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

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 especially when the constraints are inherited from a base class and modified in the derived class using one of the various mechanisms that have been proposed in the literature dealing with the inheritance anomaly issue. 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, 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 problems in applying OO ideas to concurrent programming is the *inheritance anomaly* [MandY]. To see the problem, consider the classic 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]

[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.

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 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. 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 [Ecoop97] to simplify this task; the key idea behind the approach in [Ecoop97] 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 formalism we develop in the current paper is a natural extension of that in [Ecoop97]. 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, and about the synchroniza-

tion constraints, i.e., what operations may be invoked at various points during the execution of the program.

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 our proof rules will have to be ...

In the fourth section we use our approach to specify and verify some simple classes. In the final section we summarize our work, and address the question of what types of ...

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.) As mentioned in the introduction, we will associate two specifications with C, an *abstract specification* for use by a client 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.[2] Second, the client must be able to tell at what points in the execution of the system the various methods of C may be invoked. 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 will accept calls 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 *syn-*

[2]We could also specify some of the functionality of the methods of C in terms of an invariant of C. We will use invariants but, as we will see shortly, for specifying synchronization properties. We should also note that our examples will be 'light' on computation and focus more heavily on synchronization aspects.

chronization invariant S_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.

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 from the sequence. Thus in terms of pre- and post-conditions the abstract specification of the class is:

$\{\ \mathit{true}\ \}$ `put( k )` $\{\ \mathit{self}_a = \#\mathit{self}_a \hat{}\ \texttt{k}\ \}$

$\{\ \mathit{true}\ \}$ `get( )` $\{\ \mathit{value} = \mathit{head}(\#\mathit{self}_a) \wedge \mathit{self}_a = \mathit{tail}(\#\mathit{self}_a)\ \}$

where self_a is the abstract bounded buffer object, *value* denotes the value returned by the operation under consideration; $\hat{}$ 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 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 (or rather *may*) be suspended, to be resumed when the appropriate synchronization constraint is satisfied.[3]

So next we need to specify the (abstract) synchronization invariant $AS_{\texttt{Bbuffer}}$ which will capture these synchronization constraints:

$$AS_{\texttt{Bbuffer}} = [\,(\mathit{self}_a \neq \langle\rangle \Rightarrow \texttt{get} \in \mathcal{A}_{\texttt{Bbuffer}}) \\ \wedge\, (|\mathit{self}| < n \Rightarrow \texttt{put} \in \mathcal{A}_{\texttt{Bbuffer}})\,]$$

where $\mathcal{A}_{\texttt{Bbuffer}}$ is the *acceptance set* of `Bbuffer`, i.e. the set of all methods of `Bbuffer` that can currently be invoked. $AS_{\texttt{Bbuffer}}$ 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 invariant, again on the data members. We will term these pre- and post-conditions and the invariant 'concrete' in order to distinguish from their abstract counterparts of the abstract specification. Thus the concrete specification of C *will* contain more information than its abstract spec-

[3]The constraint may become satisfied at a later time since another –concurrent– client that shares this buffer might invoke the `put` operation, after the execution of which the buffer will no longer be empty. In this paper we will not worry about what the client code looks like, including the question of how concurrent clients might be implemented.

ification, 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.

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

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

$\{$ *true* $\}$ `get( )` $\{$ *value* $= \#$`Elems`$[\#$`out`$]$ $\wedge$ `out` $= \#$`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`.

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 part of the concrete synchronization invariant:

$$CS_{\texttt{Bbuffer}} = [\,(\texttt{in} \neq \texttt{out} \oplus 1 \Rightarrow \texttt{put} \in \mathcal{CA}_{\texttt{Bbuffer}}) \\ \wedge\, (\texttt{out} \neq \texttt{in} \Rightarrow \texttt{gett} \in \mathcal{CA}_{\texttt{Bbuffer}})\,]$$

where we have used $\mathcal{CA}_{\texttt{Bbuffer}}$ 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,

[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 [Ecoop97] we explain how we can deal with both kinds of variables.

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.

Let us consider the second part first. For this part we must first define an abstraction function ε that maps the concrete state to the abstract state. 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. Let CS be the concrete synchronization invariant, and AS 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):

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

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

$$CS(\omega) \Rightarrow AS(\varepsilon(\omega)) \tag{2}$$

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 [Ecoop97]; for instance if we were to include a *functional* invariant for the class (in addition to the synchronization invariant), that would also have to appear in these rules. Since our interest in this paper is in synchronization issues, we ignore these complications.[5]

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 code for the entire class meets $C\mathcal{S}$, the concrete synchronization invariant. Note that so far we have been able to ignore the programming language notation in

[5]It should be noted though that except for very simple classes, functional invariants are necessary in order to completely specify class behavior.

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 [for example, 7, 11, 10] for specifying the synchronization conditions, may be handled.[6] We will consider several of these notations in turn.

A natural approach to expressing synchronization code in classes would be the approach of *method guards* as proposed in Frølund [7].

One possible approach to expressing synchronization code would be for the programming language to provide direct access to the acceptance set, or something very similar to it, and provide primitives for its direct manipulation in the bodies of the various methods. This is the approach, for instance, of Kafura and Lee [11]; their *behavioral abstraction* is essentially 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 –not necessarily, not even usually, disjoint from each other– of the set of all methods, and names these subsets with distinct names, say $B_1, \ldots, B_m$. 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, $B'_1, \ldots, B'_m$

In order to verify the properties of classes constructed using such behavioral abstractions, we need t

Frølund [7] proposes the idea of

[6] 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.