Covariance, Contravariance, and Synchronization Constraints

Neelam Soundarajan

Computer and Information Science
The Ohio State University
Columbus, OH 43210
USA

e-mail: neelam@cis.ohio-state.edu

Abstract: Object Orientation and concurrent progamming are a natural match. Objects correspond to processes in a concurrent program; a message from one object to another invoking a method of the latter, corresponds naturally to (message passing) interaction between processes in a concurrent program. Despite this close correspondence, progress in developing an OO approach to concurrency has been limited. One important problem has been the so-called *inheritance anomaly* which is concerned with *how* and *how easily* synchronization constraints specified in a base class may be modified in a derived class. Our concern in the current paper is more basic. We address the question of *what* types of modifications in the synchronization constraints inherited from the base class ought to be permitted to be made in the derived class. We identify the modifications that are reasonable, especially when the problem is viewed from the point of view of a *client* of the class. We point out that when viewed from this perspective, some of the current proposals for expressing synchronization constraints allow precisely the wrong types of modifications to be made in the derived classes. We also analyze the relation between this question and the problems that inheritance encounters already in the sequential case, such as the covariance versus contravariance conflict.

1 Introduction

Object Orientation (OO) and concurrent programming are a natural match. Active, autonomous objects correspond naturally to processes in a concurrent program. Interactions between these objects in the form of messages, i.e., method invocations to carry out various tasks, correspond to interactions between the processes. Since individual objects do not have access to the internals of other objects, they can exist on separate machines with no shared memory. Despite this close correspondence, progress in using OO ideas in concurrent programs has been very slow.

Inheritance is one of the cornerstones of the Object Oriented approach. It not only allows us to create classes in an incremental manner from existing classes, but also, in conjunction with *polymorphism* and *dynamic dispatch* makes it possible to write very flexible client code that can manipulate objects that are instances of different but conceptually related classes in a uniform manner. Although some authors have criticized inheritance, others like Meyer [16] have presented persuasive arguments in its favor.

One of the main reasons for the slow progress in bringing OO ideas to concurrent programming is the *inheritance anomaly*. Essentially the problem is that code that is needed to ensure that appropriate synchronization constraints (such as not reading from an empty buffer, or not writing to a full buffer) is often interspersed with the code of the methods (such as *get* and *put*). Suppose, for instance, we use the idea of *behavior abstractions* of Kafura et al [11]. Then in the code of each method, such as *get* and *put*, we will find commands like *'become empty'*, *'become partial'*, etc. to denote the subsequent *behavior* of the buffer. In addition, outside the method bodies, the behavior abstractions *empty*, *partial*, etc. would be identified as corresponding to the respective sets $\{put\}$ or $\{put, get\}$ etc. to specify which operations may be invoked by the client when the buffer is *empty* or *partially full* etc.

Suppose now we want to define a derived class that provides another operation called *get2* that allows a client to extract two elements from the buffer at once. Then we will not only have to specify the code for *get2*, and introduce another behavior abstraction, call it *partial1,* to denote the state of the buffer when there is exactly one element in the buffer so that *get* may be invoked but not *get2*, but also have to rewrite the code of the *get* and *put* operations to make sure that the buffer *becomes partial1* rather than *partial* at the appropriate times. In other words, although the code for *performing* the *get* and *put* operations has not changed –indeed that is the reason we are trying to use inheritance and *reuse* these operations defined in the base class– we are obliged to rewrite them anyway in order to encode the new synchronization conditions needed in view of the new operation. This is the problem that was dubbed the *inheritance anomaly* by Matsuoka and Yonezawa [14] who perfomed a thorough analysis of this problem. A natural solution to the problem, and this has been suggested by several authors such as Frolund [7], would be to introduce notation that will allow us *separate* the synchronization constraints from the bodies of the operations. We will use one of these notations later in the paper. Although the problem of inheritance anomaly doesn't entirely go away with such a separation, it makes the problem much more manageable.

Our interest in the current paper is a somewhat different problem concerning synchronization issues and inheritance. We are interested not in *how* synchronization constraints ought to be modified in derived classes but *what* sorts of modifications are reasonable. As we will see in section 3, our analysis shows that while they go part way to solving the inheritance anomaly problem, approaches like that of [7] do not help with this question. Indeed they allow the *wrong type of changes* to be made in derived classes to synchronization constraints inherited from the base classes.

It turns out that the problem we analyze bears considerable similarity to the problem of covariance *versus* contravariance in sequential OOP. So in the next section we review that problem and the solutions that have been proposed. In the third section we consider synchronization issues and discuss the problem with synchronization constraints of the kind used by [7] and compare this problem with the covariance / contravariance question. We then propose some possible solutions to the problem and analyze the implications. We do not propose any new liguistic constructs that will ensure that synchronization constraints are only modified in ways we deem appropriate. We intend to analyze existing notations to see if one of them may be embellished to serve the purpose instead of introducing entirely new notation. In the final section we summarize the problem and our conclusions, and mention some questions that remain to be answered.

2 Covariance versus Contravariance

Consider the following (classic) example taken from Hursch [9]. Suppose we have a class called *Point* to represent points on a 2-d plane. The instance variables of the class are, of course, the x- and y-coordinates of the point. Suppose also that we have a function called *eq* defined as a member of this class. *eq* receives a second *Point* object as argument and returns *true* or *false* depending on whether or not the current *Point* and the second *Point* have the same x- and y-coordinates. In a C++ like notation, the function *eq* would be written as

```
Boolean eq( p:Point )
 { return ( ( x==p.x ) && ( y==p.y ) ) }
```

Suppose now we wish to define a new class *ColoredPt* to represent 2-d points that have color. The natural approach would be to define this class by inheriting from the *Point* class and redefine the *eq* function:

```
Boolean eq( p:ColoredPt )
 {return (( x==p.x ) && ( y==p.y ) && ( c==p.c)) }
```

where `c` is the new field in the derived class (to denote the color of the point object in question).

But this leads to a serious problem. Suppose in the client program we have the following code (in C++ we would have to declare `p1, p2` to be of type *pointers* to `Point` but that is a language detail we will ignore):

```
p1, p2:  Point;
cp:  ColoredPt;
p2 := new Point( ... );
cp := new ColoredPt( ... ); p1 := cp;
p1.eq( p2 ); // run-time error!
```

This is the problem introduced by the *covariance* of the `eq` function: In the derived class we have restricted the domain of the `p` argument of `eq` to `ColoredPt`. Because the (implicit) first argument `p1` in the call to `eq` is of type `ColoredPt`, the dynamic dispatch mechanism results in the `eq` function of that class being called, resulting in a run-time error since `p2` is of type `Point`, not `ColoredPt`.

It is because of such problems that authors like Cardelli [3] have required contravariance: when we override the `eq` function defined in the base class by a new defintion in the derived class, we are *not* allowed to restrict the domain of the (explicit) arguments of the function.

On the other hand, some authors, Meyer [OOSC] being the most prominent of these, argue for covariance. Their primary concern is with *naturalness* in the definition of the derived class. It seems more natural for the implementer of the `ColoredPt` class to define an `eq` function which receives a second argument that is also a `ColoredPt` rather than a `Point`. Indeed unless this were allowed, it would be impossible to write a function in `ColoredPt` that would override the `eq` function of `Point` *and* would take account of the `color` component of its second argument. While there may be some merit to this argument, if we use covariance we run the risk of encountering the run-time error 'message not understood' as in the example above. It will become the responsibility of the client program to avoid this problem. (Eiffel tries to provide some warnings on the basis of link-time analysis, when there is a possibility of such situations arising.)

A very general analysis of the contravariant versus covariance conflict is given by Castagna [4]. His conclusion essentially is that covariance is ok on arguments on which we do dynamic dispatch, but on others we must use contravariance. This, of course, disagrees with Meyer's approach. Thus in the `Point` example, following the analysis in [4], we would have to define the `eq` function in `ColoredPt` to receive a second argument of type `Point`, not `ColoredPt`, since no dynamic dispatching is being done on this argument. Indeed, for standard OO languages like C++, since dynamic dispatch is done only on the first argument, the analysis in [4] doesn't contribute anything new to what has been said in earlier papers requiring contravariance.

There is one possible way to stay with covariance without running the risk of run-time errors. Every derived class would have to provide a *constructor function* that, given an instance of the base class, constructs an instance of the derived class. In the `Point` example, the `ColoredPt` class would have to contain a function:

```
ColoredPt( Point p ) { ... }
```

which specifies how a `ColoredPt` can be constructed given a `Point p` (probably by setting the color component to some default value). Then if `eq` of the `ColoredPt` class were to receive a `Point` argument, we would first apply this function to construct a `ColoredPt` object and use that instead. This approach seems to work at least for simple examples but we won't pursue the idea further in this paper.

Another attempt at trying to retain covariance in a limited situation (but still avoiding run-time errors) is that of Dodani and Tsai[6]. Their idea was to allow covariance when inheriting from an *abstract* base class but not when inheriting from a concrete base class. The hope was that since the client cannot construct an instance of an abstract class, the problem would not arise. But as Hursch [9] shows this doesn't always work. He constructs an example with an abstract `Point` class and two (concrete) derived classes `ColoredPt` and `ThreeDPt` and writes client code that results in calling the `eq` function of the `ThreeDPt` class with a second argument that is an instance of `ColoredPt`.

One important point has often been ignored in the debate over contra versus covariance. Even contravariance only ensures *syntactic or type compatibility*. As Liskov and Wing [13] point out, it does *not* ensure that if D is a derived class and B the base class, that objects of type D are *conceptually* actually objects of type B. The methods defined in D could behave *entirely differently* from the methods that they override (while still obeying the contravariance restriction). To take care of this [13] proposes extending contravariance to include a *semantic* requirement. Essentially the *pre-condition* of the function defined in the base class would have to imply the pre-condition of the corresponding function in the derived class, and the *post-condition* of the latter would have to imply the post-condition of the former. If this requirment is not met, the user will discover (or worse – fail to discover!) that his program, although it does not produce any 'message not understood' error, behaves differently when he uses objects that are instances of the derived class than when he uses objects of the base class.

We will see in the next section that the problem with synchronization constraints of the kind proposed in [7] is also semantic.

3 Redefining synchronization constraints, or When is a bounded buffer not one?

Let us start with an example from Frolund [7]. Consider a standard *bounded-buffer* class of size *MAX* that provides the usual *put* and *get* operations. The synchronization constraints for this class expressed in the notation of [7] are:

($size \geq MAX$) `prevents` *put*

($size \leq 0$) `prevents` *get*

This ensures that the *put* operation is not executed when there is no space in the buffer and the *get* operation is not executed when the buffer is empty.

Suppose now we wish to define a new class, say, *bounded2-buffer* with a new operation, *get2* that allows the client to retrieve *two* elements at once from the buffer. Clearly we will have to provide a new synchronization constraint corresponding to this operation to ensure that it is not executed if the buffer is empty or has only one element. We can write this additional constraint as:

($size \leq 1$) `prevents` *get-2*

This constraint is added to the earlier constraints inherited from the *bounded-buffer* class.

This examples demonstrates the need for additional synchronization constraints if we want to define a derived class that provides operations not defined in the base class. Presumably *bounded2-buffer* is a better buffer class than *bounded-buffer* since it can be used wherever the latter can be used, plus it allows an additional operation.

But suppose we wish to define a *bigger-buffer*, one that is twice as big as a *bounded-buffer*, rather than one that defines an additional operation. The corresponding synchronization constraint would be

($size \geq 2 * MAX$) `prevents` *put*

But we cannot define *bigger-buffer* as a derived class of *bounded-buffer* in the formalism of [7] because the synchronization constraint of the latter class preventing the application of *put* if the buffer already has *size* elements would be *inherited* and would prevent the client of *bigger-buffer* from using the *bigger-buffer* object to store more than *MAX* elements. So in the formalism of [7] there would be no reasonable way to define *bigger-buffer* by inheriting from *bounded-buffer*. But clearly, any user program that works properly when using an instance of *bounded-buffer* class will work equally well if we were to substitute an instance of *bigger-buffer*. In other words, an instance of *bigger-buffer is-an* instance of *bounded-buffer*, so the former class should be definable as a derived class of the latter.

In fact, if we are allowed to strengthen synchronization constraints in the derived classes, as in systems like [7], we will be able to define a class *smaller-buffer* as a derived class of *bounded-buffer*:

($size \geq MAX \div 2$) `prevents` *put*

This suggests that a *smaller-buffer* object is also a *bounded-buffer* object. But that is not true! If a client program that is functioning correctly were to replace an instance of *bounded-buffer* by one of *smaller-buffer*, it might well lead to deadlock! In other words, from the client's point of view, a *smaller-buffer* object cannot be considered as a *bounded-buffer* object, so the former class should not be defined by inheritance from the latter.

We have arrived at our central dilemma. The *bounded2-buffer* example suggests that we should be able, in the derived class, to strengthen the synchronization constraints specified in the base class. The *bigger-buffer* / *smaller-buffer* example dictates that if we do that, client programs may be hurt. This is similar to the covariant – contravariant controversy in the sequential case. Except here it is more serious, because while in that situation the client-view supported one point of view (contravariant) and the class implementer-view supported the other (covariant view), in the current situation the client-view seems to be supporting both points of view depending on the details of the example.

Before we proceed to look at possible solutions, we should mention that other proposals than [7] have the same problem. Consider, for instance, the system of Lopes and Lieberherr [12]. The central argument of [12] is that an important source of difficulties in concurrent OOP, in particular of the inheritance anomaly, is that code that ensures the synchronization constraints is often mixed in with code for performing individual operations. While this argument is also the starting point of [7] and others, Lopes and Lieberherr propose a different solution: A class in [12] consists of a *synchronization pattern* and a *behavioral block* which essentially contains the bodies of the various methods without any synchronization code in them. ([12] also introduces a *propogation pattern* which doesn't need to concern us here.). The compiler is then expected to combine these two parts appropriately to obtain the actual code, including that required to ensure appropriate synchronization requirements, corresponding to the methods. Thus these *synchronization patterns* are compile time entities; all that exists at at run-time is code for the methods of the various classes with synchronization code embedded.

A derived class in [12] will inherit both the synchronization pattern and the behavior block of its parent class. We can then modify either or both of these as appropriate. Again the compiler is responsible for compiling the derived class from the information inherited from the parent and the changes that have been made in defining the derived class. One problem with the approach of [12] is that their synchronization patterns tend to be quite complex, sometimes exceeding in size the code for all the methods of the class. More important for the problem we are considering is that [12] doesn't really impose any conditions on *what* changes may be made in the derived classes to the synchronization patterns inherited from the base class, so exactly the same problem we saw can be reproduced in this system.

Let us now return to our *buffer* examples. The key difference between *bounded2-buffer* and *bigger-buffer* is that while the former extends the capabilities of *bounded-buffer* by providing additional operations, the latter allows certain sequences of calls to *put* and *get* that *bounded-buffer* does not allow. But any sequence of calls to *get* and *put* allowed by the *bounded-buffer* class should also be allowed by both *bounded2-buffer* and *bigger-buffer*. This is precisely the condition we need: *while the derived classes may impose additional synchronization constraints to take care of new operations like* get2 *that they provide, any sequence of operations that satisfies the constraints of the base class must also satisfy the constraints of the derived class*.[1]

How do we ensure that this requirement is met? That depends partly on the notation that we adapt to specify the synchronization conditions: synchronization constraints of the kind in [7], synchronization patterns as in [12], method sets as in [10], or as in [14] etc. In none of these cases though will the check be simple. The point is that all of these have been designed to deal with the problem of inheritance anomaly to one extent or the other, i.e, to make it easy to reuse and modify in the derived class synchronization constraints of the base class. None of them was designed from the client's point of view. We are currently studying each of these methods to see which makes it easiest to check our requirement. We are also exploring the possibility of developing an alternative notation for expressing constraints that will make it difficult to design derived classes that violate our requirement.

Before we conclude this section let us consider another example. Suppose we have a number of different types of resources in a system. It would seem useful to introduce an abstract base class *abstract-resource* of which each type of resource is a derived class. In the abstract class we would expect to specify constraints that would apply to all types of resources (such as we cannot allocate more instances of a particular type of resource than we have), and in the individual concrete classes we would expect to specify additional constraints that apply to the particular type of resource that the individual class represents. This would seem to violate our requirement since in the derived class(es) we are adding to the constraints specified in the base class. Allowing such constraints to be added in the concrete derived classes can indeed lead to the same kind of problems as we saw earlier. Thus a client program that declares a variable of type *abstract-resource* and then instantiates it to an instance of any one of several *concrete-resource* classes may work

[1] This will not be the case for *smaller-buffer* so that class should *not* be defined as a derived class of *bounded-buffer*.

perfectly well and then fail when instantiated to an instance of a new *concrete-resource* class. For instance, the client program may deadlock because one of the synchrnization constraints added in the new concrete class is not met. The suspicion would naturally fall on the new class when in fact it is the client program which is at fault for using an instance of this class without meeting its constraints. Or according to our analysis above, the *concrete-resource* classes should not be considered derived classes of the *abstract-resource*. A more pragmatic answer might be to allow derived classes of abstract base classes to add such constraints but client programs that declare variables of type *abstract-resource* would have to be analyzed to ensure that they are correct with respect to the synchronization constraints of *all* possible concrete classes whose instances might be assigned to these variables.

4 Discussion

As in the sequential case, inheritance seems to be both a source of power and of difficulties in concurrent OOP. On the one hand, inheritance allows for code reuse in defining related classes and also allows the user to treat in a uniform manner, via the mechanisms of polymorphism and dynamic dispatch, objects that are instances of different but conceptually similar classes. On the other hand, if used with no constraints, these advantages of inheritance can be seriously compromised. We may end up with little or no code reuse, i.e., having to redefine from scratch all the methods in the derived classes including the ones that have been defined in the base classes, or having the client program enounter run time errors like ‘message not understood’ or, worse, produce incorrect results, when the client uses objects that are instances of the derived classes in place of instances of the base class.

The solution in the sequential case was first to use the co / contra-variance rule, i.e., when redefining functions, use covariance with respect to arguments of the function that are used for dynamic dispatch, and contravariance with respect to the remaining arguments. For standard languages this means use covariance only on the first argument (often also called the implicit argument) since that is the only one used in the dynamic dispatch. But this only takes care of error messages from the run-time system. To ensure that the behavior of the program is as expected by the client we must also, as Liskov and Wing [13] point out, worry about pre- and post-conditions of the methods redefined in the derived classes and show that these are appropriately related to their counterparts in the base class.[2]

In the concurrent case there is the added complication of synchronization conditions in the base class and what changes may be reasonably made to these in the derived classes, and how these changes may be made. We have been primarily concerned with the first of these questions, i.e., what changes may be made in the derived classes to the synchronization constraints specified in the base class so that the client program's behavior is not adversely affected when it uses an instance of the derived class in place of an instance of the base class. From this point of view, the conclusions of our analysis are natural. A user program that works properly, for instance is free from deadlock, when it uses an object that is an instance of a given class may well deadlock if we substitute an object that is an instance of a different class if the synchronization conditions are more demanding. Thus from the client's point of view, derived classes should not impose additional conditions beyond those imposed by the base class. This is exactly like the Liskov-Wing condition on pre-conditions of methods defined in derived classes and the pre-conditions of the corresponding methods in the base class. The former should not be more demanding than the latter. In general, in fact, both the requirements that [13] impose as well as the ones we have suggested are natural consequences of the basic relation between derived classes and base classes – an instance of the derived class *is-a*n instance of the base class and the user should be able to it treat as such. Of course, even the co/contra variant rule was a consequence of this requirement at an even more basic level, that substituting a derived class object in place of a base class object should not lead the client program to get the 'message not understood' message from the run-time system.

Frolund[7] argues that there is no need to guarantee that a client program that works correctly when it uses a base class object, will also work correctly if we replaced the base class object by a derived class object. But this guarantee, we feel, is a key requirement for inheritance. Without it mechanisms like dynamic dispatch will fail completely. If the argument in [7] were accepted, we would have to give

[2]As an aside we note that [13] also require, in effect, that any new methods introduced in the derived classes be 'explainable' in terms of applications of methods already defined in the base class. This seems too restrictive, especially in the sequential case, since if the client uses an instance of the derived class in place of an instance of the base class, he is not going to be invoking any of these new methods, so they shouldn't affect his program. In the concurrent case, if the same object can be used by another process concurrently then it is conceivable that the other process might indeed invoke one of these new methods. It may be interesting to see if the condition in [13] can be weakened in view of this.

free rein to the designer of the derived class to build arbitrary behaviors into the derived class objects. We feel such complete freedom would make it much harder for clients to use classes defined using inheritance.

We conclude with two open questions. There have been some attempts [see for example 1] to apply OO ideas to real-time programs. Evidently the same type of question that we have analyzed with respect to synchronization constraints would arise with respect to the types of real-time constraints that derived class objects would have to satisfy compared to the constraints satisfied by the base class objects. We hope to answer this question in a future paper.

All of our discussion in the current paper has been with respect to synchronization conditions with respect to *one* object. In more complex systems, we would clearly need to be able to deal with conditions relating two or more objects, possibly instances of different classes. There has been some work in studying the question of multi-object coordination (see for example [5, 8]). Here again the question of what kinds of changes in these conditions ought to be allowed in derived classes needs to be analyzed. As may be expected, this will be a far more difficult question than the one we have considered.

5 References

1. M. Aksit, J. Booch, W. Sterren, L. Bergmans, Real-time specification inheritance anomalies and real-time filters, in Proceedings of *ECOOP* '94, 1994.

2. P. America, F. van Linden, A parallel OO language, Proceedings of *OOPSLA* '90, 1990.

3. L. Cardelli, A semantics of multiple inheritance, *Inf. Comput., 76,* 138-164, 1988.

4. G. Castagna, Covariance and contravariance, *ACM TOPLAS, 17,* 431-447, 1995.

5. C. Chambers, OO multi-methods in Cecil, Proceedings of *ECOOP* '92.

6. M. Dodani, C. Tsai, ACTS: a type system for OOP, Proceedings of *ECOOP* '92.

7. S. Frolund, Inheritance of synchronization constraints, Proceedings of *ECOOP* '92.

8. S. Frolund, G. Aha, A language framework for multi-object coordination, Proceedings of *ECOOP* '93.

9. W. Hursch, Should superclasses be abstract, Proceedings of *ECOOP* '94.

10. Y. Ishikawa, Communication mechanisms on autonomous objects, Proceedings of *OOPSLA* '92.

11. D. Kafura, K. Lee, Inheritance in Actor based languages, Proceedings of *ECOOP* '89.

12. C. Lopes, K. Lieberherr, Abstracting process to function relations, Proceedings of *ECOOP* '94.

13. B. Liskov, J. Wing, A new definition of the subtype relation, Proceedings of *ECOOP* '93.

14. S. Matsuoka, A. Yonezawa, Analysis of inheritance anomaly in concurrent OO languages, in Research directions in concurrent OO programming, ed. Agha, Wegner, Yonezawa, 1993.

15. J. Meseguer, Solving the inheritance anomaly, Proceedings of *ECOOP* '93.

16. B. Meyer, Object oriented software construction, Prentice-Hall, 1988.

17. J. van den Bos, C. Laffra, PROCOL, *Acta Informatica, 28*, 511-538, 1991.