Inheritance: From Code Reuse to Reasoning Reuse

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

September 30, 1997

Abstract

Much of the power of the Object-Oriented approach to developing large systems derives from the notion of *inheritance*. Given an existing *base class*, a designer can use inheritance to build a *derived class* that extends, or that slightly differs from the base class. But in order to exploit the full potential of inheritance to building systems incrementally, the designer must also be able to *reason* about the derived class incrementally. This paper presents a specification notation and verification procedure that allows such incremental reasoning to be carried out; the approach makes important use of the *concrete specification* of a class, in addition to the usual *abstract specification*. The reasoning reuse that the approach enables is illustrated by applying it to a simple example. Finally, the motivation underlying and the details of the approach are contrasted with those of behavioral subtyping.

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. But *code reuse* by itself is of little value in the absence of *reasoning reuse*. In order to be able to exploit the full potential of inheritance to build systems incrementally, the system designer must be able to reason about the derived class in an incremental manner from the base class. The goal of this paper is to present a specification notation and verification procedure that will allow the derived class designer to reason incrementally, given the specification of the base class (and assuming that that class has been verified); our approach will mirror the code reuse that inheritance enables in the form of specification reuse and verification reuse.

Note that we are *not* asking the question 'Given a base class B and a derived class D, under what conditions can a client program use objects that are instances of D in place of instances of B?' The answer to that question is provided by the work on *behavioral subtyping* [1, 6, 7]. Informally, a class A is a behavioral subtype of another class B if the behavior exhibited by objects that are instances of A is in some sense consistent with behaviors allowed by the class B. The class A may be a behavioral subtype of B independently of whether it is implemented from scratch or as a derived class of B. And conversely, if D is a derived class of B there is no guarantee that it will be a behavioral subtype of B.[1] The work on

[1] [3] and [2] consider restrictions on how inheritance is used to ensure that the derived class is a behavioral subtype of the base class; we will relate our work to theirs later in the paper.

behavioral subtyping is motivated by the desire to free the designer of the client code from having to reverify his[2] code if instances of one class are used where objects that are instances of another class were expected. Our work on the other hand is motivated by the desire to free the designer of the derived class from having to re-specify or reverify code that the base class designer has (presumably) already specified and verified.

Despite the distinction between behavioral subtyping and inheritance, it is often claimed, and in this context this requirement is usually called the *Liskov Substitution Principle (LSP)*, that the derived class must be a behavioral subtype of the base class (see, for instance, [8], or the frequent debates on the topic in the newsgroup *comp.object*). The reason for this seems to be the concern that unless this requirement is imposed, inheritance will be no more than mere code reuse (disparaging terms such as '*code scavenging*' or '*code hack*' are often used to describe inheritance in the absence of behavioral subtyping). Our work shows that such concerns are ill-founded, and that using our approach the derived class designer can exploit inheritance to achieve not only code reuse, but significant specification and reasoning reuse as well.[3]

A key idea underlying our approach is to associate *two* specifications with each class B. The first which we call the *abstract specification* of B is for use by the clients of B and is the standard specification of the class in terms of an abstract conceptual model of the class. The other specification of B which we call the *concrete specification* is for use by derived class designers; it describes the behavior of the *methods* of the class in terms of their effects on the internal structure, i.e., the *member variables*[4] of the class. The information contained in the concrete specification of B is critically important to the designer who wishes to design a derived class D of the base class B since any new methods he wants to define in D, or methods that are defined in B but the designer wants to redefine in D, must work with these same member variables, and more important, must cooperate with the methods inherited unchanged from B and these methods, of course, use these variables. The concrete specification of B will also include a (concrete) invariant.

The derived class D will similarly have an abstract specification and a concrete specification. The portions of the concrete specification of B that correspond to methods that are inherited unchanged into D will be inherited into the concrete specification of D; this is the specification reuse that mirrors the code reuse that the designer achieves by inheriting the methods. Further, the designer does not have to reverify these methods, this being the verification reuse corresponding to the same code reuse.[5] The invariant in the concrete specification of D may be different from that of B, or it may be identical to it, or a strengthening of it. It is often argued that inheritance can be a dangerous mechanism to use because the derived class designer might inadvertently destroy, in one of the functions he introduces or redefines in D, some invariant relation that the base class methods rely upon. Our work shows that while it is true that the amount of verification reuse may be greater if the derived class preserved the base class invariant, correctness does not require it and moreover, a considerable degree of verification reuse is possible even if the derived class has a quite different invariant than the base class.

The idea of concrete specifications is not new. Such specifications are also used in standard approaches such as those of [5], although for a different purpose: Consider the class B. How do we establish that it meets its abstract specification? The actual bodies of B's operations[6] are of course written in terms of the internal data members of B and operate on those members, not on a 'conceptual' entity. So in order to show that the abstract specification of B is indeed satisfied, the implementor of B shows that the functions of B satisfy its concrete specification, defines an *abstraction function* that maps the concrete states of the class to corresponding 'abstract states', and shows that under this abstraction function, the pre- and post-conditions in the concrete specification, in some sense, imply the corresponding pre- and post-conditions in the abstract specification. Once this is done though, the concrete specification is considered, in the standard approaches, to have served its purpose and is essentially forgotten. Our work shows that while the client of the class B may have no interest in the internal details of the class or in the information contained in its concrete specification, the derived class designer is very interested in this information and by having access

[2] Following standard practice, we use 'he', 'his', etc. as abbreviations for 'he or she', 'his or her' etc.

[3] It is of course true that if the derived class is not a behavioral subtype of the base class then any client code designed to deal with base class objects that is instead supplied with instances of the derived class will have to be reverified.

[4] We will use *C++*-like terminology and notation but our approach is generally applicable to all class-based OO languages.

[5] We will see later that there are some difficulties in dealing with *dynamic binding*, i.e., the *virtual* functions of *C++*.

[6] We will use the terms 'methods', 'operations', 'functions', etc. interchangeably.

to this specification, he will be able to achieve specification (and verification) reuse that mirrors the code reuse he obtains by using inheritance. In other words, and in keeping with the spirit of reuse, the concrete specification is serving double duty in our approach, once to help in verifying the base class, and a second time to help the derived class designer in reasoning about his class.

The main contributions of this paper may be summarized as follows:

- Makes the case that common wisdom to the contrary notwithstanding, inheritance can be used to define derived classes reusing parts of the work that has gone into defining the base class, and that this reuse is not mere code reuse but can be matched in the specification and verification of the resulting derived classes; and that these derived classes need not be behavioral subtypes of the base class.
- Presents a specification notation, and a verification procedure, that allow the derived class designer to reuse appropriate portions of both the specification and verification that has been performed for the base class; and applies the approach to a simple example.
- Explores the relation between specifications (of different classes) required by behavioral subtyping, and the relation, corresponding to inheritance, between abstract and concrete specifications of the base and derived classes

The rest of the paper is organized as follows: In the next section we present abstract and concrete specifications. We consider how much specification reuse we can expect under different situations; much of the concrete specification in particular is usually reusable, and we show how the concrete specification of the derived class may be obtained from that of the base class. In the third section we present our verification procedure, again focusing on the question of how much reuse we can expect, in what steps of the verification procedure, and under what conditions. We present a simple example in the fourth section and show how our approach works for this example. Again the focus in this section is on how much reuse we are able to get in the specification and verification, so many of the details, especially with regards to the verification of actual method bodies, are omitted. In the final section, we reiterate the importance of inheritance in building systems incrementally, and the equal importance of an incremental specification and verification procedure such as ours; contrast the motivation and purpose of our system with that of work on behavioral subtyping; and explain how our approach extends easily and naturally to multiple inheritance.

2 Reusing Specifications

Consider a base class B and a derived class D (defined by inheritance from B). In a language like *C++*, any of the members of B can be declared *public, protected,* or *private*; the public and protected members of B will be accessible in D but not the private members. Following generally 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 discussion we will assume that there are no private members.[7]

The specification $\langle \mathcal{A}, \mathcal{C} \rangle$ of B will consist of a component $\mathcal{A}$, the *abstract specification* of B; and a component $\mathcal{C}$, the *concrete specification* of B. The abstract specification $\mathcal{A}$ will be the usual *ADT*-type specification [4], consisting of a conceptual, mathematical model of B, and the specifications of the methods of B in terms of pre- and post-conditions in this model. The pre-condition of a method will be an assertion on the (conceptual) state of the object that must be satisfied when the method is invoked; the post-condition is an assertion involving both the state at the time the method is invoked, and the state when the method completes execution. For simplicity, we will not include an invariant in the abstract specification.

The concrete specification will also consist of specifications of the operations of B in terms of pre- and post-conditions, but these will be in terms of the (protected) data members of the class, not in terms of the conceptual model. In addition, $\mathcal{C}$ will contain an *abstraction function* that maps the concrete state, i.e., the values of the set of protected member variables of the class, to the corresponding conceptual state. Finally, $\mathcal{C}$ 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 and the abstraction function will allow us to relate $\mathcal{C}$ and $\mathcal{A}$.

Consider now the derived class D. Three kinds of operations may exist in D: operations inherited from the base class B, new operations defined in D, and operations that are defined in B but are redefined in D.[8] In addition, the derived class may declare new member variables which are available for use by the operations defined or redefined in D.

Suppose f is an operation of D inherited from B. The concrete specification of f will be exactly the same as its concrete specification in B. This mirrors, in the specification task, the code reuse that the designer achieved in inheriting the function from the base class. There is a minor problem we need to handle: The specification of f inherited from B will not say anything about the effect of f on any new member variables introduced in D since these variables did not even exist when the base class specification was being written down. But since the body of f (as coded in the class B) will not refer to these variables, we can be sure that the execution of f will not have any effect on these variables. Hence we can strengthen the concrete specification inherited from B by adding a clause to the post-condition of f asserting that the values, at the time that f finishes execution, of the member variables introduced in D are the same as their values at the start of f's execution.[9]

As far as operations that are defined or redefined in D are concerned, we must, of course, come up with appropriate concrete specifications describing their effects on the various member variables (both those declared in D as well as those inherited from B).

---

[7] We consider private members in [11]. *Eiffel* has no private members, only public and protected; thus all members of a class are accessible in derived classes. *Java* allows not only derived classes, but all other classes that are in the same 'package' as B to access all members of B except those declared private. But these other classes themselves are not visible outside so it should be possible to extend our approach to deal with this case.

[8] In *Eiffel* when an operation is redefined, it is possible to change the types of the parameters or of the results, subject to *Eiffel*'s co-variant rules; *C++* does not permit such changes. We will follow the *C++* convention and thereby also avoid the long-running controversy regarding covariance versus contravariance.

[9] Since our post-conditions are assertions involving the states at the start of the function execution and at its end, it is easy to add such a clause. Thus if τ is the set of variables introduced in D, then all we need do is add the clause $(\tau = \tau')$ to the post-condition of f, τ and τ' denoting the values of the corresponding variables at the start and end of f.

We should note that if f is a so-called *recursive* or *polymorphic* function, i.e., it invokes another method g of B that is declared *virtual*, and g is redefined in D, then even if f is not redefined in D its behavior when applied to instances of D might be different than when applied to instances of B since in the former case the g that f will invoke will be the one defined in D, not the one in B. This means the specification of f may have to be changed to reflect this new behavior. We will consider such functions briefly later in the paper.

What about the (concrete) invariant for D? Since new variables have been declared in D and since the invariant for D may have to include information about these variables, we cannot expect to simply reuse the invariant from B. But even in the absence of such variables, the invariant of B may not be valid in D since the operations defined (or redefined) in D may not maintain this invariant.[10] The amount of reuse in the verification task, in particular in showing that the (derived) class meets its abstract specification, will partly depend upon the relation between the base class and derived class invariants. Later in the paper we will discuss three possible situations: the first where the invariants of B and D are identical to each other; the second in which the invariant for D is *stronger* than the invariant for B (i.e., implies it); and the third in which D's invariant is just different from B's.

Next consider the abstract specification of D. In general, there is no relation between the abstract specifications of the base and derived classes. This is perhaps the most important difference between behavioral subtyping and the formalism we are proposing. Behavioral subtyping imposes precise requirements on this relation; but inheritance in general allows us to define derived classes that, from the point of view of the client, look quite different from the base class, or at least are not behavioral subtypes. This difference will be reflected in the difference between the abstraction functions of the two classes. In most cases though, these functions will be quite similar to each other; and to that extent, the verification task for D, in particular showing that the concrete specification implies the abstract one since that is where the abstraction function enters the picture, can reuse the corresponding work that has been done for the base class. We will see that precisely this happens in the example we consider in section 4; in that example the abstraction function in the derived class is the same as in the base class.

Before concluding this section it may be worth noting that the specification reuse is mainly at the concrete level. This should not be surprising; inheritance is at the concrete level, and hence we can expect that we should be able to reuse much of the concrete specification of the base class. But there is no guarantee that the *abstract* specifications will be reusable since there is no necessary relation between the abstract models of the base class and the derived class. Much criticism of inheritance that does not produce behavioral subtypes seems to be based on the assumption that because the abstract specification is not reusable, that there is no reuse at all; but that is incorrect as the discussion in this section (as well as the discussion in the next section about proof reuse) shows.

[10]The invariant *Inv* of a class is not an assertion that is required to hold at the start of each operation f; rather f will guarantee that if the state when f starts execution satisfies *Inv*, then the state at the end of f will also satisfy it. Thus even if the invariant of the base class is not maintained by the derived class, one or more of the operations defined in the base class may be inherited by D. This is different from the kind of invariant that, for instance, [9] uses.

3 Reusing Verification

Consider again the base class B and the derived class D. Let us assume we have the concrete and abstract specifications for both classes, the concrete specifications for methods of D that are inherited from B being the same as in B, as we saw in section 2.

How do we verify that the class D meets its specifications? We will assume that B has already been verified to meet its specification; indeed the focus in this section will be to see how much of the verification already carried out for B can be reused for D. We have three sets of proof obligations in verifying D:

1. The various methods of D, both those inherited from B and those defined (or redefined) in D, meet their concrete pre- and post-conditions.
2. The various methods D leave the invariant satisfied when they complete execution.
3. Given that D meets its concrete specification (which is what steps 1 and 2 establish), D also meets its abstract specification.

Let us consider each of these tasks in turn. For task (1), if a given function f is defined or redefined in D then we may use a standard axiomatic approach to verify that the body of f meets the concrete specification in terms of its pre- and post-conditions. We have to, of course, make sure that we take care of the parameters appropriately but classes and inheritance introduce no new problems here.

When a function f is redefined in D, it is often the case that the reason for the redefinition is that we want to *add* something to what f does. Thus the redefined f usually calls the f defined in the base class, plus has some additional code. When verifying such a redefinition, we will of course rely upon the base class (concrete) specification of f to understand what this call will do. Thus even in this case, we achieve some reuse by having access to the concrete specification of the base class f. If we did not have the information provided by this concrete specification, we may well be forced to look at the code of the base class f to understand the effect of this call; in particular, the abstract specification may not provide us the required information.

If f is inherited from B rather than being (re)defined in D, then its (concrete) specification must also have been inherited from B as we saw in the last section. Since neither the body of f, nor its pre- and post-conditions have changed, there is no need to do any verification since this task has already been performed by the base class designer.[11] This reuse, in the verification of the derived class, of the verification performed in the base class matches exactly the code reuse that we have achieved by inheriting the code of f from B rather than defining it in D.

There is one situation where even if f and its (concrete) specification are inherited from B, we may still have to reverify it. This is the case of the *recursive* function we mentioned in section 2: suppose f calls another function g that is also a method of B. Suppose f is inherited unchanged into D but g is redefined. Suppose also that g is declared '*virtual*' in B.[12] In this situation if (in the client code) we apply f to an object that is an instance of D, the g that will be invoked during this call of f will be the one that is defined in D, not the one defined in B. Now, in verifying that f meets its specification as part of verifying the correctness of B, the designer of that class must, of course, have assumed that the function g invoked in the body of f would behave according to its specification in $\underline{B}$. If the redefined g's behavior is consistent with its specification in B (and this is effectively what [12] requires) then there would be no need to reverify f but this seems an unlikely scenario; why redefine g if its new behavior is going to be consistent with its old behavior?[13]

[11] In practice, of course, it is quite possible that the base class designer has not actually formally verified that the body of f meets its specification. So what we are doing here is trusting the base class designer to have implemented f correctly.

[12] In *Eiffel* all functions are virtual by default.

[13] One possibility is that g is redefined so its effects on the member variables of B are the same as before, but it has some effect also on the new member variables introduced in D. In this case it would seem that the behavior of the redefined g is consistent with its behavior in B; but even here we cannot inherit the specification of f from B because that specification does not capture the effect a call to f (applied to an instance of D) will have on the member variables introduced in D.

If g's redefined behavior is not consistent with its specification in B then there are two possibilities: First, we could simply require any f that invokes a (virtual) g that is redefined in D to be treated as if it was also redefined (with the body being textually identical to that in the base class). We may well have to come up with a new specification for f since its behavior is likely to be different than in B. Alternately, we could provide a complex specification of f, one that includes information about how and when it invokes g in such a manner that we could 'plug-in' the behavior of the redefined g into this specification to obtain the actual behavior of f (when applied to instances of D).[14] The advantage of the first approach is simplicity, but at the expense of some reuse in specification and verification. The second approach is quite complex but would probably be useful in situations, such as OO *frameworks*, where such functions are used heavily; for some preliminary ideas on the details of such an approach, see [10]. In the currentq paper we will assume that the first approach is used.

Next consider the second task we need to go through, that of verifying that each method of D leaves its invariant satisfied. Formally speaking, we need to establish, for each f the following:

$$I(\omega) \wedge c.pre_f(\omega) \wedge c.post_f(\omega, \omega') \Rightarrow I(\omega')$$

where I is the invariant on the state ω, $c.pre_f$ is the concrete pre-condition of f, $c.post_f$ its concrete post-condition (involving the initial state ω and the final state ω'). For simplicity, we have ignored the parameters of f.

Note that we do not need to refer to the *bodies* of the methods, only their concrete specifications. Is any reuse of the work done in the base class possible in this task? That depends on the relation between the invariants of B and D. If the invariant of D is identical to that in B, then we do not have to do anything in this step as far as functions inherited from the base class are concerned; for functions (re)defined in D (as well as for recursive functions whose concrete specifications have changed because one or more of the virtual methods they call have been redefined) we of course have to verify that they preserve the invariant.

If the invariant of D is stronger than that of B, i.e., it implies the invariant of B, then we can inherit part of step 2 for functions inherited from the base class. If f is such a function, then we have already verified that it preserves the base class invariant; hence all we have to do is to check that it also preserves the additional clauses of D's invariant. There is a special case that is worth mentioning here: if the only difference between D's invariant and B's invariant concerns the values of member variables introduced in D, we need do no work for step 2 for inherited functions since they do not access these additional variables.[15]

If the invariant of D is different from that of B, then clearly no reuse is possible. We will just have to verify from scratch that each method of D preserves its invariant.

Finally, consider step 3. We have to show, for each f, that an appropriate relation holds between the abstract pre-condition of f and its concrete pre-condition; and similarly for the post-conditions. Let us consider the pre-conditions:

$$[a.pre_f(\varepsilon(\omega)) \wedge I(\omega)] \Rightarrow c.pre_f(\omega)$$

ε is the abstraction function that maps the concrete state of the class to the abstract state. $a.pre_f$ is the abstract pre-condition of f; I is the (concrete) invariant.

This requires us to show that if the abstract version of a state ω satisfies $a.pre_f$, and if ω satisfies the invariant, then it also satisfies the concrete pre-condition of f. For post-conditions we have a similar requirement:

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

Note that both the concrete and abstract post-conditions are assertions involving both the initial state and

[14] Discussions of behavioral subtyping seem to suggest that it would be useful to require behavioral subtyping in such situations but, as we will see in section 5, that is true only if the client code can ignore the differences between instances of the base class and of the derived class.

[15] The reader may be wondering about the complementary case where the invariant of D is *weaker than*, i.e., is implied by, B's invariant. Unfortunately, no reuse is possible in this case. The problem is that what we have shown in the base class is that if the initial state when f starts execution satisfies B's invariant (and, of course, f's pre-condition) then the final state will satisfy B's invariant. When considering D, we can only assume that the initial state will satisfy D's invariant, and if that is weaker than that of B, no automatic conclusion can be drawn about whether or not the final state will satisfy D's invariant.

final state.

Is any reuse possible in this step? If f is (re)defined in the derived class, chances are its concrete pre- and post-conditions will be different from those in the base class and no reuse will be possible. But even if f is inherited from the base class, if the abstraction function in the derived class is not the same as in the base class, we won't be able to reuse the work from the base class in these steps. But if the abstraction function is the same, and if the invariant is the same, then no additional work is needed in the derived class (assuming, as is usually true if these conditions are satisfied, that the abstract pre- and post-conditions of f are also the same as in the base class); this will be the case in the example we consider in the next section. If the abstraction function is the same and the invariant is stronger, then again no work is needed since the stronger invariant appears on the left sides of the two implications above. If the invariant is weaker than, or different from, the invariant of the base class, no reuse will be possible in this step.

Thus the degree of reuse possible depends heavily on the amount of code that is being reused, i.e., in the number of functions being inherited unchanged; it also depends, especially in the final step, on the differences between the abstraction functions and the invariants of the base and derived classes. One point to note is that even in the final step which is concerned with the abstract specifications, reuse does not require the derived class to be a behavioral subtype of the base class. Behavioral subtyping would require *every* function f that is redefined in the derived class to have an abstract specification that is 'consistent' with that in the base class. No such requirement applies to the kind of reuse we are discussing.

4 Example

In this section we will illustrate our method with a simple example. Rather than giving complete details of the specifications and verification, we will focus on showing the general structure of our approach, and demonstrating the various levels of specification and verification reuse that we talked about earlier in the paper.

Consider a *BankAccount* class. The conceptual model of a *BankAccount* is a tuple of three values:

$$\mathsf{BankAccount} \equiv (\ \mathsf{balance} : \mathsf{real}\ ;\ \mathsf{history} : \mathsf{sequence\ of\ trans}\ ;\ \mathsf{month} : \mathsf{integer})$$

balance is the current balance in the account; *month* is the (number of the) current (calendar) month; *history* is the sequence of *trans*actions that have taken place so far in the current month, where each transaction consists of an amount and a date. The methods of the class will allow a client to perform new transactions (deposits and withdrawals) on the account, inspect the transactions that have taken place so far during the month, get the current balance, find out the starting balance for the month, and reset for the next month (which will set the *history* to the empty sequence $\langle\rangle$).

In the implementation of the BankAccount class sketched in *C++*-like style in Figure 1, we use an array to store the transactions and two real variables bal and sbal to store the current balance and starting balance, respectively.

```
class BankAccount {
protected:
      int mth;                  // the current month
      real bal, sbal;           // the current balance and the starting balance
      int size;                 // the number of transactions in the history
      trans his[MAXTRANS];
public:
      BankAccount(int m) { bal = sbal = 0.0; size = 0 ; mth = m }
      void AddTransaction(trans t) { bal += t.amt ; his[size++] = t }
      trans GetTransaction(int i) { return his[i−1] }
      int CurrentMonth() { return mth }
      real CurrentBalance() { return bal }
      real StartingBalance() { return sbal }
      void NextMonth() { sbal = bal ; size = 0; mth++ }
}
```

Figure 1: Base class BankAccount

The number of transactions so far in the current month is stored in size. Note also that all member variables are declared *protected* so that any derived classes we decide to build can have access to them.

AddTransaction is the function that lets us apply another transaction to the account. GetTransaction returns information about a transaction earlier in the month. CurrentMonth returns the value of the current month, CurrentBalance gives us the current balance in the account. StartingBalance tells us what the balance was at the *start* of the month. NextMonth resets for the next month.

Next, following the approach described in sections 2 and 3, we should provide abstract and concrete specifications for the class. We have already specified the conceptual model of the class. What remains of the abstract specification is the pre- and post-conditions for each operation. Let us consider one of these functions, AddTransaction:

$$\mathit{abs.pre.AddTransaction(trans\ t)} \equiv \mathit{balance} + \mathit{t.amt} \geq 0$$

$$\mathit{abs.post.AddTransaction(trans\ t)} \equiv (\mathit{balance'} = \mathit{balance} + \mathit{t.amt}) \wedge (\mathit{history'} = \mathit{history} * <t>) \wedge (\mathit{month'} = \mathit{month})$$

where $*$ represents concatenation of sequences, and the notation var' in a post-condition refers to the value of var in the state immediately after the method completes. The pre-condition here states that a withdrawal greater than the balance is not allowed. The post-condition states that the balance is updated, and the transaction added to the *history*; *month* is unaffected.

Next we consider the concrete specification of the class. The abstraction function ε is easy: it maps bal to *balance*, mth to *month*, the first size elements of the his array to the abstract *history* sequence, and throws away sbal. The invariant assures us that size does not become negative, and that the values of sbal and bal are consistent with each other given the current value of his

$Inv \equiv (\mathit{size} \geq 0) \wedge (\ \mathit{bal} = \mathit{sbal} + \sum_{i=0}^{size-1} \mathit{his}[i].\mathit{amt}\ )$

Note that Inv is critical to showing that the implementation of StartingBalance is correct. Since the starting balance of the month is not a part of the abstract state, the abstract post-condition of StartingBalance would be written in terms of *balance* and *history*.

$\mathit{abs.post.StartingBalance()} \equiv (\mathit{returns\ balance}\ - \sum_{j=1}^{length(history)} \mathit{history}_j)$
$\quad\wedge\ (\mathrm{balance}' = \mathrm{balance}) \wedge (\mathrm{history}' = \mathrm{history}) \wedge (\mathrm{month}' = \mathrm{month})$

However, the *concrete* post-condition of StartingBalance will simply state that the value of sbal is returned. It is only with the information that is provided by Inv, namely that in all concrete states reachable by this class, the value of sbal is consistent with the current balance and the transaction history, that we will be able to conclude that the abstract post-condition is valid. This reasoning would be part of step 3 (from section 3) for the verification of *BankAccount*.

Next let us consider the concrete specification of the AddTransaction function:

$\mathit{con.pre.AddTransaction(trans\ t)} \equiv \mathit{size} \geq 0$

$\mathit{con.post.AddTransaction(trans\ t)} \equiv\ (\mathit{bal}' = \mathit{bal} + \mathit{t.amt}) \wedge (\mathit{his}' = \mathit{his}([\mathit{size}] : \mathit{t}) \wedge (\mathit{sbal}' = \mathit{sbal}) \wedge$
$\quad(\mathit{size}' = \mathit{size} + 1) \wedge (\mathit{mth}' = \mathit{mth})$

where $\mathrm{his}([i] : val)$ is the same as the array his with the item which was originally at position i replaced by val. Note that the concrete pre-condition of AddTransaction does not rule out negative balances in the account. There is no need to do so because the code of the function does not depend on it. Not including this in the concrete pre-condition means a later derived class that wants to allow, in the conceptual model, negative balances will be able to do so.

However, we will consider a different derived class, one in which we want to add an operation that will let us cancel a preceding transaction; perhaps this operation corresponds to a new facility provided to the bank customers – that of canceling fraudulent transactions. Let us name this operation RemTransaction, and the new class BetterAccount.

We could of course implement BetterAccount from scratch but that would be a crime against the reuse philosophy given that we can implement it incrementally by using inheritance from the existing BankAccount class. It is also worth noting that we cannot implement BetterAccount by *layering* on the BankAccount class since that class does not permit transactions to be selectively removed from the history.

The implementation of BetterAccount as a derived class of BankAccount appears in Figure 2.

```
class BetterAccount : BankAccount {
public:
    void RemTransaction(int i) {    bal −= his[i−1] ; size−− ;
                                    // Shift each element of his from i - 1 to size - 1 down one. }
}
```

Figure 2: Derived class BetterAccount

The RemTransaction function removes the appropriate transaction from his, shifts the remaining transactions one element forward in his, and updates bal appropriately. Note that the code of RemTransaction would not have compiled if we had declared bal and his as *private* data members. In that case we would be in the same position as if we had tried to *layer* the implementation of BetterAccount on top of BankAccount.

Next we must specify and verify BetterAccount. Much of the specification of the base class can be inherited. Let us first consider the abstract specification. The conceptual model of the class is the same as for BankAccount class. The specifications of all the functions inherited from that class are also the same. The only additional thing we need to provide is the abstract specification of the newly introduced method, RemTransaction:

$$\mathit{abs.pre.RemTransaction(int\ i)} \equiv 0 < i \leq length(\mathit{history})$$

$$\mathit{abs.post.RemTransaction(int\ i)} \equiv (\mathit{month}' = \mathit{month}) \wedge (\mathit{history}' = \mathit{history}_{1\ldots i-1} * \mathit{history}_{i+1\ldots length(history)}) \wedge (\mathit{balance}' = \mathit{balance} - \mathit{history}_i.amt)$$

The post-condition states that the transaction has been removed from the history and the balance restored appropriately.

Now consider the concrete specification. The abstraction function is identical to its counterpart in the base class, as is the invariant. The concrete specifications of all the functions defined in the base class can be inherited since none of them is redefined. The only thing we need to do to complete the concrete specification is to specify the pre- and post-condition of the new operation:

$$\mathit{con.pre.RemTransaction(int\ i)} \equiv (i > 0) \wedge (i \leq \mathit{size})$$

$$\begin{aligned}\mathit{con.post.RemTransaction(int\ i)} \equiv\ & (\mathit{bal}' = \mathit{bal} - \mathit{his}[i-1].\mathit{amt}) \wedge (\mathit{sbal}' = \mathit{sbal}) \wedge (\mathit{size}' = \mathit{size} - 1) \wedge \\ & (\mathit{mth}' = \mathit{mth}) \wedge (\forall j, 0 \leq j < \mathit{size}-1, (j < i-1 \Rightarrow \mathit{his}[j]' = \mathit{his}[j]) \wedge \\ & \qquad (i-1 \leq j < \mathit{size}-1 \Rightarrow \mathit{his}[j]' = \mathit{his}[j+1]))\end{aligned}$$

The post-condition asserts that the method subtracts the amount of the transaction to be removed from the balance, decrements size, and for each array element after the one to be removed, copies it to the position immediately below it.

This is a remarkable degree of specification reuse that handily matches the code reuse. One might be tempted to argue that the reason for this is that BetterAccount is a (weak) behavioral subtype [2] of BankAccount.[16] But we will shortly mention another equally simple example which is not a behavioral subtype, strong or weak, of BankAccount for which also we will be able to inherit much of the base class specification.

Next let us consider the verification task. As we saw in the last section, there are three steps in the verification task. Here again we will be able to inherit almost all of the work done in the base class.

Step 1

Recall that in this step we verify that all the methods meet their concrete specification. Since all the methods of the base class have been inherited unchanged, our approach allows us to skip the verification of their method bodies. This is the most basic level of verification reuse allowed by inheritance. The only thing that has to be done for this step is to verify that the newly defined method RemTransaction meets its concrete specification. We will leave that to the interested reader.

Step 2

In this step we have to verify that each method, as per its pre- and post-conditions, maintains the invariant of the class. But as we saw in specifying the class, the invariant is the same as in the base class, and so, as we saw in section 3, there is no work to be done as far as the inherted methods are concerned! This is the second level of verification reuse that is often possible with inheritance. That only leaves us with the

[16] BetterAccount is not a behavioral subtype of BankAccount according to the definition of Liskov and Wing [6, 7], but it is a *weak* behavioral subtype, as per the definition of Dhara and Leavens [2].

task of verifying that RemTransaction preserves the invariant. This is easily done. Suppose Inv holds in a state ω. This asserts that the sum of sbal and each of the transaction amounts in the array his is equal to bal. RemTransaction removes one transaction from the his array, effectively subtracting that amount from the sum of the transactions. However, at the same time it subtracts the same amount from bal, thereby maintaining the original equality.

Step 3

The final step is to check that, for each method, the abstract specification and concrete specification are consistent with each other. Again, since the abstraction function as well as the invariant are the same in the base and derived classes, the verification procedure described in section 3 approach allows us to reuse the verification work done in the base class for all of the inherited functions. This is the third level of verification reuse that we can obtain in the verification of a derived class. All that remains for us to do is to check that the relations specified toward the end of section 3 between the abstract and concrete pre- and post-conditions of RemTransaction hold.

To see this, note that the abstract pre-condition requires that *i* is a number anywhere from one to the length of the *history* sequence in the current state. ε maps the first size elements of the array his to the *history* sequence, and thus size represents the length of the sequence. This means that the abstract and concrete pre-conditions are identical. The concrete post-condition states that the *i*th element of the his array is subtracted from bal and removed from the array. Since ε maps bal to *balance* and, as we saw above, the elements of his to *history*, this condition is identical to the abstract post-condition.

That completes the verification of the derived class. Again, as in the case of specification reuse, the degree of verification reuse is remarkable and easily matches the degree of code reuse.

Before concluding this section, let us briefly consider another possible derived class of BankAccount. Suppose the bank decides to impose a per transaction service charge c. How do we implement this? Again we can implement it as a derived class, let us call it WorseAccount, of the BankAccount class. This time we cannot inherit all the functions from the base class. In particular we have to redefine the AddTransaction function so that it reduces the final balance by an amount equal to the per transaction fee. The remaining functions can be inherited without change.

```
class WorseAccount : BankAccount {
public:
    void AddTransaction(trans t) { BankAccount::AddTransaction(t); bal −= c; }
}
```

Figure 3: Derived class WorseAccount

Let us briefly consider the specification of this class. The conceptual model of this class is again the same as in BankAccount class. The abstract specifications of all the operations are also the same, except for the StartingBalance operation and the redefined AddTransaction. The concrete specification also can be mostly inherited from the base class; the abstraction function is the same, as are the pre- and post-conditions of all the inherited operations. The invariant is different because we have to allow for the transaction fee when considering the relation between bal and sbal.

What about the verification? Because the invariant is different, we can expect only limited reuse in steps 2 and 3. There is considerable reuse in step 1 because of all the inherited operations. Even for the redefined operation, because it invokes the base class operation, the verification task is simplified since we do not have to go through the body of th base class operation, although we are making use of it. Thus although the situation is not as good as in the case of BetterAccount, there is still considerable reuse in both specification and verification in this example also. And this reuse is possible although WorseAccount is not a behavioral subtype, weak or strong, of BankAccount.

5 Discussion

Discussions of inheritance often use the term '*is-a*' to describe it, the idea being that every object that is an instance of the derived class D is also an instance of the base class B. While at the informal level this is a useful way to describe inheritance, imposing this as a formal requirement can ham-string a designer. Consider our example of bank accounts. While intuitively instances of the derived class WorseAccount can be considered as BankAccount objects, from a formal point of view they are not, since, for instance, the behavior of the AddTransaction operation, when seen from a client's point of view, is different in the two classes. This is why we believe that imposing a formal version of '*is-a*', such as (strong or weak) behavioral subtyping, will outlaw many reasonable uses of inheritance.

But at the same time we do not want inheritance to be merely a code reuse mechanism. Unless the code reuse is matched by reasoning reuse, system designers would gain little by using inheritance. That is the challenge we have tried to meet in this paper: To design a specification notation and verification procedure that will allow the designer of the derived class to reuse as much as possible of the specification and verification that has been performed for the base class. The key insight behind our approach is that the reuse that inheritance enables is (usually) not at the abstract level, but rather at the concrete level. That is not to say that the designer looks at the *code* of the individual methods and decides to reuse bits and pieces of them. Rather, once he understands the effects of the various methods on the member variables of the class, he can see which can be reused and which need to be redefined, and what additional methods (and variables) need to be introduced, in order to arrive at a class that will serve his current purposes. Since this is the information contained in the concrete specification of the class, once he has access to this specification, the designer can design the derived class without having to study the base class code. This is particularly important in situations where the designer does not have access to the base class code, as might be the case if the base class was purchased from a software vendor who may not have provided the source code for the class.

This is perhaps the most important difference between behavioral subtyping [6, 3, 2] and our work; whereas the work on behavioral subtyping focuses on the relation between the abstract specifications of classes, our work shows how the derived class designer can use the concrete specification of the base class to build derived classes that are similar to but not (necessarily) behavioral subtypes of the base class; and how, in the derived class, he can reuse much of the specification and verification that has been performed for the base class. A figure may make the relations between the various specifications clearer. In Figure 4, B as usual is the base class and D the derived class. The abstract specifications are represented by boxes with dashed lines, concrete specifications by boxes with solid thin lines, and implementations by boxes with solid heavy lines. The vertical lines joining the boxes represent the various steps that we described in section 3 that we have to go through in establishing that a class meets its specifications.

Inheritance is essentially a programming mechanism that the designer uses to construct the implementation of D by reusing (possibly a part of) the implementation of B. The horizontal line joining the boxes corresponding to these implementations represents this code reuse. The two horizontal lines above that one represent the verification reuse and the specification reuse that our approach enables. The line labeled 'specification reuse' represents the fact that the concrete specifications of methods of D that are inherited from B are the same as in B. The line labeled 'verification reuse' represents the fact that these methods do not have to be reverified; this line also represents the verification reuse we gain when verifying methods that are (re)defined in D, but invoke methods of B.

If further, we abide by the requirement that the derived class must be a behavioral subtype, then the abstract specifications of B and D will have the relation described by, for instance, Liskov and Wing [6, 7]; this is represented by the top most horizontal line in the figure, joining the boxes corresponding to the absract specifications of B and D. But this line doesn't really represent a reuse of specifications because in general there is no guarantee that (the abstract specification of) D will be a behavioral subtype of (the abstract specification of) B. Edwards [3] and Dhara and Leavens [2] consider restrictions on how inheritance is used to ensure that the behavioral subtype relation holds. Our work shows that even if these restrictions are not satisfied, we have considerable specification and verification reuse, represented by the two intermediate

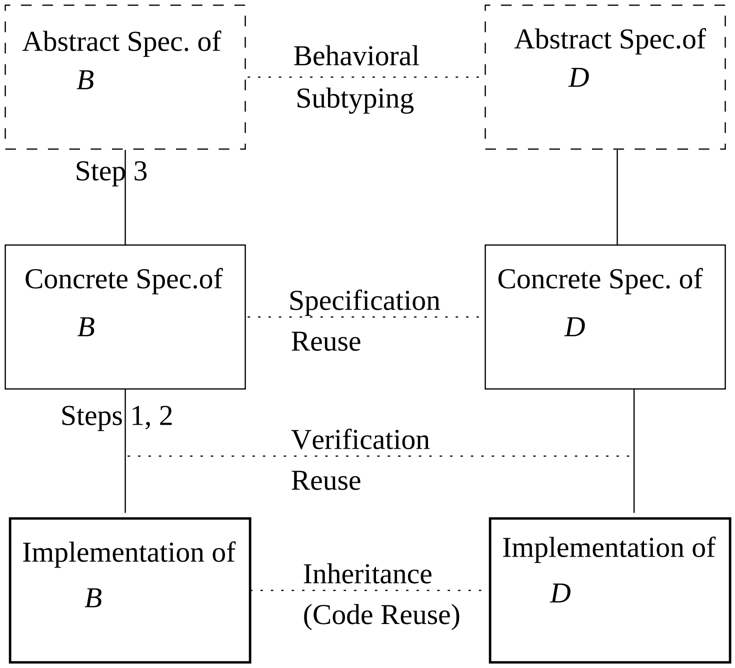

Figure 4: Levels of Reasoning Reuse

horizontal lines. Of course, if the derived class designer redefines every single method of the base class, we will not be able to reuse any part of the specification (or verification) of B; but then in this case, there was no code reuse either. Thus the degree of specification and verification reuse that our approach allows closely parallels the degree of code reuse.

As we noted earlier in the paper, the motivation underlying behavioral subtyping is different from the motivation underlying our work. Our work was motivated by the desire to match, in the task of reasoning about the derived class, the code reuse that the *derived class designer* achieves in inheriting from the base class. Behavioral subtyping, on the other hand, tries to promote reuse in the task of reasoning about the code of the client that uses these classes. The idea is that if the client has verified that his code will meet its specification if we use objects that are instances of a class B, then the code will also work correctly if we instead use objects that are instances of a class D, provided D is a behavioral subtype of B. But note that this assurance comes at a price: we must be willing to ignore the differences between these classes. Dhara and Leavens [2] call this the '*information loss*' problem since we are losing information about this difference. This might be acceptable if, for instance, the difference between the two classes was essentially with respect to performance. Another situation where we may need to be able to do this, as Liskov and Wing point out, is if we have an object that is used by two different clients, one of whom thinks it is of type B, and the other thinks it is of type D. If however, the client needs to be able to distinguish between objects of type B and D, it would seem that we cannot make use of behavioral subtyping. In all cases, however, if D is defined by inheritance from B, our approach to specifying and verifying the classes will apply.

We will conclude with a brief note about *multiple inheritance*. Although we have focused on single inheritance in the paper, it would be straightforward to extend our approach to deal with classes that are defined by inheritance from more than one base class. Suppose $B1$ and $B2$ are two base classes and D is defined by inheritance from both. To keep the discussion simple, let us assume there are no name clashes etc. If f is a function defined in $B1$ and inherited into D, then we also inherit the concrete specification of f from $B1$; we do have to strengthen this specifcation in the same manner as described in section 2, by adding a clause to the post-condition stating that the values of member variables introduced in D are the same at the end of f as at the start; and another similar clause stating that f does not affect the values of member variables of $B2$. Thus the specification of f is essentially inherited from $B1$; no re-verification is necessary either since the body of f is not being changed. If f was inherited from $B2$ instead, we would have a similar situation. One important question would have to do with the invariant for D. The most useful case, and one that would enable the greatest degree of reasoning reuse, would be if D's invariant implied the invariants of both $B1$ and $B2$. We will not go into further details here, but it should be clear that the approach extends easily and naturally to handle multiple inheritance. And the specification and verification reuse that we achieve matches the code reuse the designer has achieved by inheriting from $B1$ and $B2$.

References

[1] P. America. Designing an object oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, LNCS 489, pages 69–90. Springer-Verlag, 1991.

[2] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance.

[3] S. Edwards. Representation inheritance: A safe form of 'white box' code inheritance. In *Software Reuse*, pages 27–51. Springer-Verlag.

[4] J. Guttag, J. Horning, and J. Wing. The larch family of specification languages. *IEEE Software*, 2, 1985.

[5] C. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1990.

[6] B. Liskov and J. Wing. A new definition of the subtype relation. In *ECOOP*, 1993.

[7] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16:1811–1841, 1994.

[8] R. Martin. *Designing object oriented C++ applications using the Booch method*. Prentice-Hall, 1995.

[9] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[10] N. Soundarajan. Understanding frameworks. Technical report, Ohio State University, 1997.

[11] N. Soundarajan and S. Fridella. Inheriting and modifying behavior. In *TOOLS*, 1997.

[12] R. Stata and J.V. Guttag. Modular reasoning in the presence of subclassing. In *OOPSLA*. ACM Press, 1995.