On the Specification, Inheritance, and Verification of Synchronization Constraints

Neelam Soundarajan

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

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

Key phrases: Specification and verification; Synchronization constraints; Inheritance anomaly.

Abstract: Object-orientation and distributed systems are a natural match. Objects correspond to processes in a distributed program; the invocation of a method of one object by another object corresponds naturally to a message being passed between the corresponding processes in the distributed 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 slightly different. We are interested in developing ways to abstractly *specify* these synchronization constraints, and ways to *verify* them. In other words we are interested in *what* these synchronization constraints do, and this is, of course, the critical question from the point of view of the users of these objects. We use the mechanism of *acceptance sets* in our specifications. We develop a proof method to verify that (base as well as derived) classes meet their specifications. We also consider the question of what kinds of modifications of synchronization constraints in the derived classes are easy for the clients of the class to deal with.

1 Introduction

Object-orientation (OO) and distributed systems are a natural match. Active, autonomous objects correspond naturally to processes in a distributed 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 distributed systems has been 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, without having to repeat what the base class already provides. Although some authors have criticized inheritance, others like Meyer [6] have presented persuasive arguments in its favor. One of the problems in applying OO ideas to concurrent programming is the *inheritance anomaly* [MandY]. To see the problem, consider the standard example of a *bounded-buffer* class. The problem arises because the 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*); as a result, if we attempt to develop a derived class, for example a *better-buffer* class that provides an additional operation *get2* that allows us to read two elements from the buffer, we may be required not only to provide the code for this additional operation, but also to rewrite the code of some of the existing operations to take account of appropriately synchronizing with the new operation. In other words, although the code for *performing* the existing operations, *get* and *put* in the case of the *bounded-buffer* class, 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. Various notations have been proposed to solve this problem, and we will consider some of them later in the paper.[1]

Our interest in the current paper is somewhat different. From the point of view of the *client* of the class(es), the important question is not how the classes are constructed, but how they can be used. To serve this purpose, we need formal and

[1] [MandY] consider various schemes that have been proposed to specify the constraints and for each scheme present an example that forces us revisit the code in the base class because of the way in which the synchronization constraints are expressed.

abstract specifications of both the functional properties of the various operations provided by the classes, as well as their synchronization properties. Further, we need appropriate axioms and proof rules that can be used by the class designers to establish that the operations, either inherited from the base class, or defined or redefined in the derived class, do indeed meet these specifications.[2] We develop a simple notation for expressing such specifications, and develop a method for showing that classes do meet their specifications. While object-oriented distributed systems can be expected to be easier to design and understand than non-OO systems, easy-to-use specification notations and verification methods of the kind developed in this paper are important to ensure that the resulting systems are reliable and behave as they are intended to.

Inheritance anomaly which is essentially a problem with how synchronization constraints are implemented in certain classes, has a counterpart in the specification/verification task. Indeed there it appears even in the absence of synchronization issues. The problem arises because an operation defined in a derived class may modify base class member variables. As a result, the derived class designer may be forced to reverify all of the base class operations, including those that are not redefined in the derived class, since these operations also use the same variables. We have developed a formal approach in [9] to simplify this task in the sequential case. The key idea behind the approach in [9] is to have *two* formal models of the (base) class. The first is the usual, abstract model for use by the client of the class. The second, for use by the designer of the derived class, is a *concrete* model in which operations of the (base) class are specified in terms of pre- and post-conditions on the *actual* data structures used in the representation, rather than in terms of an abstract math model. As a result, the derived class designer only needs to ensure that when an operation defined (or redefined) in the derived class finishes, the values of the various member variables including those inherited from the base class are acceptable to the operation that might be next invoked, according to the concrete pre-conditions of the various operations. The key point is that the concrete specification in terms of the actual data structures contains the additional information that a derived class designer needs, and that is missing from the abstract specification. With this additional information in hand,

[2] We will also need an appropriate formalism that clients of the class can use to show, given the specifications of various classes, that their own code that uses instances of these classes, is correct, i.e., is free of deadlock, or exhibits appropriate functional behavior, etc. We will not consider this formalism in the current paper. It should be possible to design such a formalism along the lines of existing systems such as [8, 7] that are applicable in the absence of user defined classes.

the derived class designer does not need to look into the actual code of the base class operation.

The formalism we develop in the current paper is a natural extension of that in [9]. Each class will have two specifications associated with it, an *abstract specification* for use by a client of the class, and a *concrete specification* for use by a derived class designer. Each specification will give us information about the functionality, i.e., input-output behavior of each operation in the class, as well as –and this is the extension being proposed in the current paper– about the synchronization properties, i.e., what operations may be invoked at various points during the execution of the program. As in [9], the abstract specification will be in terms of an abstract mathematical model, whereas the concrete specification will be in terms of the actual data structures used in the class. We should note that we are *not* proposing a solution to the inheritance anomaly. Rather we are addressing the question of how behavior of classes, including synchronization behavior, can be specified and verified especially in the case when we use inheritance in defining some of the classes.

In the next section of the paper we introduce our specification notation by means of a few simple examples. In the third section we present our proof system using which we will be able to establish that a given class meets its specifications. One important point to note is that our specification notation as well as the general method of verifying that a given class meets its specifications is applicable to the various (programming) notations that have been proposed in the literature. Of course, the details of the proof method, and to an extent the primitives needed to express appropriate properties, do depend to an extent on the details of the programming mechanisms under consideration, but the overall approach is generally applicable. Thus in the third section we consider two different approaches (those of [1] and [Kafura]) to using inheritance in conjunction with concurrency and consider how our specification and verification method may be applied in each case. In the final section we briefly summarize the main ideas behind our approach, and consider some open problems.

2 Abstract and Concrete Specifications

Consider a class C with public methods $m_1, \ldots, m_n$. (We will generally use C++-like terminology and notation.) We will provide two specifications for C, an *ab-*

stract specification for use by clients of C, and a *concrete specification* for use by designers of derived classes of C. Let us first consider the abstract specification. A client of C will need, first, the pre- and post-conditions of each m_i in terms of an abstract model of C. Second, the client must be able to tell at what points in the execution of the system the various methods of C will be enabled.[3] The most direct way of specifying this information is in terms of the value of the *acceptance set*, i.e., the set of methods that are enabled at the point in question. This specification could be provided in the form of a function over the *trace* or sequence of calls made so far to the various methods of C. In other words, we could specify a function $\mathcal{A}$ whose value $\mathcal{A}(t)$ for any given trace t is the subset of methods $\{m_1, \ldots, m_n\}$ that can be invoked at a given point if the sequence of methods that have been invoked thus far is t. While this approach would allow us to handle all types of synchronization schemes, it is usually more than is needed. A simpler approach would be to specify a *synchronization property* SP_C that imposes appropriate restrictions on the value of the acceptance set $\mathcal{A}$ in terms of the abstract model of C, and this is the approach we will use. (One disadvantage of the simpler approach is that we may not be able, as we will see in a later example, to deal with all possible types of synchronization behaviors.)

Let us consider the following class Bbuffer, the standard *bounded-buffer* example:

```
class Bbuffer {
 // Buffer of size n
 public:
  Bbuffer();
  void put(int k);
  int get( );
 protected:
  ⋮
};
```

A simple abstract model $M_{\texttt{Bbuffer}}$ for this class would just be a sequence of integers with initial value the empty sequence $\langle\rangle$. The `put` operation would append its argument to the current value of the sequence; the `get` operation would return the first element of the sequence as its return value and remove this element

[3] If an operation is invoked when it is not 'enabled', the call will be suspended, to be carried out at a later time when the operation is enabled. The call can become enabled at a later time because of calls from other concurrent clients who share this object with the current client.

from the sequence. Thus the first part of the abstract specification, the pre- and post-conditions of these operations is:

$\{$ *true* $\}$ `put( k )` $\{$ $self_a = \#self_a \mathbin{\hat{}} \mathtt{k}$ $\}$

$\{$ *true* $\}$ `get( )` $\{$ $value = head(\#self_a) \wedge self_a = tail(\#self_a)$ $\}$

where $self_a$ is the abstract bounded buffer object, *value* denotes the value returned by the operation under consideration; ˆ is the *append* operation; *head*, and *tail* are the usual functions on sequences; $\#$ denotes the value of the variable that follows the $\#$ at the start of execution of the operation in question.

Note that the pre-conditions of the operations are *true* rather than that there must be space in the buffer, or that the buffer must be non-empty respectively. That is because these conditions are part of the synchronization constraints, rather than the pre-conditions of the respective operations. If we had included these conditions as part of the pre-conditions, then that would mean that if the client were to invoke, say, the `get` operation when the buffer was empty, the (class) would be at liberty to do anything (such as returning a random value) since its pre-condition would not have been satisfied. By expressing the condition as part of the synchronization constraint, the client is assured that in this circumstance, the call will be suspended, to be resumed if and when the appropriate synchronization constraint is satisfied – as a result of calls by other concurrent clients who share this buffer. This will happen if the other client invokes the `put` operation, after the execution of which the buffer will no longer be empty.

So next we need to specify the abstract synchronization property, $ASP_{\mathtt{Bbuffer}}$ which will capture these synchronization constraints:

$$ASP_{\mathtt{Bbuffer}} \equiv [\ (self_a \neq \langle\rangle \Rightarrow \mathtt{get} \in \mathcal{A}_{\mathtt{Bbuffer}}) \\ \wedge\ (|self| < n \Rightarrow \mathtt{put} \in \mathcal{A}_{\mathtt{Bbuffer}})\ ]$$

where $\mathcal{A}_{\mathtt{Bbuffer}}$ is the *acceptance set* of `Bbuffer`, i.e. the set of all methods of `Bbuffer` that can currently be invoked. $ASP_{\mathtt{Bbuffer}}$ essentially states that the method `get` can be invoked if the buffer is not empty, and that `put` can be invoked if the buffer is not full.

Next, consider the *concrete specification* of a class C. While the abstract specification of C gives us information about the functionality of each method and the synchronization properties of the various methods of C in terms of its abstract model, the concrete specification of the class will provide us with similar information in terms of the *internal* representation, i.e., the actual data members of C. The client of the class of course has no use for this information since she has no direct access to the data members of the class. (We are assuming here, following widely accepted principles of OO design, that no data members of a class are declared `public`.) But a *designer* of a class that inherits from C is in acute need of this information. Without this information that designer would be forced to study the actual code of the various operations of C to understand how they operate on the various data members, as well as the synchronization code to see what conditions the values of the data members must satisfy in order for specific operations to be enabled. It is only then that this designer will be able to ensure that the new operations that she introduces in the derived class, as well as redefinitions that she makes of operations inherited from the base class, are consistent with the design of the base class. Our concrete specification for C will provide this information in the form of pre- and post-conditions on the actual data members for each operation of C; and a synchronization property, again on the data members. We will term these pre- and post-conditions and the synchronization property 'concrete' in order to distinguish from their counterparts in the abstract specification. In addition, as in [9], the concrete specification should include an invariant on the data members of the class; since our interest in this paper is on synchronization issues, we will generally ignore this invariant. Thus the concrete specification of C *will* contain more information than its abstract specification, but the details of the actual code bodies of the various operations of C will be abstracted away. That is the big advantage for the derived class designer; she needs to deal only with the concrete specification of C, not the actual code.[4]

Consider again the `Bbuffer` class. Suppose we use an array `Elems[0:n-1]` to store the elements currently in the buffer, and two variables `in` and `out` as pointers into this array to indicate where the next element added to the buffer must be stored, and where the next element read from the buffer should be fetched from. Thus the protected part of the class would look like:

```
protected:
 int Elems[n];
 int in, out;
```

The constructor function of the class will simply initialize `in` and `out` to 0. The operations `put` and `get` will store elements into the buffer and return elements from the buffer using the `Elems` array as a 'circular' array, i.e., increasing, modulo `n`, the appropriate `in` or `out` pointer by 1.

[4] In this paper we will assume that all data members are `protected`, there being no `private` variables. Recall that `private` variables would not be accessible even in the derived class. In [9] we explain how we can deal with both kinds of variables.

Thus the concrete pre- and post-conditions of these operations are:

$\{$ *true* $\}$ `put( k )` $\{\, \mathtt{in} = \#\mathtt{in} \oplus 1 \wedge \mathtt{Elems} = \#\mathtt{Elems}[\#\mathtt{in} \leftarrow \mathtt{t}] \,\}$

$\{$ *true* $\}$ `get( )` $\{\, \mathit{value} = \#\mathtt{Elems}[\#\mathtt{out}] \wedge \mathtt{out} = \#\mathtt{out} \oplus 1 \,\}$

where the post-condition of `put` asserts that the value of `Elems` when `put` finishes is the same as when it started except the value of `Elems[#in]` is `k`. (The concrete invariant for this class would contain clauses like $0 \leq \mathtt{in} < \mathtt{n}$.)

As in the case of the abstract specification, the pre-conditions of the operations are just *true*. The requirements of a non-empty buffer for `get` and a non-full buffer for `put` are captured by the concrete synchronization property:

$$\begin{aligned} CSP_{\mathtt{Bbuffer}} = [\, &(\mathtt{in} \neq \mathtt{out} \oplus 1 \Rightarrow \mathtt{put} \in C\mathcal{A}_{\mathtt{Bbuffer}}) \\ &\wedge (\mathtt{out} \neq \mathtt{in} \Rightarrow \mathtt{gett} \in C\mathcal{A}_{\mathtt{Bbuffer}}) \,] \end{aligned}$$

where we have used $C\mathcal{A}_{\mathtt{Bbuffer}}$ to denote the concrete acceptance set of `Bbuffer`, although in fact there is no difference between the concrete and abstract acceptance sets.

So far we have only considered the specifications for the class. We still need to verify that the class, as actually implemented, meets these specifications. This verification method is the topic of the next section. Once that is done though, neither the client of the class nor the designer of the derived class would have to look at this actual implementation.

3 Verification of Class Behavior

The verification task may be divided into two parts. First, we need to verify that the class operations and synchronization requirements as actually implemented, meet the concrete specification of the class. Second, that the concrete specification in some precise sense implies the abstract specification. The second part is the easier of the two and we deal with it in 3.1. How we handle the first part depends upon the details of the particular programming notation, specifically the notation used to express synchronization code. In 3.2 we consider two very different programming notations and explain how to deal with each one.

3.1 Verifying Abstract Specifications

In order to show that the concrete specification implies the abstract one, we must first define an abstraction function ε that maps the concrete state to the abstract state. Thus if ω is a concrete state, $\varepsilon(\omega)$ is the corresponding abstract state as seen by a client of the class. Note that since the abstract acceptance set $A\mathcal{A}$ is the same as the concrete one $C\mathcal{A}$, ε will map $C\mathcal{A}$ to $A\mathcal{A}$. Next let us introduce some conventions. Let f be any of the methods of the class. Let $a.pre_f$ and $a.post_f$ be its abstract pre- and post-conditions; similarly $c.pre_f,\ c.post_f$ are its concrete pre- and post-conditions. CSP be the concrete synchronization property, and ASP the abstract one. Then in order to show that the concrete specification implies the abstract one we need to establish the following ($(1a)$ is for each f):

$$a.pre_f(\varepsilon(\omega)) \Rightarrow c.pre_f(\omega) \tag{1a}$$

$$c.post_f(\omega) \Rightarrow a.post_f(\varepsilon(\omega))) \tag{1b}$$

$$CSP(\omega) \Rightarrow ASP(\varepsilon(\omega)) \tag{2}$$

These implications simply represent the fact that the abstract and concrete specifications are directly related via the mapping function ε. $(1a)$ and $(1b)$ are simplified versions of the corresponding rules in [9]; the rules in [9] take account of the concrete invariant. We should also note here that while $(1a),\ (1b),\ (2)$ are generally sufficient, they may have to be modified, as we will see in section 3.2, depending on the details of the programming notation.

3.2 Verifying Concrete Specifications

Now consider the first part of the verification task. This task consists of two subtasks. First we must verify that the bodies of the various methods meet their respective concrete specifications, i.e., their pre- and post-conditions. Second that the class as a whole meets CSP, the concrete synchronization property. So far we have been able to ignore the programming language notation in which these method bodies and synchronization code is written but now we must take account of this notation. The interesting question here is how the various notations that have been proposed in the literature [such as those of 1, 3, 2, 5] for specifying the synchronization conditions, may be handled.[5] We will consider, in turn, two of

[5] We will assume that the non-synchronization portion of the individual methods are written using a standard collection of constructs and will omit discussion of them.

these notations (those of 1, and 3); we believe a similar approach will work with the others.

Frølund [1] proposes a natural approach to expressing synchronization code in classes. His proposal is to use *method guards* to express synchronization conditions. A method guard specifies a condition that *disables* a particular operation. Thus, for instance, in the case of the bounded buffer, one of the method guards would be:

$(\texttt{out} = \texttt{in})$ *prevents* `get`

This states that the `get` operation is *disabled* if the condition $(\texttt{out} = \texttt{in})$ is satisfied.

How do we capture, in our specification, the effect of such a method guard? Consider, for instance, the guard above. All we need to do, corresponding to this guard, is to include the following clause in our (concrete) synchronization property *CSP*:

$(\texttt{out} = \texttt{in}) \;\Rightarrow\; \texttt{get} \notin C\mathcal{A}_{\texttt{Bbuffer}}$

Thus each method guard g in the program would be represented in the *CSP* by the corresponding clause *CSP*(g) as above. If a class has guards $g_1, \ldots, g_m$, the *CSP* for this class is be *CSP*$(g_1) \wedge \ldots \wedge$ *CSP*(g_m); this assumes that this class is a base class.

An important part of Frølund's proposal is that method guards become increasingly more restrictive in derived classes. In other words derived classes can only *add* new method guards; method guards of the base class will necessarily be inherited by the derived class and become part of its collection of method guards. Correspondingly all we need to do in our formalism, to obtain the CSP_D corresponding to a derived class is to conjunct to the base class CSP_B, the clauses *CSP*$(g_1) \wedge \ldots \wedge$ *CSP*(g_m) corresponding to the guards $g_1, \ldots, g_m$ of the derived class.

Two final points are worth noting: The requirement that synchronization conditions can only become more restrictive in derived classes does mean that Frølund's approach will not work in certain circumstances. For instance, if we wanted to define a derived class of the `Bbuffer` class which was of size $2n$ (where instances of `Bbuffer` class are of size n), we could not do so. Nevertheless, it is partly because of this restriction that our formalism corresponding to this approach turned out to be relatively simple. Second, Frølund has a construct called `all-except` which allows us to write guards that disable all methods except those that are named in the statement. To handle this construct, we would have to assume that we have a set called, say, *AllMethods* which is the set of all method names. The clause corresponding to an `all-except` guard would then simply say that none of the method names in *AllMethods* less those named in the construct are not in the set $C\mathcal{A}$ of the class (if the specified condition is satisfied).[6]

Next consider the approach of Kafura and Lee[3]. They introduce the notion of *behavioral abstraction* which is more or less the same concept as our acceptance set. The value of the behavioral abstraction at any point is the set of method names that are *enabled* at that point. Each method body, as its last action, is required to execute a `become` command that essentially assigns a new value to the acceptance set. The reason for the name 'behavioral abstraction' is that separately from the bodies of the individual methods, in a (protected) section that may be called the '`behavior`' section, the class designer identifies a number of subsets of the set of all methods, and names these subsets with distinct names, say $B_1, \ldots, B_m$. Note that these subsets need not, and usually are not, disjoint from each other. The `become` statement at the end of each method is then of the form `become` B_j which says that when this method finishes, the set of enabled methods will be all (and only) those whose names appear in B_j. Thus the details of which methods are enabled are partly abstracted away in the definitions in the `behavior` section.

Further, when a class is defined by inheritance from another class, the derived class introduces its own `behavior` section. The corresponding behavior abstractions, let us call them, $B'_1, \ldots, B'_m$ are defined either from scratch or in terms of the $B_1, \ldots, B_m$ of the base class. An important construct that Kafura and Lee introduce is the `redefines` construct. This allows us them to redefine, in the derived class, the set of methods in a given behavior abstraction B_i of the base class. The advantage of this is that code that is inherited from the base class –although it has apparently not been rewritten– may do something different than it does in the base class! To see how this may happen, suppose that in a base class C the behavioral abstraction B_1 has been defined as the set of methods $\{g_1, g_3\}$. Suppose also that the method g_4, when it finishes, executes the command `become` B_1. Suppose also that in a derived class $C1$ of C we redefine B_1 to be the set $\{g_1, g_2\}$. Then, if we consider a call `a`.g_4 where `a` is an instance of C, at the end of the call the enabled methods will be $\{g_1, g_2\}$. If we consider a call `b`.g_4 where `b` is an instance of C_1, at the end of *this* call the enabled methods will be $\{g_1, g_2\}$!

[6] Note that this does *not* say that the methods that are named in the construct *are* actually in $C\mathcal{A}$; some or all of them may well be prevented from being in $C\mathcal{A}$ because of other guards.

This is the important capability of their approach that allows Kafura and Lee to tackle several cases of inheritance anomaly – because with this mechanism, they are not forced to rewrite the code of operations like g_4 just in order to change, in the derived class, the set of methods enabled when it finishes.

How do we handle such a mechanism in our approach to specification and verification? We should first make an important point: The *client* of the classes C and C_1 has no interest in mechanisms like behavioral abstractions used by the designers of these classes to avoid the inheritance anomaly. What the client is interested in, as far as synchronization issues are concerned, is only information about the set of operations that are enabled at various points. Correspondingly, in our formalism, no information about behavioral abstractions will appear in the abstract specification of a class. This information must, however, appear in the concrete specification, for use by the derived class designer.

In order to deal with behavioral abstractions, we need to split the concrete synchronization property CSP_C of the class C into two parts. The first part, ${CSP^d}_C$ will give us the *definitions* of the various behavioral abstractions, i.e., for each B_i defined in the class, the set of method names corresponding to B_i. The second part, ${CSP^c}_C$ will specify the actual synchroniztion conditions, but in terms of the behavioral abstractions, rather than in terms of the actual method names. Moreover, we also need to include information about which behavioral abstraction will be enabled at the end of each method of the class, since this information is also of great importance to the derived class designer, as should be clear from the earlier discussion involving methods g_1, g_2 etc. In order to do this, ${CSP^c}_C$ will be expressed as a conjunction of a number of clauses, each of the form:

after g_i $::$ $(b \Rightarrow B_j)$

where b is a boolean (on the member variables of C). This clause means that following the execution of g_i, if the condition b is satisfied, then the set of methods in B_j will be enabled. The abstract synchronization property, ASP_C, will also consist of such clauses, except that rather than the behavioral abstractions B_j, the actual set of method names will appear in the corresponding clauses, and the conditions b will be on the abstract model of the class, rather than the concrete member variables.

${CSP^d}_C$ will be established on the basis of the definition in `behavior` section of the class C. Essentially it is identical to this section if C is a base class. For a derived class $C1$ of C, we obtain ${CSP^d}_{C1}$ from ${CSP^d}_C$ by adding the new definitions provided in $C1$ and replacing the ones that are `redefined` in $C1$.

The individual clauses of $CSP^c{}_C$ will be established by considering the individual methods of C. If a method g_i is inherited without change from the base class, then we need do no work as far as the corresponding clauses in $CSP^c{}_C$ are concerned. This is the verification analog of the implementation reuse of Kafura and Lee's system. If the derived class implementor can reuse the base class operation although the behavioral abstraction that it enables when it finishes may mean a different set of methods in the derived class, then we, as verifiers, do not have to reverify the method body either. Of course, if the method g_i is redefined in the current class, or if it is entirely new, then we do have to verify the appropriate clause of CSP^c. To do this, we need a little bit of formal machinery to allow us to establish the appropriate clauses of CSP^c, i.e., the ones that begin *after* g_i ; we will omit the details of this machinery since it is fairly straightforward.

One final point is worth noting. When we apply the rule (2) of section 3.1 to the system of behavioral abstractions, we should note that CSP is the conjunction of CSP^d and CSP^c. The former gives us the sets of methods corresponding to the various B_j's that appear in the latter and it is these method names that appear in ASP. It is not clear to us whether information about different methods being enabled following calls to certain other specific methods is of interest to the client of the class or not. In any case, if we decide that it is, we will have to write not just the concrete synchronization property, but also the abstract one as a conjunction of `after` g_i clauses.

4 Discussion

One school of OO thinking (see, for example, [4]) has argued that inheritance should be used in a very limited way, specifically that it should not be used for purely implementation purposes. In other words, if D is a derived class of C, then it must be the case that instances of D behave in ways that are strictly consistent with behaviors that instances of C might exhibit. But other schools disagree. And indeed, it would seem that a better approach would be to develop ways of reasoning about different types of inheritance usage rather than disallow some usages by fiat. In the case of using OO ideas for concurrent or distributed systems, various proposals in the literature (1, 3, 2, 5) for expressing synchronization constraints have been designed to allow derived classes to exhibit behavior, in particular synchronization behavior, different from that exhibited by the base class.

An important consideration in all of these proposals is, of course, to try to limit the amount of work that needs to be done in the derived classes, and to take as much as possible from the base class.

Our goal in this paper has been to develop a general approach that can be used for specifying and reasoning about the behaviors of systems built using the notations proposed by these authors. The key idea underlying our approach is that each class needs *two* sets of specifications, one for use by clients of the class, the other for use by designers of classes that inherit from the given class. This is hardly surprising, given that these two groups of users need to know different things about the class. We applied this idea to the sequential case in [9], and have tried to extend to concurrent languages in the current paper. For this purpose we had to associate with each class an abstract and a concrete synchronization property in addition to pre- and post-condition specifications of the various operations of the class. We sketched the proof techniques that would be used to show that a given implementation of a class meets both sets of its specifications. As we saw the details of the proof techniques do depend on the particular programming notation in question, but the basic approach seems general.

We wll conclude with a brief remark about future work. Kafura and Lee's [3] system depends heavily on their idea of *behavioral abstractions*. In a sense, this is a *second order* construct. In particular, commands like '`become`' have a distinct 'second-order-flavor' about them. Nevertheless, we were able to deal with their system using our approach. This is very promising, and we believe that we will be able to extend our approach to other systems that use what look like second-order constructs even more extensively. That is not to say that we could deal with arbitrary second order operations on functions, but there is no need to do that. All we need to be able to do is to handle the types of opertaions that seem useful in building OO systems. Matsuoka and Yonezawa [5] also remark on the importance of such limited second-order capabilities in building powerful systems. But it is not enough to *build* the systems. We need to be able to reason about them; that is why we believe our approach will prove useful.

5 References

1. S. Frolund, Inheritance of synchronization constraints, Proceedings of *ECOOP* '92, pp. 185-196.

2. Y. Ishikawa, Communication mechanisms on autonomous objects, OOPSLA 1992, pp. 303-314.
3. D. Kafura, K. Lee, Inheritance in Actor based languages, Proceedings of *ECOOP* '89, pp. 131-145.
4. B. Liskov, J. Wing, A behavioral notion of subtyping, ACM TOPLAS, vol. 16, pp. 1811-1841, 1994.
5. S. Matsuoka, A. Yonezawa, Analysis of inheritance anomaly in concurrent OO languages, in Research directions in concurrent OO programming, ed. Agha, Wegner, Yonezawa, 1993, pp. 107-150.
6. B. Meyer, Object oriented software construction, Prentice-Hall, 1988.
7. J. Misra, KM. Chandy, Proofs of networks of processes, IEEE TSE, 1981.
8. N. Soundarajan, Axiomatic semantics of CSP, ACM TOPLAS, 1984.
9. N. Soundarajan, S. Fridella, Inheriting and modifying behavior, submitted to ECOOP 1997 (available at URL: http//www.cis.ohio-state.edu/∼neelam).