# Inheriting and Modifying Behavior

Neelam Soundarajan and Stephen Fridella
Computer and Information Science
The Ohio State University
Columbus, OH 43210
e-mail: {neelam,fridella}@cis.ohio-state.edu

November 27, 1996

**Abstract**

In many languages, the mechanism of protected members is critical to making inheritance work. It allows access to some of the internals of a base class $B$ to the designer of a derived class $D$, while denying it to clients of $B$. We develop an axiomatic approach in which class behavior is specified in terms of an abstract specification AND a concrete specification, the former for use by a client, the latter containing information about how the protected members change, for use by a derived class designer. We present proof rules that allow us to establish that a class meets both specifications.

## 1 Introduction

Much of the power of the object-oriented approach derives from the key notion of *inheritance*. Inheritance allows us to conveniently define new classes that extend, or that differ slightly from, existing classes. Naturally it is important to develop appropriate formalisms that can be used to specify and reason about the *behavior* of classes defined by inheritance from other classes. Indeed, several such formalisms have been developed, some notable ones being those of Liskov and Wing[8, 7], Leavens[6], Meyer[9], etc.[1]

In standard OO languages like *C++*, the mechanism of *protected members* of a given class plays a critical role in making inheritance work. It is this mechanism that gives access to certain aspects of the internals of the base class $B$ to the designer of a derived class $D$, while at the same time denying this access to *clients* of $B$. Once he has access to the protected members of $B$, the designer of $D$ will be able to implement the changes, and *only the changes*, necessary to obtain a class that exhibits behavior that extends or is appropriately different from that of $B$. Not having protected members in a class often leads to serious inefficiencies in the implementation of the derived class since any type of access to any of the (data) members defined in $B$ would have to be achieved via (often a long sequence of) calls to the public member functions of $B$; we will see some simple and natural examples of this in the next section.

Despite the obvious importance of protected members, most formalisms seem to have paid little attention to them. Most of the work has instead focussed on the question of behavioral subtyping, i.e., given two classes and their (abstract) specifications, what conditions must they satisfy in order for us to conclude that one is a subtype of the other?[2] This is obviously an important question but answering it doesn't help us in obtaining

[1] It is worth stressing that our interest is in *behavioral* questions. A number of interesting models corresponding to covariance and other flexible forms of inheritance and the relations between the resulting subclasses and the base classes have been explored by a number of authors[2,3]. Our goal, on the other hand, is to consider a simple kind of inheritance, and develop a formalism that can be used for *reasoning, in an axiomatic style, about the behaviors of the resulting classes.*

[2] Given our interest in behavioral questions, we will, in the rest of the paper, use the term 'subtyping' to mean 'behavioral subtyping'.

the specification or verifying the correctness of (the implementation of) a derived class $D$ that inherits from a base class $B$. Indeed, formalisms such as those of [6, 8, 7] can be used to establish the subtype relation between two classes that may have been implemented completely independently of each other, rather than by using inheritance, so these are really orthogonal issues. Our goal in the current paper is to develop ways of specifying classes that will provide appropriate information to both clients of the class, as well as to designers of classes that inherit from the given class; and ways of establishing the correctness of the implementation of the derived classes with respect to their specifications.

Each class $B$, in our formalism, has two specifications, one for use by a client of the class, another for use in establishing the correctness of a class $D$ that inherits from $B$ and has direct access to the protected members of $B$. Protected members have often been criticized because they 'break the encapsulation' by exposing the internals of a class. This is true in a sense, but as we said earlier, protected members make possible much code reuse that would otherwise have to be re-implemented. Moreover, it is only the designer of the derived class $D$ that knows about the protected members of the base class $B$, not clients of $B$. This, as we will see in section 3, will be perfectly reflected in the two sets of specifications we will associate with each class. It is also worth noting that any information that the designer of $B$ feels should be inaccessible even to derived classes can always be defined as being *private* to $B$; this will also be reflected in our specifications.

Once we have verified the correctness of a derived class $D$ using the formalism developed in this paper, we can, if we so choose, use say the formalism of [8], to show that this class is a (behavioral) subtype of the class $B$. This will not be always appropriate since not every derived class will be a subtype of its base class. This is the other side of the orthogonality between inheritance and subtyping. Thus we have three possible cases:

1. Classes $B$ and $D$ are implemented independently of each other and we show that $D$ is a subtype of $B$ using, say, [6].
2. $D$ is implemented by using inheritance from $B$ and there is no subtyping relation. In this case, we would follow the approach developed in the current paper and use the two sets of specifications of $B$ to establish that the implementation of $D$ meets its (two sets of) specifications.
3. $D$ is implemented using inheritance from $B$, and is a subtype of $B$. In this case, we would first proceed as in step (2), then as in step (1).

We should note here that there is one situation where a function $f$ of $B$ is not overridden in $D$ but another function $g$ (also of $B$) that $f$ *calls*, *is* overridden, that is not handled by the system we present in this paper. We will return to this question in the final section of the paper.

The rest of the paper is organized as follows: In the next section we provide a number of examples that show that protected members are essential both from the point of view of efficiency and from that of reuse. In section 3 we present our formalism; 3.1 presents the specification notation, and 3.2 presents the verification system. In section 4 we use the system of section 3 to specify the behavior of, and verify the correctness of the implementation of, a *priority queue* as a derived class of a *queue*. In the final section, we reiterate the need for protected members, and the resulting need for formalisms for specifying and verifying classes built using this mechanism. We summarize our approach, compare it to others, and point out that our approach does not, somewhat surprisingly, require any additional work on the part of the base class designer than would be needed if we didn't allow protected members.

## 2 Protected Members and Reuse

The usage of protected members in classes has often been criticized (see, for example, Horstmann [5]) because they allow us to break the encapsulation provided by the class. But, in fact, as the examples in this section will show

- Efficiency as well as reuse considerations dictate the usage of protected members in several very natural situations; and this is true even when the derived class being defined is a subtype of the base class.

In order to ensure that such usages are *safe*, it is then necessary, in view of the potential breakage of encapsulation, to have a specification notation to express appropriate properties of such classes to enable designers of derived classes to express the behavior they expect their classes to exhibit; and a verification procedure that can use to establish that their classes do indeed meet their specifications. We will present both of these in section 3 and apply them to a simple but non-trivial example in section 4.

Our first example illustrates the need for protected members on the basis of efficiency considerations in the implementation of the derived class. Suppose a class library has the following declaration of a *stack* class:

```
class Stack
public:
    Stack();
    (* constructor *)
    int Pop();
    (* returns top element of stack *)
    void Push(int item);
    (* places item on the top of the stack *)
    int IsEmpty();
    (* returns true if stack has no items *)
private:
    int top;
    int items[100];
```

Suppose we wish to implement (using inheritance) a *readable stack* that has an extra operation Read that takes as a parameter an integer $i$ and returns the stack element which is $i$ elements from the *bottom* of the stack. How would such an operation be implemented? Given the Stack class above, one possible reasonable algorithm would have two phases: first Pop each item off of the stack and Push them in the opposite order onto a temporary stack until you reach the bottom of the stack; next repeat the same process in the opposite direction, pausing before the $i$th element has been restored to the original stack to save its value so that it may be returned as the function result when the stack has been completely restored. The efficiency of this operation could be improved greatly if the private members of Stack had instead been protected members:

```
class Stack
public:
    ... (* as above *)
protected:
    int top;        (* now accessible to derived classes *)
    int items[100];
```

Now the implementation of Read in the derived class Readable_Stack will be a *single array reference*. It is true that with this approach, if the representation in the base class were changed from an array to a linked list, we would be forced to reimplement the Readable_Stack class. However, the efficiency gains from using protected members are so large that the designer must have the option of using them.

But even if efficiency were not an issue, it turns out that protected members are essential. In the following example we wish to implement a behavioral subtype of an existing class, but in the absence of protected members we will be forced to implement it from *scratch*; there cannot be any reuse of the work done in the implementation of the base class. The example is taken from [8]; Liskov and Wing discuss the subtype-supertype relation between the *set* class and the *bag* class but do not present an implementation.

The *bag* class is defined as follows:

```
class Bag
public:
    Bag();
    (* constructor *)
    int Remove();
    (* returns a single element from the bag *)
    void Add(int item);
    (* places a single element within the bag *)
    int IsEmpty();
    (* returns true if the bag has no elements *)
private:
    (* private implementation details of bag *)
```

A *bag* is essentially a multiset of elements. Single elements can be added to the *bag*, using Add, and they are stored in no particular order. The *bag* can be asked to return one of the elements it contains and remove it from the *bag*. It may pick *any* of the elements currently in the *bag*.

Suppose a class designer wishes to implement a *stack* as a derived class of Bag. (We will assume for the purposes of this example that the usual stack operation Push will instead be named Add and the operation Pop will be named Remove). This is a reasonable idea since a *stack is-a bag*, i.e., as [8] points out, *stack* is a (behavioral) subtype of *bag*. All that is needed to achieve this implementation is to override the Remove and Add operations in the derived class. Instead of choosing an *arbitrary* element to return, Remove should choose to return the *most recently added* element. Presumably Add will also need to be overridden so that it stores the item in such a way that it can be identified by the new Remove operation. How will these changes be implemented in the derived class?

The answer is that, given that the representation is private to the *bag*, not only will Add and Remove have to be completely rewritten, but *new* data members must be added to store the items that make up the stack. Why is this necessary, when the derived class Stack will already contain all the data members of Bag? Because the only access the operations of Stack have to the *private* members of Bag is through its public operations Add and Remove. And the behaviors of these operations are not sufficiently deterministic to be of any use in implementing Stack. In short, implementing Stack as a derived class of Bag is exactly as difficult as implementing Stack as a completely independent class.

On the other hand, if the private members of Bag were made protected, these members can be reused in the implementation of Stack. For instance, if Bag is implemented as a linked-list structure then Stack can reuse the pointers to the head of the list, the tail of the list, and possibly some protected member functions which operate on the list. In fact, there is a chance that we might not even have to reimplement Add and Remove! This would be the case if these operations, as implemented in Bag already *happen to* return the right element (from the point of view of Stack), even if their abstract description does not say so. Other implementations may allow more or less reuse of code and data members in the derived class, but the point is without protected members *no* reuse is possible.

For our final example, let us consider a more complicated case that might arise in the real world of software development. Suppose there is already an existing class, perhaps purchased from a software vendor, called Expression which supports three operations: Parse, Dump, and Evaluate. Parse takes a string containing an expression as a parameter, parses it, and stores the tree in a private data member. Dump returns the contents of the parse tree as a string. Evaluate returns the value of the parsed expression. Suppose we wish to create a derived class called Expression_Operand. This class supports the same interface as Expression with one extra operation, Binary_Op. This operation takes two parameters: a single binary operator and another Expression_Operand object. The effect of invoking the operation on an expression $E$, with binary operator $\diamond$ and expression $F$ as parameters is to replace $E$ with $E \diamond F$. Since the parse tree of an expression is hidden within the Expression class, an implementation of this operation will be forced to call Dump for each of the expressions, create the appropriate new string and re-parse the result. However, if the data members which represent the parse tree are made protected in the base class, the implementation of Binary_Op can in one step combine the two expression subtrees together to form the new expression tree. No conversion from the parse tree to the string would be necessary.

There is certainly a tradeoff whenever protected members are used. While derived classes can be implemented more efficiently and reuse more code, the encapsulation of the base class is broken. However, note that the breakage is only from the point of view of the derived classes, not from that of the client. Therefore no couplings between the implementations of components will be introduced except between the base class and its derived classes. While minimizing such couplings is generally a good thing, it is likely that the derived classes and the base class are a group of closely related classes that will likely be maintained together in any case, so this kind of coupling may be acceptable in view of the advantages we have seen.

A more serious result of the encapsulation breakage is that *class invariants* may be violated in the derived classes. If a Stack class has a Size operation defined for it, a simple implementation scheme would be to keep an extra data member called size to record the current size of the stack. Whenever a Push is executed size is incremented and whenever a Pop is executed it is decremented. The Size operation would simply return the value of size. The invariant for this class is that the number of elements in the stack is equal to the value of size. If this invariant is violated by any of the operations of Stack then the Size operation will no longer work. It is easy for the implementor of Stack to preserve this invariant, and if there are no protected members in the class, we can rest assured that the invariant will not be violated by any routines defined in derived classes. Once the encapsulation of Stack is broken however, perhaps by making size protected, this assurance is no longer available. If a derived class programmer were to mistakenly change the value of size the invariant will be violated and the Size operation will no longer work correctly for instances of that derived class. Thus it is essential, in the presence of protected data, to have a way to precisely specify the expected behaviors of various classes, and mechanisms for validating these expectations. (In the example above, one aspect to be verified would be the requirement that size has the appropriate value.)

In summary, one of the principal claims of the object-oriented paradigm is that it promotes code reuse through inheritance. The examples we have presented illustrate that disallowing protected members undermines this claim. But at the same time protected members can easily lead to problems. The formalism we present in the next section will allow us to avoid these problems.

# 3 Specification and Verification of Classes

In 3.1 we introduce our specification notation for specifying the behavior of classes, including those of derived classes. In 3.2 we present our verification procedure to establish that a given class meets its specifications.

## 3.1 Abstract and Concrete Specifications

Consider a class $B$. In a language like *C++*, $B$ will have three kinds of members: *public, protected,* and *private*. Following widely accepted principles of OO design, we will assume that all the public members are *methods*, i.e., there are no public data members. Further, in order to simplify the presentation we will assume that there are no private members. Towards the end of 3.2 we will briefly indicate how we can deal with private members.

The specification $\langle \mathcal{A}, \mathcal{C} \rangle$ of $B$ will consist of a component $\mathcal{A}$, the *abstract specification* of $B$, for use by clients of $B$; and a component $\mathcal{C}$, the *concrete specification* of $B$, for use by (designers of) derived classes. The abstract specification $\mathcal{A}$ will be the usual *ADT*-type specification, in terms of a conceptual, mathematical model of $B$, with the individual methods of $B$ being specified as pre- and post-conditions in this conceptual model. In formalisms like that of [8], $\mathcal{A}$ would essentially be the *complete* specification of $B$, and indeed we will borrow notation for this part of our specification from [8]. We can also include an invariant in $\mathcal{A}$ although for simplicity we will not do so.

What information should $\mathcal{C}$ contain? The key component of $\mathcal{C}$ will also be a specification of the methods of $B$ using pre- and post-conditions but there are some important differences with $\mathcal{A}$. First, the designer of a derived class of $B$ needs to know what effect a call to one of these methods will have on the *protected members* of $B$. So the pre- and post-conditions will be in terms of the actual data members of $B$ (which, by the assumptions of the first paragraph of this section, are all protected members). In addition, $\mathcal{C}$ will contain an *invariant* over the data members of $B$ that will be satisfied prior to and at the end of execution of each of the methods of $B$. Strictly speaking the invariant does not have to be included in $\mathcal{C}$ since it is needed only when establishing that the bodies of the methods of $B$ meet their specifications. The derived class will have its own invariant which the designer of the derived class will have to identify, and in general there is no necessary relation between the invariant of the base class and that of the derived class. In practice though, the invariant of the base class often contains information useful to the derived class designer, and the derived class invariant is usually an extension of the base class invariant – the additional clauses serving to specify conditions on the new data members introduced in the derived class.

So the specification of a class will consist of a number of items. First the conceptual model of the class which can be given, for instance, in terms of a Larch-style [4] *trait*, although in order to keep the presentation simple, we will omit the formal details. Next we need an *abstraction or encapsulation function* $\varepsilon$ that maps from the concrete domain (of actual data members) to the conceptual model. Thus $\varepsilon(\sigma)$ is the 'abstract state', or the client's view, corresponding to the 'concrete state' $\sigma$. The concrete state is essentially the values of all the data members of the class, including any protected data inherited from the base class of the class being specified.

Next we must specify the abstract pre- and post-conditions corresponding to the various methods of the class. The pre-condition will be an assertion over the abstract state of the class. The post-condition will be an assertion over the abstract state that exists when the method in question starts execution and the state that exists when the method finishes execution rather than just over the state when the method finishes. This form of the post-condition makes it easier to specify interesting behaviors; and in the case of the concrete post-conditions, it makes part of the verification task easier. Next we need to specify the invariant for the class, which will be an assertion over the concrete state. Finally, we need to specify the concrete pre- and post-conditions for the methods; the pre-condition will be an assertion over the concrete initial state and the post-condition an assertion over the concrete initial and final states.

Consider a base class $B$ and a derived class $D$. $D$ will inherit many of the methods from $B$ and not change them. *The concrete specifications of these methods will be identical in the two classes. This is the* *specification analog of code reuse: if the designer does not have to reimplement certain methods, he also does not have to provide new concrete specifications for them.* The designer must, of course, provide concrete specifications for any methods that he redefines, as well as any new methods. He must also provide the invariant for $D$ although this will usually be nearly identical to the invariant of $B$. And the designer must provide a new conceptual model and the abstract pre- and post-conditions for all the methods of this class, including the ones inherited unchanged from $B$. In principle the model for $D$ could be very different from that for $B$ as could the abstract pre- and post-conditions for the methods, but as in the case of the invariant, they are usually closely related.

### 3.2 Verification of Class Behavior

In order to establish that a class meets its specifications, we need to verify the following:

1. The actual implementations (method bodies) of the various methods of the class meet their concrete pre- and post-conditions.
2. The individual methods leave the invariant satisfied when they complete execution.
3. If the requirements in the concrete specification are met, then the requirements in the abstract specification will be met.

Consider a class $D$. Consider the first requirement above. Suppose $f$ is a function defined in $D$ with concrete pre-condition $c.pre_f(\omega)$ and concrete post-condition $c.post_f(\omega, \omega')$ where $\omega$ and $\omega'$ denote the initial and final (concrete) state before $f$ starts and when it finishes. We just need to establish, using standard axioms and rules, that the body $f.b$ of $f$ meets the condition

$$\{c.pre_f(\omega)\}\ f.b\ \{c.post_f(\omega, \omega')\} \tag{1}$$

If $f$ is *inherited* from the base class $B$ of $D$ rather than being defined in $D$, then this step is not performed, since the correctness of $f$ (with respect to its concrete pre- and post-conditions) has already been established. *This is the verification analog of code reuse: if $f$ is not being reimplemented, then there is no need to reverify it.* This is particularly important since the designer of $D$ may not even *have* access to the source code of $f$; this would be the situation, for instance, if $B$ was obtained from a software vendor who only provided the header files (containing information about public and protected members) and the object code but not the source code for the various methods of $B$.

Next suppose $I$ is the invariant of the class $D$. To meet the second requirement, we would have to establish:

$$I(\omega) \wedge c.pre_f(\omega) \wedge c.post_f(\omega, \omega') \Rightarrow I(\omega') \tag{2a}$$

This ensures that if when $f$ starts its pre-condition and the invariant are satisfied, then when it finishes the invariant will be satisfied. But this works only if $f$ is a method defined in $D$. If $f$ was inherited from the base class then the post-condition of $f$ will be in terms of the concrete state of the base class, whereas $I$ will in general also involve the data members introduced in the current class.

To see how to meet requirement (2) in this case, note that the state $\omega$ consists of two components, $\sigma$ being the component inherited from the base class $B$, and $\tau$ the new data members defined in the derived class $D$. We will use the notation $\omega = \sigma \oplus \tau$ to denote this. $c.pre_f$ and $c.post_f$ will only involve $\sigma$ whereas $I$, being the invariant for the derived class, will also involve $\tau$. But even though $c.post_f$ doesn't involve $\tau$, we can be sure that $f$ will not modify this part of the state since $f$ does not have access to it. Thus we can show that the invariant $I$ will hold when $f$ finishes by establishing:

$$[I(\sigma \oplus \tau) \wedge c.pre_f(\sigma) \wedge c.post_f(\sigma, \sigma')] \Rightarrow I(\sigma' \oplus \tau) \tag{2b}$$

When $f$ finishes, the $\tau$ component of the state is the same as it was when $f$ started; so the complete final state is $\sigma' \oplus \tau$, and (2b) requires us to show that this state satisfies $I$, thereby establishing $I$ when $f$ finishes.

Finally we need to establish that the abstract specifications of $D$ will be met. Let $\varepsilon$ be the abstraction function for $D$ that maps the concrete state to $D$'s conceptual model. Let $a.pre_f$ and $a.post_f$ be the abstract pre- and post-conditions of $f$. We need to establish the following:

$$[a.pre_f(\varepsilon(\omega)) \wedge I(\omega)] \Rightarrow c.pre_f(\omega) \tag{3a}$$

$$[I(\omega) \wedge c.pre_f(\omega) \wedge c.post_f(\omega, \omega')] \Rightarrow a.post_f(\varepsilon(\omega), \varepsilon(\omega')) \tag{3b}$$

If $f$ was inherited from the base class, $(3a)$ and $(3b)$ should be handled in the same way as we did with the invariant in $(2b)$. Thus $(3a)$ should be read as:

$$[a.pre_f(\varepsilon(\sigma \oplus \tau)) \wedge I(\sigma \oplus \tau)] \Rightarrow c.pre_f(\sigma)$$

and $(3b)$ as:

$$[I(\sigma \oplus \tau) \wedge c.pre_f(\sigma) \wedge c.post_f(\sigma, \sigma')] \Rightarrow a.post_f(\varepsilon(\sigma \oplus \tau), \varepsilon(\sigma' \oplus \tau))$$

It is perhaps in these final steps $(3a)$ and $(3b)$ that the difference between our formalism for dealing with inheritance and others such as [8] that deal with behavioral subtyping shows up most prominently. In our formalism, the derived class designer works with his own conceptual model, and abstract pre- and post-conditions. Even if a function $f$ is inherited unchanged from the base class, he must relate the *concrete* specification of $f$ to its abstract specification in the derived class. By contrast, formalisms for subtyping focus on the relation between *abstract* models of the supertype and subtype.

Before concluding this section, we should note two points briefly. First, constructor functions are somewhat different from other functions since there is no object in existence when a constructor function is invoked. Thus when we apply our various rules to these functions, the invariant on the initial state should be omitted.

The other point is how we deal with *private* members. Although the derived class has no access to the private members of the base class, they cannot be ignored altogether. The reason is that the effect of the member functions of the base class cannot be understood (or specified) completely unless we somehow refer to their effect on the private members. For instance, the base class might store, in its private members, some information about the history of the calls to one of its functions and use this information to modify what later calls to the same function do. But at the same time, we do not want the derived class designer to think in terms of the details of the private members of the base class since these details are not relevant to the derived class. The solution is for the derived class to take an *abstract* view of the private members of its base class. In other words, the 'concrete' specification of the various members of the class $B$ will be in terms of pre and post-conditions on states of the form $\sigma_{prot} \oplus \delta(\sigma_{priv})$, i.e., states consisting of the protected component and an abstract version of the private component, $\delta$ being the abstraction function. We will not go into further details here.

# 4 Examples

In this section we will present specifications of a base class and a derived class, and verify that the implementation of the derived class is correct. The base class is a *queue*. A class designer wishes to implement a *priority queue* as a derived class. The definition for the base class is given in Fig. 1. The abstract specification is included in the public section of the class and the concrete specification is included in the protected section. This seems like a reasonable convention since the concrete specification is of interest only to the derived class designer. In each of our post-conditions $post_f(\omega, \omega')$, we use the notation *#var* to refer to the value of *var* in the starting state $\omega$, and simply *var* to refer to the value of *var* in the ending state $\omega'$. (Our specifications are precise but to make the presentation simpler, somewhat informal.)

The abstract conceptual model of a queue is a sequence of items, expressed as $< i_1, i_2, ... >$. **** is the sequence concatenation operator; *length(seq)* is the length of *seq*. The implementation uses a linked list to store the items in the queue, and a data member size to record the number of elements in the queue. head is the first node in the list, tail the last node; if (and only if) the list is empty, head will be equal to tail. The concrete pre- and post-conditions and the invariant *Invq* are written in terms of these protected data members. The concrete post-condition for the Size operation is particularly simple since that routine just returns the value of the data member size. The notation $list(node, node_1)$ refers to the linked list of nodes with head *node* and tail $node_1$. Of course the assumption here is that the value of size accurately reflects the number of items in the queue at all times. This is precisely the concrete invariant *Invq*. The function *Numlist(node,* $node_1$*)* returns the number of nodes in the linked list whose head is *node* and whose tail is $node_1$ (not counting the first element). The second clause in the concrete post-condition of *Enqueue* deserves a word of explanation; it says that the *final* value of the *next* component of the node that was pointed to by the *initial* value of *tail* is equal to the final value of *tail*, i.e., the *next* component of the originally last element of the list now points to the new node.

The definition for the derived class Priority_queue appears in Fig. 2. Note that the abstract conceptual model has changed. Instead of being modeled as a sequence of items a Priority_queue is modeled as a multiset of items. This change reflects the fact that in the derived class the order that items are inserted into the queue is no longer important, since they will be retrieved based solely on their priority. We could have reused the conceptual model of the base class, but in that case we would have had to add an invariant to the abstract specification stating that the items are stored in the sequence in order of their priorities. We chose the alternative of using a different conceptual model. We also assume that the conceptual value of an Item includes a component that represents its priority.

The only operation from the base class which is overridden is Enqueue. Instead of adding the new item to the tail of the linked list, the new implementation of Enqueue traverses the linked list and inserts the item in order of its priority. For brevity we have omitted the actual implementation of this operation and will therefore skip the verification of its concrete pre- and post-conditions. The invariant *Invpq* of this class then consists of two clauses; the first stating, as before, the relation between the size and the number of items in the list; the second that these items are in decreasing order of priorities. The notation $\forall nodes\, m\, in\, list(node, node_1)$ quantifies over all of the nodes in the list except the first (dummy) node. Note that although the other operations are not overridden, it is our responsibility, in verifying the correctness of the derived class implementation, to show that these operations will leave the new invariant unchanged. This is the implication ($2b$) of section 3 that we are required to check.

We will now verify that the implementation of Priority_queue meets its specifications. Since we have assumed that the concrete pre- and post-conditions of Enqueue have been verified, and since each of the other operations has been inherited from the base class, requirement (1) from section 3.2 is satisfied. Our next obligation is to show that the individual methods preserve the invariant *Invpq*. In the following we will refer to the first conjunct of the invariant, which asserts that the value of size must match the number of items in the list, as $I_1$, and the second conjunct, which requires that the items in the linked list are maintained in order of their priority, as $I_2$. For Enqueue which is defined in this class, we must establish (2a). Specifically, we must show class Queue;

public:

*Conceptual model of a* Queue *is a sequence* $< i_1, \ldots, i_n >$

Queue();
*requires: true*
*ensures: self =* <>

void Enqueue(Item i);
*requires: true*
*ensures: self = #self ** <i>*

Item Dequeue();
*requires: length(self) > 0*
*ensures: returns i* $\wedge$ *#self = <i> ** self*

int Size();
*requires: true*
*ensures: returns length(self)* $\wedge$ *self = #self*

protected:

Node head;
Node tail;
int size;

*Concrete Specification*

*Abs. fn.: map* (head,tail,size) *to sequence of data items in list(head,tail)*

*Invq: Numlist(head, tail) = size*

*Queue()*
*req: true*
*ens: head = tail* $\wedge$ *tail.next = nil* $\wedge$ *size = 0*

*void Enqueue(Item i)*
*req: true*
*ens: tail = new(node)* $\wedge$ *(#tail).next = tail* $\wedge$ *tail.next = nil* $\wedge$
*tail.data = i* $\wedge$ *size = #size + 1*

*Item Dequeue()*
*req: head* <> *tail*
*ens: head = #head.next* $\wedge$ *return value is head.data* $\wedge$ *size = #size - 1*

*int Size()*
*req: true*
*ens: returns size* $\wedge$ *size = #size* $\wedge$ *list(#head, #tail) = list(head, tail)*

Figure 1: Definition and Specification of Queue

class Priority_queue;

public:

*Conceptual model: A* Priority_queue *is a multiset of items.*

Queue();
*requires: true*
*ensures: self* $= \emptyset$

void Enqueue(Item i);
*requires: true*
*ensures:* $self = \#self \cup i$

Item Dequeue();
*requires:* $|self| > 0$
*ensures: returns i* $\wedge\, i \in self \wedge$
$\forall j \in self, i.priority \geq j.priority$

int Size();
*requires: true*
*ensures: returns* $|self| \wedge self = \#self$

protected:

Node head;
Node tail;
int size;

*Concrete Specification*

*Abs. fn: Defined in the discussion.*
*Invpq: Numlist(head, tail) = size* $\wedge$
$\forall$ *nodes m in list(head, tail), (m.next* $\neq$ *nil)* $\Rightarrow$ *(m.data.priority* $\geq$ *m.next.data.priority)*

*Queue()*
*inherited from base class Queue*

*void Enqueue(Item i)*
*req: Invpq*
*ens: n = new(node)* $\wedge$ *n.data = i* $\wedge$
*n is inserted into list(#head,#tail)* $\wedge$
$\forall$ *nodes m in list(head, n), n.data.priority* $\leq$ *m.data.priority* $\wedge$
$\forall$ *nodes m in list(n, tail), n.data.priority* $\geq$ *m.data.priority* $\wedge$
*size = #size + 1*

*Item Dequeue()*
*inherited from base class Queue*

*int Size()*
*inherited from base class Queue*

Figure 2: Definition and Specification of Priority_queue

$$I_1(\omega) \land I_2(\omega) \land c.post_{Enqueue}(\omega, \omega') \Rightarrow I_1(\omega') \land I_2(\omega') \qquad (2a_{Enqueue})$$

Here is an informal proof:

1. Assume $I_1(\omega)$ , $I_2(\omega)$, and $c.post_{Enqueue}(\omega, \omega')$
2. $I_1(\omega)$ asserts that the size of the list in state $\omega$ is equal to $\omega(size)$.
3. $c.post_{Enqueue}(\omega, \omega')$ asserts that $\omega'(size) = (\omega(size) + 1)$ and the list is state $\omega'$ has one more element than the list in state $\omega$.
4. (2) and (3) together imply $Numlist(\omega'(head), \omega'(tail)) = \omega'(size)$.
5. $c.post_{Enqueue}(\omega, \omega')$ asserts that the newly added node is in its proper sorted position in the list in state $\omega'$.
6. $I_2(\omega)$ asserts that the other elements in the list are in proper sorted order in state $\omega$.
7. Since inserting an item into a linked list does not change the relative order of the items already present, (5) and (6) imply that the list in state $\omega'$ is in sorted order.
8. (4) is $I_1(\omega')$ and (7) is $I_2(\omega')$. □

The other operations were inherited from the base class so we will need to establish (2b) for each of them. However, no new data members were added in the Priority_Queue class so the $\tau$ component of the concrete state is $\emptyset$. So in this particular case (2b) is the same requirement as (2a). For the constructor, we note that it establishes the object with an empty linked list and with size equal to 0 and therefore satisfies the invariant. For the Size operation there is no pre-condition. Furthermore the post-condition assures us that $\omega = \omega'$. Since

$$I(\omega) \land (\omega = \omega') \Rightarrow I(\omega')$$

we know that Size preserves the invariant. Finally we turn our attention to Dequeue.

We must show

$$Invpq(\omega) \land c.pre_{Dequeue}(\omega) \land c.post_{Dequeue}(\omega, \omega') \Rightarrow Invpq(\omega') \qquad (2b_{Dequeue})$$

Again, our proof will be informal.

1. Assume $Invpq(\omega)$ , $c.pre_{Dequeue}(\omega)$, and $c.post_{Dequeue}(\omega, \omega')$
2. Moving the head of the linked list in state $\omega$ to the next node in the list in state $\omega'$ decreases the size of the list by one. (i.e. $\omega(size) - 1 = \omega'(size)$ )
3. $Invpq(\omega)$ asserts that the value of $\omega(size)$ is equal to the number of items in the linked list in state $\omega$.
4. $c.pre_{Dequeue}(\omega)$ guarantees that the linked list will not be empty.
5. $c.post_{Dequeue}(\omega, \omega')$ asserts that the head of the linked list is moved forward one item from state $\omega$ to state $\omega'$, and that the value of $size$ is decremented.
6. (2), (3), (4), and (5) together imply that the $size$ has the correct value in state $\omega'$.
7. If the items of the linked list are sorted in descending order is state $\omega$, then removing the first element still leaves the remaining items sorted in descending order in state $\omega'$.
8. (7) and $c.post_{Dequeue}(\omega, \omega')$ together imply that the items in the linked list in state $\omega'$ are sorted in descending order by priority.

9. (6) and (8) imply $Invpq(\omega')$ □

All that remains is for us to establish that the concrete pre- and post-conditions of the operations of Priority_queue, together with the class invaraint and sufficient to ensure that the abstract specification of Priority_queue is met. To do this we must show that $(3a)$ and $(3b)$ hold for each of the operations of Priority_queue.

The concrete pre-conditions of the constructor and Size are both *true*. Therefore $(3a)$ holds for these operations trivially. For Enqueue the concrete pre-condition is the invariant $Invpq$. Since $Invpq$ obviously implies itself, $(3a)$ again holds trivially.

To verify $(3a)$ for Dequeue we need to relate the concrete instances of Priority_queue to the abstract instances of Priority_queue. Let us first define the abstraction function $\varepsilon$ which maps concrete states to multisets:

$\varepsilon(\omega) = makeset(list(\omega(head), \omega(tail)))$, where

$$makeset(list(node_1, node_2)) = \begin{cases} \emptyset & \text{if } node_1 = node_2 \\ node_1.next.data \cup makeset(list(node_1.next, node_2)) & \text{otherwise} \end{cases}$$

*makeset* simply collapses the elements of a linked list into a multiset. Therefore, for any concrete state $\omega$, the corresponding abstract state $\varepsilon(\omega)$ is the multiset of all of the items in the linked list in state $\omega$. The abstract pre-condition of Dequeue states that the size of the multiset $\varepsilon(\omega)$ is greater than zero. This could only be true if $\omega(head) \neq \omega(tail)$, which is exactly the concrete pre-condition of Dequeue.

To verify $(3b)$ for the constructor we first observe that

$(\omega(head) = \omega(tail)) \Rightarrow \varepsilon(\omega) = \emptyset$

The concrete post-condition of the constructor $c.post_{Queue}(\omega, \omega')$, asserts that $\omega'(head) = \omega'(tail)$ so from the above observation we can conclude that

$c.post_{Queue}(\omega, \omega') \Rightarrow a.post_{Queue}(\varepsilon(\omega), \varepsilon(\omega'))$

The concrete post-condition of Enqueue asserts, among other things, that a new node with a data value of $i$ is inserted into the linked list. This effectively adds i to the multiset of the elements in the linked list. As a result, the concrete post-condition of Enqueue directly implies the abstract post-condition.

The concrete post-condition of Size asserts that the concrete state of the object does not change when the operation is invoked. Thus the second conjunct of the abstract post-condition is implied immediately. As for the first conjunct, to prove this we need the first conjunct of $Invpq$ which states that $Numlist(\omega(head), \omega(tail)) = \omega(size)$. Looking at the definition of $\varepsilon$ it is clear that

$Numlist(\omega(head), \omega(tail)) = |\varepsilon(\omega)|$

Given these two facts, it is clear that Size returns the proper value.

Finally we must verify (3b) for Dequeue. Here we must rely heavily on $Invpq$. The concrete post-condition of Dequeue asserts that the first item from the linked list is returned. $Invpq$ asserts that this first item must necessarily have the highest priority of any item in the list. Since we know from the definition of $\varepsilon$ that

*an item i is in the linked list in state* $\omega \Leftrightarrow i \in \varepsilon(\omega)$

It follows that the item returned is the highest priority item in the Priority_queue.

This concludes our verification that the specification of class Priority_queue holds.

## 5 Discussion

Inheritance, combined with the mechanism of protected members, is a powerful tool. It allows a designer to design new classes that extend the functionality (by providing completely new functions), or that alter the behavior (by redefining some functions), of existing classes without having to redo the work that has been done by the designer of the base class. But in order to effectively and reliably use this tool, the designer must have appropriate and precise information about the base class. In particular, the *abstract* view of the base class $B$ that is sufficient for a *client* of $B$ is not sufficient for the designer of a derived class $D$. *He* needs to know what effect the various functions of $B$ will have on the protected members of $B$ (as well as on an abstract model of the private members of $B$). This was why we introduced in our approach the idea of two sets of specifications, the usual *abstract* one for use by the client, and the *concrete* specification for use by designers of derived classes. Armed with the information in the concrete specification of the base class, the designer of the derived class is able to build his class appropriately. The class $D$ that he designs will also, of course, have an abstract specification for use by *its* clients and a concrete specification for use by designers who wish to build further on $D$. And our rules (1, $2a$, $2b$, $3a$, $3b$) specify the conditions that the derived class designer must check to ensure that $D$ meets its specifications assuming that $B$, in turn, meets its specifications. (Of course, some designers may prefer to just use the specification notation and do the verification informally.)

Interestingly, our system doesn't really require any additional work on the part of the designers of the base class! Suppose, for instance, that we disallow protected members. It would then seem because the base class would need only an abstract specification (since not even the designer of a derived class would have access to any internals of the base class), that all the work in coming up with the concrete specification and in checking that it is consistent with the abstract specification (rules $3a$, $3b$) can be avoided; but neither of these is in fact true. This is because the bodies of the operations of the class manipulate the actual (protected or private) data members of the class and not some conceptual model. So the first step in showing that the class meets its abstract specification will have to be to write down the concrete pre- and post-conditions for the various member functions, and show that the bodies of these functions are correct with respect to these concrete specifications. Once that step is complete, we will then have to show, using some abstraction function (our $\varepsilon$), that the functions also meet the abstract specifications. Thus the concrete specifications are already there; our approach simply allows –or rather requires– the designer of the derived class to reuse the portion of the concrete specifications of the base class that correspond to functions that are not being redefined in the derived class. Our proof system, by reusing these specifications, perfectly mirrors the reuse of the code.

We should reiterate that our goal has not been to show that a derived class is a behavioral subtype of its base class. Rather, our system allows us to specify the behavior of the derived class, and show that its implementation meets the specification. Once this is done, we can proceed to establish any relations, such as behavioral subtyping, between it and any other classes, including the base class, using systems like those of [8, 6]. We should also note that our interest has been exclusively on specifying and verifying, using an axiomatic approach, the behaviors of classes defined using inheritance with the derived classes being allowed to access the base classes' protected members. Much work has been done in allowing for much more flexible forms of inheritance[2,3]. The focus of much of this work has been to build precise models, in a denotational style, of how these mechanisms are intended work. Once that question is settled, it would be appropriate to investigate specification and verification methods for classes built using these forms of inheritance.

Meyer [9] in his influential book on the object oriented approach develops a specification and verification method for classes defined using inheritance. Although he generally seems to advocate treating derived classes as subtypes, his formalism seems to be flexible enough to deal with cases where they are not. The more important difference with our approach is that Meyer does not allow for protected members. This, of course, as our examples in section 2 show, will lead to serious inefficiencies or lack of code reuse. And when it comes to specification and verification, the derived class designer has to work with the same abstract specification that clients use.

We conclude with one problem that we have not addressed. Consider a base class $B$ with two functions

$f$ and $g$. Suppose in the body of the function $f$ there is a call to the function $g$ (applied to the *self* object). Suppose $D$ is a derived class of $B$ and that, in $D$, we redefine the function $g$ but not the function $f$. Suppose in the client code that we have an instance of the class $D$ and that we apply the function $f$ to this object. Then the definition in the class $B$ will apply –since $f$ has not been overridden in $D$– but during this execution of $f$, the call to $g$ will result in the *new* $g$ being invoked.[3] But our system does not account for this. Since $f$ is not overridden, the designer of $D$ will simply reuse the (concrete) specification of $f$ from the base class $B$. (The system in [9] also seems to suffer from this problem.) To handle this situation correctly, we would have to somehow record in the (concrete) specification of $f$ that there is a call to $g$, and if $g$ is redefined in a derived class, to have that appropriately change the specification associated with $f$. We are currently working on a system that tries to do this.

# 6 Acknowledgments

We thank Gary Leavens for e-mail discussions on the relation between inheritance and subtyping.

# 7 References

1. M. Abadi, L. Cardelli, On subtyping and matching, *ECOOP '95*, LNCS 952, pp. 145-167.

2. K. Bruce, A. Schuett, R. van Gent, Polytoil: a type-safe polymorphic object-oriented language, *ECOOP '95*, LNCS 952, pp. 27-51.

3. W. Cook, W. Hill, P. Canning, Inheritance is not subtyping, *POPL, 1990.*

4. J. Guttag, J. Horning, J. Wing, The Larch family of specificatio languages, *IEEE Software*, vol. 2, 1985.

5. C. Horstmann, Mastering OO design in C++, Wiley, 1995.

6. G. Leavens, W. Weihl, Specification and verification of object-oriented programs using supertype abstraction, *Acta Informatica*, vol. 32, pp. 705-778, 1995.

7. B. Liskov, J. Wing, A behavioral notion of subtyping, ACM TOPLAS, vol. 16, pp. 1811-1841, 1994.

8. B. Liskov, J. Wing, A new definition of the subtype relation, ECOOP '93, pp. 118-141.

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

[3] In *C++* this would require $g$ to have been declared as a virtual function.