Incremental Specification and Verification of Object-Oriented Systems

Neelam Soundarajan and Stephen Fridella
Computer and Information Science
The Ohio State University
Columbus, OH 43210

e-mail: {neelam,fridella}@cis.ohio-state.edu

March 1, 1997

Abstract

Inheritance is the key mechanism of the Object-Oriented approach that enables designers to develop systems in an incremental manner to meet evolving needs. Given an existing *base class* B, a system designer can use inheritance to build a new *derived class* D that extends, or that differs appropriately from, B. But to make effective use of inheritance, it is essential for the designer of D to have a clear understanding of certain internal details of B. We propose an approach that will enable designers of a class, say, B to specify exactly the information about B that will be required by designers of derived classes like D that will inherit from B. Further our approach will allow the designer of D, in turn, to specify the precise information about D that future designers who wish to inherit from D will require. The designer of D will also be able to validate the specification of D, so these future designers will have a solid and reliable base to build on or rather, inherit from.

Keywords: Incremental system design, Incremental specification and verification, Abstract and concrete specifications.

1 Introduction and Motivation

The Object-Oriented approach to developing large systems is extremely powerful. Much of this power derives from the key notion of *inheritance*. Given an existing *base class*, a system designer can use inheritance to build a new *derived class* that extends, or that slightly differs from the base class. The effort involved in building a new class using inheritance from an existing base class is often significantly less than if the designer had started from scratch. Existing code can continue to use the base class while new code can exploit the added or modified functionality provided by the new class. And the newly developed derived classes can themselves serve as base classes for developing yet other derived classes. It is this incremental style of system development that inheritance allows, and even encourages, that makes the OO approach so valuable for the building and evolution of large systems.

In order to be able to effectively use inheritance, the designer of the derived class must have a deep understanding of how the base class operations function. In particular, he needs to understand thoroughly how these operations manipulate the *protected* data members of the base class. By contrast, a *client* of the base class has no interest in this information since the client has no access to the protected members of the (base) class. Rather, the client has an *abstract* view of the class and its operations. But the derived class designer must understand how the protected data members are used. Otherwise the new operations he introduces in the derived class, or his redefinitions of operations inherited from the base class, are not likely to function correctly. Worse, even the operations inherited unchanged from the base class may no longer work because the new or the redefined operations may modify the protected data members –which are shared by all the operations of the class– in ways unanticipated by the unchanged operations. Because of this problem, *implementation inheritance* of this kind has been criticized and it has been suggested that designers not be allowed to build new classes in this fashion. But experience shows that in the hands of careful designers this style of system development is very effective, and to disallow it would be unreasonable. Rather than outlawing implementation inheritance, what we need to do is to provide tools that the designer can use to develop the requisite understanding of the internals of the base class so that he is able to design correctly the changes he makes to the base class; and tools that he can use to validate his design. The goal of our paper is to develop such tools.

One of the most effective approaches to understanding the behavior of systems and validating their designs is that of formal specification and verification and this is the approach we adopt. The distinction drawn in the last paragraph between the needs of a client of a class B and a class designer who intends to design a new class by inheritance from B suggests that it would be useful to have two specifications of the class. The first specification which we will call the *abstract specification* of B will be for use by clients of B; it will describe the functionality of the various operations of B in terms of a conceptual model of the class. The second specification which we will call the *concrete specification* of B will be for use by designers of derived classes. This specification will contain information on the effect that the various operations of B have on its protected data members. Armed with the information in the concrete specification of B, the designer of a derived class D will be able to ensure that the changes he implements in D will cooperate harmoniously with the components of D inherited from B (in their use of the protected data members) and provide the functionality that D is expected to provide. This designer will also have to come up with appropriate abstract and concrete specifications for D, so that clients of D as well as future designers who might wish to use D as a base class in building yet other classes will have the information they need. Finally, the verification procedure that we propose in the paper will allow the designer of D to validate his design, i.e., demonstrate that his design of D does indeed meet its abstract and concrete specifications. An important aspect of our approach is that it is *incremental* in exactly the same manner that the derived class is incrementally developed from the base class. The concrete specifications of operations that are not redefined in the derived class are essentially identical to their specification in the base class, and no re-verification is needed for these operations. This is particularly important in situations where the derived class designer may not even have access to the code of the operations of B; this may happen if, for instance, B was purchased from a software vendor who may have –because of the proprietary nature of the source code of B– only provided object code versions of B. In this case re-verification of these operations would clearly be impossible.

One problem with approaches that use formal specifications and verification is that they can be difficult to use in practice. There is certainly some truth to this, and we will not try to pretend that 'it ain't so'. The solution, we believe, is to try to use the approach in a pragmatic manner. What we mean by this is that, for instance, in verifying that a new operation introduced in the derived class, the designer may use informal and operational arguments rather than formal axiomatic proofs. The important part in any case is to ensure that the specification, the concrete one in particular, does capture the effect of the various operations on the (protected) data members, so that future designers have the information they need to be able to inherit from this class. This is the procedure –fairly precise specifications combined with operational arguments– that we will follow in the example we consider later in the paper. We will return to this point in the final section of the paper.

The rest of the paper is organized as follows: In the next section we introduce a simple example to show how inheritance can be very useful in building systems incrementally to meet evolving needs. In section 3.1 we present our notation for providing abstract and concrete specifications of a class. In 3.2 we explain how the specifications can be validated, in particular how part of the specification of the base class –the part that corresponds to operations that are inherited unchanged– is inherited and used in the derived class, and how we can show that these operations cooperate with the new operations as well as with the redefined operations as intended by the derived class designer. In section 4 we show how our approach can be used to deal with the example presented in section 2. In the final section we summarize our approach and explain how it allows system designers to understand, specify, and validate, systems built incrementally using inheritance.

2 Incremental Design

Consider a Flight class such as might be part of an airline system. (Other classes in the system might be City, Passenger, etc.) A Flight object will be a collection of passengers who have reserved seats on a particular flight. We need to be able to add and remove passengers to and from the flight. The operations Reserve() and Cancel() will serve this purpose. We will also have an operation IsIn() which allows us to check if a given passenger is currently on the flight or not. ¿From the point of view of a client of this class a Flight object is the set of passengers who have reserved seats on the flight, Reserve() will add a passenger to this set, Cancel() will remove a passenger from the set. This will be stated precisely in the abstract specification of the class we present in section 4.

A simple implementation of the class is given in figure **??**. We use C++ syntax for concreteness, but there is nothing particularly C++ specific in our approach.[1] In the implementation, the array plist[] contains the set of passengers currently on the flight. The code for Cancel() has been omitted; the interested reader should be able to supply the missing details. Note that due to space considerations, this class is artificially simple. The concrete specification of the class, as we will see in section 4, will specify the effects of these operations on the (protected) data members of the class such as plist[]. 'last' is index of the plist element where the next passenger to be added to the flight will be stored.

```
class Flight {

    protected:

        Passenger plist[200]; // array of passengers; max. no. of passengers in a flight = 200.

        int last;    // passengers currently on the flight are in
                     // plist[0] through plist[last-1].

    public:

        Flight( ) { last = 0; } // This is the constructor function that initializes every flight object.
                     // All we need do is set last to 0.

        void Reserve( Passenger& p )
        {     plist[last] = p; last = last + 1;}

        void Cancel( Passenger& p )
        {     // Find index i of the passenger p;
              // Move backward by one element, passengers plist[i+1]
              // through plist[last-1].
              last = last − 1;}
};
```

Figure 1: Implementation of Flight class

Suppose now that the airline needs to be able to 'bump' passengers from a flight; i.e., because of the all-too-prevalent practice of over-booking flights, the airline might have to remove a passenger from the flight. But this is not the same as the Cancel() operation; that operation takes, as argument, a passenger

[1] For the reader unfamiliar with C++ syntax [8], the following notes should help in reading the code: The symbol '//' denotes that the rest of that line is a comment. Array elements are numbered starting at 0, not 1. '=' is the assignment operation. '&' denotes that the particular parameter is passed by reference, not by copy. Function definitions specify the type of value returned by the function (void for functions that don't return any values), the name of the function, then the list of parameters, and finally the body of the function, enclosed inside braces ({. . . }).

The class definition consists of the name of the class, followed by the body of the class in braces; for derived classes, the name of the class is followed by the name of base class and then the body of the class in braces. The body of the class specifies the data members and the member functions (also called operations). The data members may be public, protected, or private, but in this paper we will follow the fairly common convention that all data members are protected. For each member function we specify the type of value returned by the function, the name of the function, the parameters expected by the function, and the body of the function. Member functions may also be public, protected, or private, but we will assume that all are public.

If f is a flight object and p1 a passenger object in a piece of client code, we can add p1 to f by saying f.Reserve(p1).

p and removes p from the flight. Bump() should take no argument and remove an 'appropriate' passenger from the flight. How do we choose which passenger to bump? The most reasonable approach might be to bump the passenger who was added to the flight *last*. Such a class, let us call it Bumpy_Flight(!), can be easily constructed by using inheritance from Flight:

```
    class Bumpy_Flight : public Flight {
        public:
        Bump( ) { last = last − 1; }
};
```

Figure 2: Implementation of Bumpy_Flight

The abstract specification of the Bump() operation will say that it will remove the most recently added passenger from the flight. If all we knew about the Flight class was what was contained in its abstract specification, we would have to implement Bumpy_Flight from scratch. It is our knowledge about how Reserve() stores information in the plist[] array and last, and our ability, in the derived class Bumpy_Flight to access these protected members of Flight that enabled us to build on the work that has already gone into the design of the Flight class.

In this example we did not have to redefine any of the operations defined in the base class and inherited into the derived class. In other examples this will not be the case. For instance, what if the airline decided that passengers should be bumped not in the reverse order of when they made their reservation but in some other order (say, people with the fewest frequent-flier miles will be bumped first)? We can implement such a class, call it, Bumpy_FF_Flight, as a derived class of Bumpy_Flight by redefining the Reserve() operation so it adds passengers into the flight list according to their frequent-flier mile status. No other changes would need to be made. As we said in the introduction, it is this ability of inheritance to meet evolving system needs in an incremental manner that makes it such a valuable tool; and the possibility of making mistakes when designing these changes that makes it essential to validate the designs using an approach like that proposed in the next section.

3 Specification and Validation of Class Behaviors

In 3.1 we will present our notation for abstract and concrete specifications of classes. In 3.2 we will see how such specifications, especially of derived classes, can be validated.

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.[2]

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 [2], consisting of a conceptual, mathematical model $\mathcal{M}$ of B, and the specifications of the individual methods of B in terms of pre- and post-conditions in this conceptual model. In formalisms like that of [4,5], $\mathcal{A}$ would essentially be the *complete* specification of B, and indeed we will borrow notation for this part of our specification from [4,5]. Specifically, the pre-condition of an operation will be an assertion over the conceptual initial state, and will express the condition that must be satisfied when the operation in question is invoked. For instance, for the Flight class, the conceptual model is the set of passengers on the flight. The abstract specification of the Cancel(p) operation, will have as the pre-condition the requirement that p must be a member of this set. The post-condition of an operation will be an assertion over the conceptual state that exists when the method in question starts execution and the state that exists when the method finishes execution. For the Cancel(p), the post-condition will say that the value of the set of passengers when the operation finishes should be equal to the value when the operation started, less p.

What about $\mathcal{C}$? It will also consist of specifications of the operations of B in terms of pre- and post-conditions, but these will be concrete pre- and post-conditions, i.e., will be in terms of the (protected) data members of the class, not in terms of the conceptual model $\mathcal{M}$. 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. The invariant will be an assertion over the concrete state; the pre-condition and post-condition of an operation will be similar to their counterparts in $\mathcal{A}$ except that these assertions are over the concrete state. One other item must be specified in $\mathcal{C}$: an *abstraction function*, ε, that maps the concrete state, i.e., the values of the set of protected members, to the corresponding conceptual state. ε will play an important role in relating the concrete specification $\mathcal{C}$ to the abstract specification $\mathcal{A}$.

When a derived class D is designed that inherits from B, we will be able to reuse much of the specification of B, mirroring the reuse that occurs in the design of D.

3.2 Verification of Class Behavior

In order to validate a class, we need to verify the following:

1. The 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.

[2]We consider how our approach can be extended to handle private members in [7].

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 ω and ω' denote the initial and final (concrete) state before f starts and when it finishes. We then need to establish that the body $f.b$ of f meets the condition

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

Whether we formally establish (1) using appropriate axioms and rules of inference, or we convince ourselves with informal, and operational, arguments that (1) holds, depends upon the particular system being designed, and perhaps on the designer involved. For critical systems we might prefer to use a formal approach; for other systems, informal arguments might suffice.

If f is *inherited* from the base class B of D and not redefined 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.[3] *This is the verification analog of code reuse: if f is not being reimplemented, then there is no need to reverify it.*

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. If f is a constructor function, the object does not exist at the start of the function. The pre-condition will only be a requirement on the parameters of the function,[4] and the invariant will not appear on the left side of $(2a)$. Of course, the invariant must appear on the right side, since we want to be sure that once the object is constructed, it satisfies the invariant.

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 ω consists of two components, σ being the component inherited from the base class B, and τ 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 σ whereas I, being the invariant for the derived class, will also involve τ. But even though $c.post_f$ doesn't involve τ, 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 τ 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 ε 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:

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

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

$(3a)$ ensures that if the abstract pre-condition of the operation f is satisfied (which is all the client can be expected to worry about), the concrete one will also be satisfied. $(3b)$ ensures that if when the operation finishes, the concrete post-condition is satisfied, so will the abstract one (which is what the client is interested

[3]We should note here that this claim is strictly true only if f does not invoke any *virtual* functions that are redefined in D. If f did not meet this condition, the fact that it is not redefined in D is no assurance that its behavior will be the same in D as in B; the problem is that if this f is invoked on an object of type D, during this execution it will invoke the redefined g, not the original one. As a result, the net effect of this invocation of f can be different from that specified in the concrete specification of f in the base class. This type of *polymorphic* function can be very useful in certain OO systems; in [7] we discuss the problems involved in dealing with such functions. In the current paper we will not consider polymorphic functions.

[4]The pre-condition for the other functions will also, in general, include requirements on their parameters; so $c.pre_f$ will be an assertion involving both ω and these parameters. In order to keep the notation simple, we didn't include the parameters explicitly in our rules.

in).

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:

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

and $(3b)$ as:

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

In the next section we will show how to apply the rules we have presented here to the example introduced in section 2.

4 Incremental Specification and Verification

Let us now apply our approach to the Flight and Bumpy_Flight classes of the example presented in section 2. The conceptual model for the Flight class is a set of passengers. When a Flight object is constructed, this set will be empty. Reserve(p) will add p to the set, Cancel(p) will remove p from it. More precisely, the abstract specification of flight is[5]:

Conceptual model of Flight is a set of passengers.

$a.post_{Flight} \equiv self' = \emptyset$

$a.pre_{Reserve(p)} \equiv p \notin self$
$a.post_{Reserve(p)} \equiv self' = self \cup \{p\}$

$a.pre_{Cancel(p)} \equiv p \in self$
$a.post_{Cancel(p)} \equiv self' = self - \{p\}$

Figure 3: Abstract specification of Flight

The conceptual model for Bumpy_Flight is not a set of passengers but rather the *sequence* of passengers, the order in the sequence being the same as the order in which the passengers made their reservations. It is important to reflect this order in the conceptual model, for otherwise the behavior of the Bump() operation might look strange to the client – it would seem to remove an arbitrary passenger from the flight; this may be how airlines sometimes operate in practice, but our Bumpy_Flight class is better behaved than that! The pre- and post-conditions for the various operations will also change because they have to be in terms of the new conceptual model. We will use the following symbols for useful operations on sequences: $<>$ is the empty sequence; $\in s$ means p is in the sequence s and $p \notin s$ means p is not s. $s\hat{\ }p$ is the sequence obtained by appending the element p to the tail end of the sequence s. $s - p$ is the sequence obtained by removing all occurrences of p in s. AllButLast(s) is the sequence obtained omitting the last element of s.

Conceptual model of Bumpy_Flight is a sequence of passengers.

$a.post_{Flight} \equiv self' = <>$

$a.pre_{Reserve(p)} \equiv p \notin self$
$a.post_{Reserve(p)} \equiv self' = self\ \hat{}\ <p>$

$a.pre_{Cancel(p)} \equiv p \in self$
$a.post_{Cancel(p)} \equiv self' = self - p$

$a.pre_{Bump} \equiv self \neq <>$
$a.post_{Bump} \equiv self' = AllButLast(self)$

Figure 4: Abstract specification of Bumpy_Flight

From the client's point of view, these two classes are quite different since their conceptual models are different from each other. But from the class designer's point of view, Bumpy_Flight is a simple extension of Flight. This is reflected in the similarity of the concrete specifications of the two classes. We will start with

[5]self in the specifications, following standard OO conventions, refers to the flight object that we are considering. As in section 3, primed variables (like self') refer to the value of the variable when the operation in question finishes, unprimed variables denote the values when the operation begins.

the concrete specification of Flight, the base class. In the concrete specification we must specify an invariant, the pre- and post-conditions of the various operations, and the abstraction function which maps the concrete state to the conceptual model of the class:

Abstraction fn: $\varepsilon_{\mathsf{f}}(plist, last)$ = *set of all passengers from plist[0] through plist[last-1]*

Invariant: $Invf \equiv$ *no duplicate elements among plist[0] through plist[last-1]*

$c.post_{Flight} \equiv last' = 0$

$c.pre_{Reserve(p)} \equiv$ *there is no i* $(< last)$ *such that* $plist[i] = p$
$c.post_{Reserve(p)} \equiv (plist'[last] = p) \wedge (plist'[0..last\text{-}1] = plist[0..last\text{-}1]) \wedge (last' = last + 1)$

$c.pre_{Cancel(p)} \equiv$ *there is some i* $(< last)$ *such that* $plist[i] = p$
$c.post_{Cancel(p)} \equiv (last' = last-1) \wedge$ *there is some i* $(< last)$ *such that*
$(plist[i] = p) \wedge (plist'[0..i\text{-}1] = plist[0..i\text{-}1]) \wedge (plist'[i..last'\text{-}1] = plist[i+1..last\text{-}1])$

Figure 5: Concrete specification of Flight

The post-condition of Reserve(p) says that p has been added at the end of plist; that of Cancel(p) says that p has been removed from the array. The invariant says that there are no duplicate passengers in the array. This is needed because Cancel operation removes only one occurrence of p from plist, but the abstract specification requires that *all* copies be removed. The invariant ensures this.

The concrete model of Bumpy_Flight has much in common with that of flight:

Abstraction fn: $\varepsilon_{\mathsf{bf}}(\mathsf{plist}, \mathsf{last}) =< \mathsf{plist}[0], \ldots, \mathsf{plist}[\mathsf{last}] >$

Note: Invariant, and pre- and post- conditions for inherited operations are the same as in Flight

$c.pre_{Bump} \equiv last \neq 0$
$c.post_{Bump} \equiv (last' = last-1) \wedge (plist' = plist)$

Figure 6: Concrete specification of Bumpy_Flight

The abstraction function $\varepsilon_{\mathsf{bf}}$ maps the concrete state to the *sequence* of passengers in the same order as in the array. The invariant is the same as in the base class; the pre- and post-conditions of Reserve and Cancel are also the same as in the base class since they are not redefined. The specification of Bump says that the operation reduces the value of last by 1.

What about validation of our classes? Since our focus is on inheritance, we will just consider the Bumpy_flight class. According to the formalism of section 3, we need to verify three things:

Rule (1)– Method Bodies: First we must be sure that the code of each operation satisfies the stated concrete pre- and post-conditions. As we said in section 3, there is no work here as far as operations inherited from the base class and not redefined in the derived class are concerned. This leaves just the Bump operation whose implementation (figure 2) obviously meets its concrete specification. For complex operations this check might be complex, and we might prefer to use operational arguments.

Rule(2)– Invariant: Next we must make sure that each operation preserves the concrete invariant $Invf$. In our case the invariant in the derived class is the same as in the base class, so again there is no work as far as Reserve and Cancel are concerned. For Bump it is easy to check that it preserves $Invf$ since it does not change the array plist.

Rule(3)– Abstract Specifications: Finally we need to be sure that if the concrete specifications are satisfied, the abstract ones will also be. That is, for each operation we must be sure that if the abstract pre-condition holds on the abstract state, then the concrete pre-condition holds on the concrete state. Also, we must be sure that if the concrete post-condition of an operation holds on a concrete state, then the abstract post-condition of the same operation holds on the corresponding abstract state. In both cases we are allowed to assume that the concrete invariant holds on the concrete state. For pre-conditions note that a passenger is in the sequence $\varepsilon_{bf}(\omega)$ if and only if the passenger is present in the plist array in state ω. Therefore, the abstract pre-conditions clearly imply the concrete pre-conditions for Reserve, Cancel, and Bump. For post-conditions note that the order of the passengers in the sequence $\varepsilon_{bf}(\omega)$ is the same as the order of the passengers in the plist array in ω. Thus adding a passenger to the end of the array in ω corresponds to adding a passenger to the end of the sequence $\varepsilon_{bf}(\omega)$. From this it should be clear that the concrete post-conditions of Bump and Reserve imply their abstract post-conditions. For Cancel however, we need help from $Invf$. $c.post_{Cancel(p)}(\omega, \omega')$ clearly implies that one copy of p is removed from the sequence $\varepsilon_{bf}(\omega)$ to make $\varepsilon_{bf}(\omega')$, but what about any other copies? $Invf$ assures us that there are *no* other copies of p in plist and thus in the sequence $\varepsilon_{bf}(\omega)$.

With that we can be sure that our implementation of Bumpy_Flight is indeed correct; we were able to reuse much of the validation of the Flight class, closely mirroring the way the former class was implemented by inheritance from the latter.

5 Discussion

Inheritance, when used carefully, is a very effective tool in designing systems to meet evolving needs. What we mean by 'used carefully' is that it is important for a derived class designer to fully understand how the operations of the base class B that he intends to inherit use the protected data members of that class. A high-level understanding of B's operations, in terms of a conceptual model of B, while it satisfies the needs of a client of B is not enough for the derived class designer. Further, this designer does not always have the option of examining the source code of B's operations to see how these operations use the data members, since the source code may not be available. The only alternative is to rely on what we have called the concrete specifications of B. In other words, the abstract specification of a class makes it possible for clients of B to use the class. The concrete specification of B makes it possible for other designers to inherit from B and build new classes that meet the needs of the application as it evolves.

Note also that the need to have a concrete specification of the class B is independent of whether or not the designer of B believes in the usefulness of the specification / verification approach. If B does not come with a concrete specification, every designer who intends to build new classes by inheriting from B will be obliged to examine the source code of B. And if the source code of B is not available, these designers will be forced to build their classes from scratch. Similarly, as has been argued persuasively by authors like Meyer [6], clients of B will be unable to use B if it does not come equipped with what we have called abstract specifications. One could then ask, why not use informal rather than formal ones? After all, many practicing programmers find it easier to work with informal specifications. This is certainly true and often informal specifications suffice, especially for clients of a class. We believe, however, that a derived class designer needs a much better understanding of how the operations of B manipulate the protected data members than can be expressed using informal specifications. It would be easy to miss a subtle aspect of how the operations of B use the data members; the result could well be a (derived) class that is extremely hard to debug since even the operations that were inherited unchanged from B might not work as they did in the base class. Thus while a client of B with an incomplete understanding of one of the operations of the class might find that operation exhibiting puzzling behavior, a derived class designer with holes in his understanding might end up with a derived class that doesn't work at all. This is exactly the basis of the criticism of implementation inheritance; but a derived class designer who relies on the concrete specification of the base class will be able to avoid this problem. The example we considered in sections 2 and 4 is a case in point. This example is so simple –and we had to choose a very simple example because of space limitations– that it would seem unnecessary to have a precise specification for it. But even in this simple example it would be possible for a derived class designer to misunderstand an informal specification; on the basis of an informal specification, he might think, for instance, that the most recently added element is stored in `plist[0]` rather than in `plist[last-1]`. The `Bump()` operation that a designer with this misunderstanding will implement will bump not the *last* passenger but the *first* one who made his reservation! Having a *formal* concrete specification of the `Flight` class will eliminate such problems.

Formal verification is a different question, however. While it can be done in principle, in practice most designers find it a forbidding task. But we believe that not all parts of the verification task are equally difficult. In practice we often have a good operational understanding of how a given piece of code works and how it meets the conditions expressed in its pre- and post-conditions. Converting this understanding into a formal axiomatic verification that the code meets its specification is often difficult. But checking, for instance, that any final state that satisfies a given post-condition also satisfies a particular invariant does not involve such conversions, and tends to be straightforward. This is why we have proposed what we think is reasonable middle ground: the designer uses operational reasoning in establishing that methods meet their pre-condition, post-condition specification; and he establishes the other parts of the specification, rules (2) and (3) of section 3, using logic-based arguments.

We conclude with a few final remarks: authors such as [4,5,3] have done a significant amount of formal verification work on the notion of a *behavioral subtype*. This is a relation that can be established between the abstract specifications of two classes, and intuitively means that instances of one class can be substituted for instances of the other class without affecting the program behavior. This information is of interest to *clients* of the two classes. We are considering a different issue in this paper: how can we verify that the implementation of a class defined by inheritance meets its specifications? This is a question of great interest to *class designers*. Another important issue relating to inheritance is the question of *covariance* vs. *contra-variance* (See for example [1]). Researchers in this area have tried to answer the question: how should the *types* of class operations by allowed to change in derived classes? While there are interesting theoretical problems involved in this question, in practice popular OO languages like C++ avoid this question by requiring the types of operations to be identical in the base class and derived class. Since our interest is in applying formal reasoning techniques in practice we have assumed that functions inherited into the derived class expect the same types of parameters as they do in the base class.

6 References

1. G. Castagna, Covariance and contra-variance: conflict without a cause, *ACM TOPLAS*, vol. 17, 1995, pp. 431-447.
2. J. Guttag, J. Horning, J. Wing, The Larch family of specification languages, *IEEE Software*, vol. 2, 1985.
3. G. Leavens, W. Weihl, Specification and verification of object-oriented programs using supertype abstraction, *Acta Informatica*, vol. 32, pp. 705-778, 1995.
4. B. Liskov, J. Wing, A behavioral notion of subtyping, ACM TOPLAS, vol. 16, pp. 1811-1841, 1994.
5. B. Liskov, J. Wing, A new definition of the subtype relation, ECOOP '93, pp. 118-141.
6. B. Meyer, Object oriented software construction, Prentice-Hall, 1988.
7. N. Soundarajan, S. Fridella, Inheriting and modifying behavior, submitted to *TOOLS '97*.
8. B. Stroustrup, The C++ programming language, Addison-Wesley, 1991.