces states knowledge regarding the aspects that may influence the corresponding module. This information is required by their claim that interfaces are not fully known until the complete *deployment* configuration of a system is revealed. Again, as with Clifton and Leavens’ technique, using this approach requires that, before any reasoning can be done about the base-code,

knowledge of the applicable aspects must be involved. In other words, Kiczales and Mezini's reasoning approach assumes that the aspects are present at the time of reasoning. Our approach takes a different view by allowing the developer to reason about influence that aspects may have on the base-code despite these aspects currently existing.

Aldrich [1] proposes an approach that allows for reasoning about the effect of base-code on advice. The approach is mainly interested in revealing when changes to the base-code produce a program that no longer triggers the expected joinpoints. This issue is also related to the pointcut coupling problems depicted in [8]. To combat the problem, Aldrich's approach calls for interfaces between the client (aspect) code and the module (base) code that essentially acts as a contract between the two subsystems. Then, the aspect's intended behavior becomes robust to *semantics-preserving* changes of the base-code, thus not requiring rebuilding previous reasoning efforts under these situations. Our approach has a similar flavor in that re-reasoning efforts under certain situations are also limited, namely, when the *rely* clause assertion remains satisfied following changes to the program. However, we do not make a general distinction between the types of changes that allow for this efficiency. Rather, these types of changes are on a per-joinpoint basis. Moreover, our approach allows isolated reasoning in regards to the aspect as it is capable of utilizing *guar()* assertions as stated earlier. Therefore, using our method, advice authors may be able to come to conclusions about the state prior to execution of their advice even in the absence of the base-code. We leave the notion of utilizing a *guar* clause to preserve joinpoint triggering as future work.

Krishnamurthi et. al. [12] introduce the ability to verify aspect and base code independently through the use of interfaces. Although verification is a related concept of reasoning, our approach is more language-centric and is meant to be used by program developers as opposed to automatic verification mechanisms. Also, our work differs in that our assertions are not over temporal properties of the system. It suffices to say, however, that our approach may be enhanced by such verification tools which may be a focus of future work.

Dantas and Walker [3] define a category of "Harmless Advice" that does not affect the reasoning of base-line code and who's behavior can be enforced at compile-time. Such highly restricted advice is limited to producing the same result the unadvised base-code would have produced normally. In addition, I/O manipulation is allowed by the advice. Although restrictive, advice of this kind has been shown to prove useful especially in the domain of security. Our approach also leverages the notion of restricting the capabilities and behavior of advice, however, our approach is much more flexible and adaptable in this respect. Specifically, the base-code is allowed to explicitly state on a per-joinpoint basis as to the types of advice behavior it considers "harmless" by means of its associated *rely()* assertion.

Devereux [5] exploits the similarities between aspect-oriented programs and concurrent programs. The approach translates an aspect-oriented program into a concurrent program. The process continues to transform that program into an equivalent, low-level, concurrent program in an alternating-time logic formalism. The idea is that once this is done, we can proceed to reason about the concurrent program using the *rely-guarantee* approach. Such a transformation, however, would seem to make the task of reasoning about the original program rather complex since any change to either subsystem (base or aspect) would require re-establishing previous reasoning efforts. Additionally, *rely/guarantee* clauses in reasoning about a concurrent program are normally developed using knowledge of the behavior of all the processes. In other words, in this approach, when reasoning either about the base code

or about the aspect, one uses the knowledge of the behavior of the other. Our goal has been to avoid such dependence since the aspect may not even have been developed yet when we wish to reason about the base code; or the base code might be combined with different aspects in different systems. We believe that our proposed adaptation of *rely-guarantee* is tailored more to the needs to aspect-oriented programs specifically, and in particular, in the sense of evolution of such systems.

4. Future Work and Conclusion

The *rely-guarantee* approach for reasoning about aspect-oriented programs proposed in this paper is a departure from traditional mechanisms. By using our approach, the base-code is potentially cognitive to the influence of present and/or future advice that may be applied to it. Therefore, the base-code may be not be completely oblivious to aspects. This total obliviousness property was once viewed as crucial to AOP [6], however, recent work [1, 12, 11, 2, 15] seems to admit otherwise. So-called partial obliviousness, instead, could provide many benefits in reasoning about AOP that outweigh any trade-offs. Such benefits, as exploited by the *rely-guarantee* approach presented in this paper, may be even more apparent as boundaries are crossed between software artifact. Furthermore, the flexibility and adaptability possible with our proposal may allow for ease of transition into crossing over artifact boundaries, perhaps serving a model for future approaches.

In summary, the *rely-guarantee* approach to reasoning about AOP proposed in this paper allows for reasoning that is tailored to the unique, evolutionary flavor of AOP that has made the paradigm popular. It enables a developer to reason about base-code subject to advice independent of whether or not advice currently exists at the time of reasoning. As future work, we plan to develop axioms and rules for our approach and show of soundness and completeness in respect to these rules. We also plan to employ such mechanisms as behavioral sub-typing to allow for proper verification of more complex pointcuts that query over a class hierarchy. Furthermore, we plan to consider more complex situations where multiple advice can be applied to a single joinpoint, as well as introductions of variables into the base-code through static cross-cutting mechanisms. We foresee many interesting issues and debates as to how best to incorporate and express a *rely* clause in the context of existing AO languages. The issues of what kinds of conditions might be needed to be included in *rely* clauses corresponding to various kinds of advices and joinpoints will need to be addressed. Lastly, we plan to explore how tool support may be leveraged to further enhance our proposed reasoning technique.

References

- [1] J. Aldrich. Open modules: Modular reasoning about advice. In *Proc. of ECOOP*, pages 144–168. Springer, 2005.
- [2] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. *Proc. FOAL Workshop.*, 2002.
- [3] D. Dantas and D. Walker. Harmless advice. In *POPL '06*, pages 383–396. ACM, 2006.
- [4] 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.*, pages 158–162. Springer, 2004.
- [5] B. Devereux. Compositional reasoning about aspects using alternating-time logic. *Proc. FOAL Workshop.*, 2003.
- [6] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, MN, October 2000.
- [7] E. Gamma. JHotdraw, 2004. Available for download from <http://sourceforge.net/projects/jhotdraw>.

- [8] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, 2003.
- [9] C.A.R. Hoare. Communicating sequential processes. *Comm. ACM*, 21:666–677, 1978.
- [10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Longtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, 1997.
- [11] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proc. of ICSE '05*, pages 49–58. ACM, 2005.
- [12] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *Proc. of the 12th FSE*, pages 137–146. ACM, 2004.
- [13] R. Laddad. *AspectJ in action*. Manning, 2003.
- [14] M. Lippert and C. Lopes. A study on exception detection and handling using AOP. In *ICSE*, pages 418–427. ACM, 2002.
- [15] A. Rashid and A. Moreira. Domain models are NOT aspect free. In *MoDELS*, pages 155–169, 2006.
- [16] Q. Xu, W. de Roever, and J. He. Rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.