Reasoning about the Behavior of Aspect-Oriented Programs

Neelam Soundarajan[1],   Raffi Khatchadourian[1],   Johan Dovland[2]

[1] Computer Sc. & Engineering
Ohio State University
Columbus, OH 43210
{neelam, khatchad}@cse.ohio-state.edu

[2] Dept. of Informatics
University of Oslo
PO Box 1080 Blindern
N-0316 Oslo, Norway
johand@ifi.uio.no

Abstract

Aspect-oriented programming (AOP) has become increasingly popular over the last few years. At the same time, a number of authors have raised serious questions about the problems involved in reasoning about the behavior of aspect-oriented programs. In this paper, we present a rely-guarantee-*approach to such reasoning. The rely-guarantee approach has proven useful 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.*

1   Introduction

Kiczales *et al.* proposed *aspect-oriented programming* (AOP) [10] as an approach to enable modular implementation of *crosscutting concerns*. Various authors [4, 13, 14] have shown how AOP may be used to write code for such crosscutting 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 difficulties for reasoning about the behavior of AO programs. The problem is that, since the addition of the aspect changes 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 interleaved execution of various pieces of advice with the base-code.

The problem of reasoning about AO programs has some resemblance to reasoning about *concurrent* programs. Consider a concurrent program with two parallel processes that share some variables that either of them may read or 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. 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. This is rather 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.

The *rely-guarantee* approach [7, 15] addresses the interference problem in parallel 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 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 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. 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 need to check 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 *parallel composition* rule requires us to check, using $guar_1()$ and $guar_2()$, that the *rely* clauses are satisfied.

The key observation underlying our proposed approach to reasoning about AO 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 the aspect is applicable, the statement at that point will execute, then control will transfer to the advice, the advice will execute, possibly changing the values of some of the instance variables of C, and then control will return to $m()$ which will then continue execution[1]. To handle this, we will introduce a *rely*() condition that will specify constraints on the state changes that the advice may perform.

While this is analogous to the situation in the case of the two parallel processes, there is one key difference. While in the case of parallel programs, the two processes are typically designed hand-in-hand, in an AO program, the base-code is often written without any particular aspect in mind. By adapting the rely-guarantee approach, we make it possible for the base-code provider to reason about the base-code independent of particular aspects.

The base-code provider specifies suitable rely conditions that must be respected by various possible aspects that may be introduced to ensure that the base-code reasoning remains valid even after the addition of the aspects.

In other words, when reasoning about $m()$, we recognize that advice contained in some aspect may apply to $m()$ at particular points during its execution. If this advice were to make arbitrary changes in the relevant state, i.e., the instance variables of C that $m()$ deals with, clearly the behavior of $m()$ will be 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, *rely*() 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 this advice.

One extreme possibility would be, when reasoning about $m()$, to define *rely*() as:

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

where σ is the state when the aspect gets control and σ' the state when control returns from the aspect. This 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] is essentially of this form. Of course, we do not have to use such a restrictive form of *rely*(); and indeed, if we want be able to exploit the full power of aspects, we should make the *rely*() clause as liberal as possible rather than imposing such a tight constraint.

In addition to the *rely*() clause we will also have a *guar*() clause. This clause will be of a different kind from that used in concurrent programs. The reason is that while there is a clear symmetry between two processes of a concurrent program, such a symmetry does not exist between the base-code and a piece of advice in an AO program. More precisely, while the base-code is "intercepted" when control in the base-code reaches particular joinpoints and the corresponding advice is executed and then the base-code resumes execution, the advice does not get intercepted in a similar manner by the

---

[1]This description assumes the advice in question is a *after*-advice. In the interest of keeping the presentation simple, for much of this paper, we will only consider after-advice. *Before*-advice can be treated in a completely analogous manner as after-advice. But *around*-advice poses some difficulties. We will return to this later in the paper.

base-code[2]. Hence, in our approach, a *guar*() clause is an assertion over a single state that must be satisfied by the state when control transfers to any advice that may be applicable at particular joinpoints.

There is another difference between our situation and that in concurrent programs. In the case of concurrent programs, the rely-guarantee approach is used to ensure that the actions of an individual process do not invalidate the reasoning about another process. In our case, we not only have to ensure that the advice does not invalidate the reasoning that has been done about the base-code, we also have to arrive at the *overall* behavior that the combined system consisting of the aspect(s) and the class(es) together exhibit. In Section 2 we will see how our approach allows this. A preliminary version of our approach was presented in [8].

The rest of the paper is organized as follows. In the next section, we consider the details of our approach to specifying and verifying AO programs. The focus is on specification rather than verification. Hence we do not define formal proof rules but consider what would be required in order to verify our specifications. In Section 3, we apply our approach to a simple example. In Section 4 we briefly describe some items of related work. In Section 5, we summarize our approach and consider possible directions for future work.

2   Specifying and Verifying AO Programs

For concreteness, we will use the syntax and terminology of *AspectJ* [9] in our discussion although the main ideas underlying our approach do not depend on the details of AspectJ. The three key features of an AOP language (in addition, of course, to the features of the underlying base language) are *joinpoints*, *pointcuts* and *advice*. Joinpoints are the specific points in the execution of base-code where advice may be applied. In order to allow flexibility, AspectJ defines a number of different kinds of joinpoints. A *set*-joinpoint corresponds to a point where a particular variable of a class is *set*, i.e., assigned a value; a *get*-joinpoint corresponds to a point where a particular variable of a class is *read*. A *method-call*-joinpoint corresponds to a point where a particular method is called;

a *method-execution*-joinpoint corresponds not to the call but rather to the actual body of the method; etc.

Our approach is applicable to all types of joinpoints in fairly similar ways. From the point of view of an AspectJ programmer, perhaps the most important kinds of joinpoints, and the ones most widely used, are method-call and execution. However, our approach is most conveniently presented as it applies to set-joinpoints, hence we mainly use these in our discussion. The example in the next section is also written in terms of set-joinpoints but it can be easily rewritten in terms of call-joinpoints.

Pointcuts allow us to specify *groups* of joinpoints. The simplest kind exhaustively lists all the joinpoints in the group. Most AOP languages provide more convenient ways to specify the group, including the use of various *wildcard*-constructs. Once a pointcut has been defined, we can define a piece of advice to apply to the pointcut; i.e., this advice will apply to *each* of the joinpoints in the particular pointcut[3]. From a programming point of view, the ability to define pointcuts in this manner and to associate advice with the pointcut rather than with the individual joinpoints is extremely valuable. Indeed, this ability, usually referred to as *quantification*, is considered one of the key features of AOP [5].

When reasoning about a method $m()$ of a class C, a single *rely*() clause may not be appropriate since the correct functioning of $m()$ may depending on having the advices that may apply at different points in the body of $m()$ satisfying different constraints. If we had to have a single *rely*() clause for $m()$, we would be forced to combine, i.e., conjunct together, all of these constraints into that clause and it would be unnecessarily restrictive. The situation with the *guar*() clause, which is the condition that the base-code ensures will be satisfied when control reaches the advice, is similar. If we were forced to associate a single *guar*() clause with $m()$, we would have to define this clause to be the disjunction of the assertions that will hold at the various joinpoints in $m()$; and this may not provide sufficient information to the different advices that may apply at these different joinpoints.

One possible solution to this would be to specify a separate *rely*() and *guar*() clause corresponding to each joinpoint in $m()$. But this may result in a lot of duplication if many of these joinpoints are *equivalent* from the point of view of $m()$ and the conditions that it ex-

---

[2]Many AOP languages enable the definition of advice $A2$ that is applicable to another advice $A1$. In other words when control reaches certain joinpoints in $A1$, it is intercepted and $A2$ will execute; and once $A2$ finishes, control will (typically) return to $A1$. If we wanted to deal with such situations, we would of course need to define suitable *rely*() clauses for $A1$; but in this paper we will not consider such possibilities in any detail. Nor will we consider the possibility of *multiple* advices being applicable, in some appropriate order determined by their "precedence", at the same joinpoint of a method $m()$.

[3]AspectJ allows the programmer to define a pointcut in such a way that particular joinpoint will be included only if the *control-flow* that brought control to that joinpoint satisfies certain specified conditions, for example, that this control-flow is not part of a given method's invocation. We will not consider such constructs in the paper. AspectJ gurus will probably notice that we take some minor liberties with the syntax of the language.

pects will hold at these points. The notion of a *specification pointcut* provides a suitable solution. A specification pointcut (s-pointcut), like a normal pointcut, defines a group of joinpoints. But, unlike a normal pointcut, the group of joinpoints in a given s-pointcut must all be from a particular method of a given class. Since we reason about one method at a time, there is nothing to be gained by defining s-pointcuts that would include joinpoints from different methods of the class (or, worse, multiple methods from multiple classes). The precise syntax for defining s-pointcuts is for future work; in this paper, we assume that s-pointcuts will be defined by explicitly listing all of the joinpoints in the particular group. In order to permit such listing, we also assume that, as part of the reasoning annotation of the method $m()$, unique labels are associated with each pointcut.

Thus the specification of a method $m()$ of a class C would consist of a pre-condition *pre*(), a post-condition *post*(), a number of s-pointcuts and, for each s-pointcut, a *rely*() clause and a *guar*() clause. The meaning of this specification is that if *pre*() is satisfied when $m()$ is invoked and each advice that is defined in any aspect and is applicable to any joinpoint in $m()$ satisfies the requirements specified in the *rely*() clause associated with the s-pointcut that includes that joinpoint, then, when $m()$ finishes, *post*() will be satisfied; and when control reaches any advice applicable to any joinpoint in $m()$, the values of the instance variables of C will be such that the *guar*() clause associated with the s-pointcut that includes that joinpoint will be satisfied.

What if a particular joinpoint in $m()$ is not included in any of the s-pointcuts defined in the specification of $m()$? One possible answer would be that default the *rely*() clause for such a joinpoint should be the highly restrictive one we considered in Section 1:

$$rely(\sigma, \sigma') \equiv (\sigma = \sigma') \tag{1}$$

where σ is the state of C, i.e., the set of values of all the member variables of C. This, as we noted earlier, forbids the advice from making any changes to the values of these variables. But note that if the aspect that the advice is a part of uses the *introduction* mechanism to introduce *new* variables into C, then the advice will be able to manipulate those variables in appropriate ways and thereby enrich $m()$'s behavior. That is, the *rely*() clause, whether explictly specified or the default one, imposes conditions only on the variables defined directly in the class in the base-code; it does not impose any conditions on any variables that the aspect may introduce into the class. This is natural because the functioning of $m()$ cannot depend on or be affected in any way by the values of any of these *introduced* variables. The default *guar*() clause could sim-

ply be *true* since, in this situation, we cannot guarantee anything about the values of these variables when control reaches the advice applicable at this joinpoint.

Let us turn to the task of establishing the *richer* behavior that the advices are intended to produce. In order to do this, the assertions in $m()$ including, in particular, the *post-condition*, need to provide a more detailed characterization of the behavior of $m()$. In effect, the standard post-condition of $m()$ essentially tells us the conditions that will be satisfied by the state when $m()$ finishes. What we need in order to reason about the enriched behavior resulting from the action of the aspects is a way to specify precisely how the behavior of $m()$ would be enriched as a result of the actions of the various advices that may be applied at various joinpoints in the body of $m()$.

Since there are any number of different advices that the aspect designer might develop, we cannot, of course, try to consider each one of them separately and specify how it would enrich $m()$'s behavior. Rather, we need an approach using *traces* that record information about the joinpoints that are reached during the execution of $m()$, the conditions satisfied by the state when control in $m()$ reached those points and, how the further behavior of $m()$ depends on the changes any advices acting at any of these joinpoints may make. These changes will, of course, be required to satisfy the requirements of the appropriate *rely*() clause but the point is that the use of the traces will enable us, in a sense, to factor into the behavior of $m()$ the changes that the advice may produce. This will be achieved in the following manner. Each trace element will represent the effect of a single joinpoint. Consider the trace element corresponding to the joinpoint labeled L. This element will be of the form (σ, σ'), these being the states just prior to control transfering to the advice acting at this point and immediately after control returns from the advice. Thus these states will (be required to) satisfy the appropriate *rely* clause. Further, the assertions beyond that point in $m()$ will be in terms of the state σ' rather than in terms of σ. This will allow us to suitably combine the effect of the advice with the behavior provided by the base-code to arrive at the enriched behavior of $m()$. And in those cases where no advice is applied at this joinpoint, this process will simply result in σ being taken to be equal to σ'. As mentioned earlier, the formal proof rules that correspond to these reasoning tasks is part of our planned future work.

3 Case Study

Fig. 1 depicts the base-code of our simple example, consisting of a single class, Point, instances of which represent points in an *xy*-plane. The constructor initial-

```
1  class Point {
2    int x, y;
3
4    public Point(int xi, int yi)
5          { L1: x=xi; L2: y=yi; }
6    public int getX() { return (x); }
7    public int getY() { return (y); }
8
9    public void move(int nx, int ny)
10          { L1: x=nx; L2: y=ny; }
11 }
```

Fig. 1. Point Class and Aspect

izes the point variables. `move()` changes the coordinates to the specified values. `getX()` and `getY()` return the current coordinate values. We have also labeled the various set joinpoints for use in the reasoning activity.

Suppose this class is now used as part of a graphics system that includes a number of other components including a *display*. The graphics designers may decide after considering various parts of their system that when any of the graphics items moves outside the bounds of the display monitor, they will use a *wrap-around* technique to bring it within bounds. This can be achieved by defining the aspect shown in Fig. 2. The pointcut corre-

```
1  aspect WrapAround {
2    pointcut m(Point p):
3      (set(int Point.x) && target(p)
4         || set(int Point.y) && target(p))
5           && !within(WrapAround);
6
7    after(Point p) : m(p) {
8        Point b = Display.getNECorner();
9        p.x %= b.x; p.y %= b.y; }
10   }
11 }
```

Fig. 2. Point Class and Aspect

sponds to all the joinpoints where a new value may be assigned to either x or y. The after-advice then adjusts the coordinates –if they are out of bounds– to be within the required bounds by acquiring the coordinates of the "north east" point of the display and setting the coordinates of the point p appropriately[4].

Let us now consider the specification of the base-code, in particular of the `move()` method that appears in (2). The "$*$" notation in the *rely/guar* clauses means that these clauses apply to all the labeled points in the method.

$$rely.L^* \equiv true \qquad\qquad\qquad (2)$$

$$guar.L^* \equiv true$$

$$post \equiv [\,(\texttt{L1.x==nx}) \,\wedge\, (\texttt{L2.x==L1.x'})$$
$$\wedge(\texttt{x==L2.x'})\wedge(\texttt{L2.y==ny})\wedge(\texttt{y==L2.y'})\,]$$

---

[4]Line (7) had to be included in order to ensure that the advice, given that it contains assignments to `p.x` and `p.y` and these match the joinpoints defined by the pointcut (lines 3 and 4), does not advice itself!

The post-condition states the following: The value of x when control reaches L1[5] is the same as the value of the input argument nx; that the value of x when control reaches the L2 joinpoint is the same as the value it had at the *end* of the L1 joinpoint, i.e., when control returned from the advice (if any) that was applied at that joinpoint; that the final value of x when `move()` finishes is the same as the value it had at the end of the L2 joinpoint. The last two clauses similarly specify y.

The information in (2) can now be combined with the behavior of the advice in Fig. 2 to arrive at the behavior of the overall system. First, we must reason about the advice, using the information in the *guar* clauses and check that the *rely*() clause requirements are met. This is straightforward for this case. Next we combine the behavior of the advice code with the information in the *post*-condition of `move()` to arrive at the following net behavior of this method:

$$post \equiv [\,(\texttt{x==(nx\%d.x)}) \,\wedge\, (\texttt{y==(nx\%d.y)})\,]$$

where we have used `d.x` and `d.y` to denote the coordinates of the northeast corner of the display.

To summarize, when reasoning about a base-code method, we decide on appropriate *rely*() clauses for the various joinpoints in the method. Then we arrive at the post-condition of the method; and, in this process, we allow for the fact that the state might change, as a result of advice being applied at any of the joinpoints. This specification can then be combined with information about the behavior of any advice that may be developed –provided the *rely*() clause requirements are met– to arrive at the net resulting behavior of the method. In future work, we will explore ways to simplify the base-code specifications as well as the work involved in combining those specifications with the advice specifications.

4 Related Work

Kiczales and Mezini [11] argue that, in the presence of aspects, we cannot expect to work with the standard interfaces provided by a class's methods and their behaviors. Instead, we must define a more detailed interface for the class in such a way that the interface includes also the various joinpoints at which various advices defined in the aspects are applicable. We can then consider the properties associated with the various items in this extended interface. Aldrich [1] introduces the notion of *open modules*. The idea is that the base-code designer should explicitly identify a specific set of points in the base-code at which advices will be allowed to apply; these are the

---

[5]Recall that we only consider *after*-advice; as a result the joinpoint is actually immediately *after* the assignment x=nx is performed.

points at which the base-code is *open* to accepting advice. Any attempt to define an aspect that contains an advice applicable at a point that is not included in this set will result in a (compile time) error. Aldrich defines a language around this concept and defines its operational semantics. Neither of these papers considers techniques for behavioral specifications of the classes nor how the enriched behavior of the classes resulting from the application of the aspects may be arrived at.

Clifton and Leavens [2] introduce the notion of *observers* and *assistants* as two kinds of after-advice based on their effect. An observer only "observes" and does not make changes that might violate any assertions in the base-code. An assistant may make arbitrary changes. Clifton and Leavens consider how the effect of multiple pieces of assistant-advice applied when a base-code method completes execution may be combined with the post-condition of the method to arrive at the net behavior.

Traces have been widely used in reasoning about *distributed* systems such as *CSP* [6]. There, traces are used to record information about the communications between the processes of the system. Our traces to record information about the situations that exist at various joinpoints in $m()$ since these are the points where advices might intercept the base-code and provide for enriched behavior.

5 Discussion

In this paper, we have proposed a rely-guarantee-based technique for reasoning about AOP programs in modular fashion. We demonstrated the technique on a simple example. Although in our presentation we focused on field-access joinpoints, the approach applies to other kinds of joinpoints. On the other hand, extending the approach to *after* advice will pose some serious challenges. The main problem is that the *after* advice may not contain a *proceed* call or may contain *multiple* such calls. In both cases, the behavior of the combined system will be substantially different from that of the base-code. Apart from dealing with *after*-advice, we intend to develop, in our future work, ways to deal with multiple advices that may apply at a single joinpoint as well as advice that may advise other advice. Another important task for future work is the development of formal proof rules for establishing the behavior of AOP programs.

Acknowledgments

References

[1] J Aldrich. Open modules. 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, 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] T. Elrad, R. Filman, and A. Bader, editors. *CACM Special Issue on Aspect Oriented Programming*. ACM, Oct. 2001.

[6] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[7] C B Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Prog. Lang. Sys.*, 5:596–619, 1983.

[8] R Khatchadourian and N Soundarajan. Rely-guarantee approach to reasoning about aop programs. In *SPLAT Workshop at AOSD*, 2007.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Overview of aspectj. In *Proc. 15th ECOOP*, pages 327–353. Springer-Verlag, 2001.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Proc. of ECOOP'97*. Springer.

[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] Q Xu, W de Roever, and J He. Rely-guarantee method for verifying concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.