Enriching Subclass Specifications

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

October 9, 1998

Abstract

The standard approach to reasoning about *polymorphic* (also called *recursive* or *self*) methods is in terms of method refinement. That is, if the correctness of a (polymorphic) method f of class B depends on certain properties of another method g of B, then in any subclass of B in which g is redefined, the designer must make sure that g continues to exhibit these properties. While this allows the designer of the derived class to avoid re-verification of f, it does not allow us to associate, in the derived class, richer behavior with the polymorphic function. We propose an approach that, by including somewhat more information in the specification of f, enables us to arrive, in the derived class, at a richer specification of the function while at the same time avoiding re-verification. The approach is illustrated on a simple example.

1 Introduction and Motivation

One of the most important ideas underlying the Object Oriented approach is *polymorphism*.[1] Polymorphism enables a derived class designer, assuming that the base class has been suitably designed, to construct interesting and varied derived classes by just redefining an (appropriate) small set of functions of the base class, since not only will the redefined functions exhibit new behavior, but so will other functions of the base class that call one or more of the redefined functions. Much has been written about the flexibility and power that OO polymorphism affords to the (derived) class designer to build classes that exhibit the behaviors that he[2] requires; see Meyer [6], for instance. But it is not enough to be able to code these classes; the designer also needs reliable methods that he can use to reason about the behavior of the resulting derived classes, and it is the goal of this paper to investigate such methods.

An important requirement that any such reasoning method should satisfy is that when arriving at the behavior of the derived class, the designer should not have to (re)analyze the code of those functions of the base class that are inherited unchanged. If this requirement is not met, while polymorphism may still be usable as a programming tool, its value in developing large systems would be severely limited since each derived class will have to be understood from scratch. And in those situations where the designer does not have access to the source code of the base class, as may be the case if it was purchased from a software vendor, the incrementality requirement is not just desirable, it is indispensable.

There are two issues we need to address if we want to be able to avoid reanalysis of inherited methods. Suppose p is such a method and that it invokes another method v that *is* redefined in the derived class. In

[1] Throughout this paper by 'polymorphism', or occasionally 'OO polymorphism', we mean the *inclusion polymorphism* of Cardelli and Wegner [1].

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

other words, p is a *polymorphic* method and v a *virtual* method[3]. If the redefined v did not exhibit some behavior exhibited by the base class v, and this behavior was used when we established the behavior of p in the base class, then clearly p will have to be reanalyzed. This problem can be addressed by requiring that the redefined v must exhibit behavior that is in some sense consistent with its base class behavior. This is essentially the approach of authors such as Stata and Guttag [9] and we will use the same approach in our system.

The second issue, and the focus of this paper, is that it is not enough to ensure that p, in the derived class, will continue to exhibit the behavior we have established for it in the base class. As we said earlier, the derived class designer expects the p to exhibit *richer* behavior than its base class counterpart; that is the *raison d'être* for using polymorphic functions. So in a reasoning system we must have a means for arriving at a richer specification of p based on the richer behavior exhibited by v. The most common reason for p to exhibit richer behavior is that new member variables have been introduced in D and the redefined v manipulates one or more of these variables. As a result, although these variables did not even exist when p was written (or even when it was compiled), a call to p may well also result in changes to the values of some of these new variables since during its execution p will invoke the redefined v. We will see precisely such behavior in our examples later in the paper. If our reasoning system only guaranteed that the behavior established for p in the base class continues to be valid, we will not be able to account for how a call to p affects these new variables (unless we resort to reanalyzing p).

The key to our approach is to associate when dealing with the base class, two specifications with polymorphic functions such as p; the first, called *functional* specification, is for use by clients of the class and the second, called *specialization* specification, is for use by the derived class designer[4]. The functional specification, as may be expected, simply describes the effect of a call to p as seen by a client of the class. The specialization specification will contain this information but in addition will specify which virtual functions p invoked during its execution, and in what order, as well as a characterization of the relation between the final state when p finishes execution and the effects of the various virtual function calls on the state. As we will see in sections 2 and 3, the derived class designer will be able to use the specialization specification of p and a characterization of the behavior of the redefined v to establish a richer functional specification of p than the one in the base class. In particular, this new specification of p will characterize the effect of p on the new member variables of the class.

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

- Makes the case that if Object-Oriented polymorphism is to be truly useful in building large systems, it is necessary to have a reasoning method that will allow us to establish richer behaviors of polymorphic functions that result from the richer behaviors exhibited by the redefined virtual functions that they invoke; and that this reasoning system should not require us reanalyze the code of the polymorphic function.

- Introduces a specification notation, in particular the specialization specification of polymorphic functions, that provides the derived class designer exactly the needed information to arrive at the richer behavior of these functions; and a reasoning method to establish these specifications without reanlysis of the code of the (inherited) polymorphic functions.

- Applies the specification notation and reasoning procedure on a simple example; and briefly considers the applicability of the approach to more complex situations.

The rest of the paper is organized as follows: In the next section we consider how base classes and derived classes may be specified. In particular, we consider the specialization specifications, in the base class, of polymorphic functions and how such specifications can lead to stronger functional specifications in the derived class. In the third section we present the formal reasoning procedure that allows us to establish

[3] We will use *C++*-like terminology and notation but our approach is generally applicable to all class-based OO languages such as *Simula*, *Eiffel*, *Java*, etc; also, we will use the terms function and method interchangeably.

[4] [2] as well as [9] also use the term specialization specification but in a different sense than we do and in a different sense than each other. We will consider their notions of specialization specification in the final section.

the specifications; we should hasten to add that we are not advocating that every specification be formally verified using the rules presented in section 3. Indeed in section 4 where we consider our example(s) we often take the liberty of arguing informally in showing that the specifications are correct. In the final section we summarize our work, explains how it extends previous work in important ways, and consider how the approach may be used for dealing with more ambitious uses of polymorphism such as in OO frameworks.

Before concluding this section we should note two points. First, we will ignore abstraction issues in this paper. That is not because we think clients of classes should know the internals of a class, but rather that these issues are fairly well understood and we could use standard techniques such as abstraction functions [5] to relate abstract specifications with the functional specifications we write. Note that the specialization specification must of necessity be concrete since it is for use by the derived class designer.

Second, polymorphism can also be used by client programmers; such usage of polymorphism also raises specification and verification questions; these are somewhat different from the ones raised by the use of polymorphism in classes which is the focus of the current paper. For a possible approach to dealing with client code use of polymorphism, see [8].

2 Class Specifications

In section 2.1 we will consider how a base class B with different types of methods (virtual, polymorphic, as well as 'normal', i.e., neither virtual nor polymorphic) may be specified. The most interesting of these will of course be the polymorphic functions and we will focus our attention on these both here and in section 3. In 2.2 we will consider a derived class D of B and see what the specifications of the methods of D will look like, and how they are related to those of B.

2.1 Base Class Specifications

Consider a class B that is defined from scratch, i.e., does not inherit from any other class. In order to simplify the presentation we will make the following assumptions (in the final section we will briefly consider how restrictive the assumptions are and how they may be relaxed):

- All member variables of B are *protected* (so they are accessible to the derived classes of B) and are of simple types such as integers, characters, etc.
- The *virtual* methods of B do not invoke each other, nor do they invoke the 'polymorphic' methods of B (see below); they may invoke the 'normal' methods (see below).
- The *normal* methods of B are non-virtual; they may invoke each other but not any of the virtual methods nor any of the polymorphic methods.
- The *polymorphic* methods of B are also non-virtual; they may invoke the virtual methods as well as the normal methods, but not each other.
- Each method may have parameters (in addition to having access to the member variables of the class); the parameters will be passed by *value-result* and will, like the member variables, be of simple types.
- We will only consider single inheritance.

The specification of the class B will just consist of the specifications of its individual methods. As noted before our specifications will be concrete, i.e., in terms of the member variables of the class, not in terms of an abstract model of the class. A class invariant is also often useful but we will not introduce one in the interest of simplicity. Each method will be specified by an appropriate pre-condition and post-condition.

The pre-condition of a normal method n will be an assertion involving the state of the member variables of B, which we will henceforth abbreviate to 'state', at the time n is called as well as the input values of

n's parameters if any. The post-condition will be an assertion involving the state at the time of the call, the state at the time n finishes execution, and the input values and final values of n's parameters. Allowing the post-condition of a method to refer to the situation at the time of the call as well as when the method finishes (rather than just the latter) makes it easy to specify certain methods but it is not essential to our approach.

Next consider a virtual method v. Interestingly, this is also specified in exactly the same manner as a normal method. The reason for this is that virtual methods may not call other virtual (or polymorphic) methods. As a result if a derived class D inherits v then its behavior will be identical to what it is in B even if other virtual methods are redefined in D.

But that is not true of polymorphic methods. If a polymorphic method p is inherited by D, then its behavior will (or can) be different from what it is in B if any of the virtual functions that p invokes is redefined in D. Indeed, our main goal as we explained in section 1 is to try to be able to arrive at this new behavior of p. But we do not want to do this by reanalyzing p; if we were willing to reanalyze p, we could arrive at a new specification for it on the basis of the behavior exhibited by the redefined virtual functions. Instead, we will enrich the specification of p (in B) to include information about which virtual methods it invokes as well as how its behavior depends upon the behaviors of these virtual methods.

We will introduce an (auxiliary) *trace* variable τ_p to record the sequence of calls that p makes to virtual methods. When p begins execution, τ_p will, of course, be the empty sequence since p has not yet made any such calls. As p executes, each time it makes a call to a virtual method v, we will append an element of the form $(v, \#\sigma, \#x, \sigma, x)$ to τ_p, where v is the name of the method being invoked, $\#\sigma$ the state at the time that v is called, $\#x$ the input value(s) of the parameter(s) of v, σ the state when v finishes execution and returns to p, and x the output value(s) of the parameters. The post-condition of p will be an assertion involving the state at the time p started, the state at the time p finished execution, and the input values and final values of p's parameters, as well as the value of the trace τ_p when p finishes, this being the record of calls that p makes to the virtual methods. Given this post-condition, we will be able (as we will see later) to arrive –without reanalysis– at the behavior that p will have in the derived class D if one or more of the virtual functions that it invokes is redefined in D.

We will call the specification described above the *specialization* specification of p since it will allow the derived class designer to arrive at the behavior of p corresponding to redefinitions of the virtual methods that p calls. In addition, we will also associate a *functional* specification of p; this will be identical to the specialization specification except it will not contain the information about τ_p. A client of the class would only be interested in the functional specifications of the various methods.

Consider a simple if artificial example; we will consider a more interesting example in section 4. Suppose we have a class with an integer member variable s, a polymorphic function p, and a virtual function v. Consider the following specifications of v and p:

$$pre_v \equiv true$$
$$post_v \equiv (s = \#s + 1)$$
$$pre_p \equiv true$$
$$post.s_p \equiv (\tau_p = \langle (v, \#s, \#s + 1) \rangle \wedge s = \#s + 2)$$
$$post.f_p \equiv (s = \#s + 2)$$

Note that since v does not have parameters, the elements of τ_p only have three components.

The specification of v says that it simply increases the value of s by 1. Any redefinition of v in a derived class will be required by our formalism to continue to satisfy this specification. Why would we want to redefine v in D? Suppose there is a new member variable t in D. We can then redefine v to manipulate t in any manner we choose so long as v continues to increase s by 1.

$post.s_p$, the specialization specification of p says that it calls v once, and increases the value of s by 2. Note that this specification does *not* say that p calls v and then increases s by 1. That may be how it is implemented but we cannot conclude that from $post.s_p$. For all we know (from $post.s_p$), p may save the value of s in some local variable (i.e., a variable declared in p), call v, restore s back to what it was before the call to

v (in other words, erase the effect on s of the call to v), and then increase s by 2. We can however, conclude from the clause $\tau_p = \langle(v, \#s, \#s+1)\rangle$ that the value of s when p invokes v is the same as it was when p started execution. It would be easy to write a weaker $post.s_p$ that would not specify this: we would just have to say that τ_p contains a single element corresponding to an invocation of v without saying anything about the value of s at the time of this invocation.

Even if we weakened the specialization specification in this manner, the functional specification $post.f_p$ could still have been the same. This simply asserts that from the client's point of view the result of calling p is to increase s by 2. In 2.2 we will see how the two possible specialization specifications could have resulted in different *functional* specifications of p in the derived class.

2.2 Derived Class Specifications

In general, in the derived class D we can not only redefine the virtual functions of B but also provide new definitions for the non-virtual functions of B. These would then override the definitions of these functions provided in B. We will not consider such redefinitions of non-virtual functions in this paper since our interest in this paper is in investigating how to allow for the changed behavior of a polymorphic function that is inherited unchanged from B, as a result of redefinitions of virtual functions that it invokes. (For possible ways of dealing with non-virtual functions being overridden, please see [7].)

A common reason for providing new definitions for virtual functions is that the derived class D contains new member variables and the designer of D wishes to redefine one or more of the virtual functions to manipulate these variables. For instance, suppose in the example we considered at the end of 2.1 that we introduce a new member variable t in D, and suppose we wish to redefine v so that it increases the value of t by 1. The new specification of v would then be:

$D.pre_v \equiv true$
$D.post_v \equiv (s = \#s + 1 \land t = \#t + 1)$

We have prefixed the pre and post-conditions with D to make it clear that this is the specification in the class D of this method. Note that this specification of v is stronger than its base class specification. That is important, since the base class specification of any polymorphic method such as v that invokes v may not be preserved if this were not the case.

But, as we have said several times, we want to *strengthen* the specification of p, not just preserve it. The stronger specification of p, given the redefined v is:

$D.pre_p \equiv true$
$D.post.s_p \equiv (\tau_p = \langle(v, (\#s, \#t), (\#s+1, \#t+1))\rangle \land s = \#s + 2 \land t = \#t + 1)$
$D.post.f_p \equiv (s = \#s + 2 \land t = \#t + 1)$

The new functional specification of p asserts that when p finishes not only will the final value of s be 2 greater than when p started, but also that the final value of t will be 1 greater than when p started. Note that we also have a new specialization specification of p. This will allow us to arrive at a new specification of p in any derived classes of D in which v is further redefined, in a similar manner.

Let us consider a slightly different situation. Suppose that the redefined v sets t to the value of s when v starts, then increments s by 1, and then finishes execution. Then using the information in the $post.s_p$ we can arrive at $D.post.f_p$ that says that the value of t when p finishes is equal to the value of s when p started. But this depends upon the information contained in $post.s_p$ that when p invokes v, the value in s is the same as when p started. If $post.s_p$ did not include this information, as was the case in the weakened $post.s_p$ we considered towards the end of section 2.1, we would not be able to establish this as part of $D.post.f_p$. Thus if the specialization specification in the base class of a polymorphic function p leaves out some information (about the elements in the trace of p), that may well affect the functional specification we can arrive at in the derived class for this function. We will return, in the final section, to the question of how a base class designer can ensure that he has included adequate information in the specialization specification of his polymorphic functions.

3 Class Verification

In this section we will define our formal reasoning method which will allow class designers to establish the correctness of the specifications we introduced in the preceding section. For methods which we have termed *normal* and *virtual* no new apparatus is necessary, and we assume that class designers will use standard axiomatic techniques to establish the correctness of their specifications. For *polymorphic* methods, we need to deal with two issues that are unique to our approach. First, when verifying (in the base class) that the body of a polymorphic method meets its pre- and post-condition, we need rules that ensure that each call to a virtual method is appropriately recorded in the trace for that method. Secondly, in the derived class designer we need a rule that will allow us to establish the new, richer specification for the inherited polymorphic method, using *only* the specialization specification of the method from the base class and the new specifications of the appropriate virtual methods from the derived class. We will consider each of these issues in turn.

3.1 Polymorphic Method Verification

To verify that a polymorphic method meets its specification we will use a standard Hoare-like logic, with a few special modifications that we will describe presently. Consider a method f from a class C which is polymorphic. That is, f invokes virtual methods $g_1 ... g_n$ from class C. The specialization post-condition of f will be an assertion over the state immediately prior to the invocation of f, the state immediately following the execution of f, and the trace[5] τ of virtual method calls made during the course of f's execution. Therefore, when reasoning about the intermediate states during the execution of f, a designer must be able to reason about not only the state of the object and local variables, but also the current value of τ. For this reason the assertions in our Hoare triples will be over three separate components:

- σ – which is a mapping from the data members of C to their current values.
- λ – which is a mapping from the local variables and parameters of f to their current values.
- τ – which represents the current value of the virtual method call trace.

The post-condition of f refers to the *initial* states of the object and the parameters, so these values must be included in σ and λ. In fact, for each data member d of class C, σ will contain an auxiliary constant $\#d$, and for each parameter p of f λ will contain an auxiliary constant $\#p$. In our model, these constants are initialized with values from the appropriate variables at the beginning of the execution of f, and are not otherwise affected during the execution of f. In this way, any intermediate assertion that we write when reasoning about the body of f can refer to the initial states of the object and parameters. Below, we will use the notation $\#\sigma$ to mean the state where the value of each member variable d of C is $\sigma(\#d)$. Similarly, the post-conditions for $g_1 ... g_n$ will refer to the state of the object and parameters before and after a call. To deal with this we introduce more auxiliary variables into the mappings σ and λ. First, each method call *self.g*(p) in f is assigned a unique label l. Then σ contains the unique auxiliary variables d^l for each data member d of C while λ contains a p^l for each parameter p of g. In our model, each such unique auxiliary variable, say d^l, is assigned the value of its corresponding regular variable immediately before the execution of the method invocation corresponding to the label l, and is not otherwise changed during the execution of f. Thus, any assertion following a statement $self.g(p)$ with label l can refer to the state of, say, p immediately before the execution of the statement by the notation p^l. We will use the notation σ^l to refer to the state where for each variable i, its value is replaced by $\sigma(i^l)$.

Of course we must also specify how the designer can reason about the current state of τ as various statements execute. In our model, the value of τ is changed only when a virtual method is invoked upon the object $self$ as in the statement, $self.g(p)$. The effect of this statement, beyond any effects the execution of g may have, is that τ is appended with an element of the form (g, ω, ω'), where ω is the composition of

[5] We will use τ rather than τ_f since we will be considering only one polymorphic function here.

the values of σ and λ immediately before the method call, and ω' is the composition of those same values immediately after the method has finished execution.

All of the above behavior is captured in the following rule for formally reasoning about the call to a virtual method g within a polymorphic method f. We will refer to this as the *virtual method call rule*:

$$\frac{P(\sigma,\lambda,\tau) \Rightarrow pre_g(\sigma,\lambda(p))}{\{P(\sigma,\lambda,\tau)\} \;\; self.g(p) \;\; \left\{ \begin{array}{l} post_f(\sigma^l,\sigma,\lambda(p^l),\lambda(p)) \;\wedge \\ P(\sigma^l,\lambda[p/p^l],head(\tau)) \;\wedge \\ tail(\tau) = (f, \sigma^l \oplus \lambda^l, \sigma \oplus \lambda) \end{array} \right\}} \qquad (VMCR)$$

The intuition behind this rule is as follows. If we know that in the state immediately prior to the invocation of a virtual method g an assertion P holds, and we also know that whenever P holds, the pre-condition for g also holds on the current object state and actual parameters of the call, then we can conclude that the post-condition of g will be satisfied when the method has completed. The post-condition of g is an assertion over the current states of the object (σ) and parameter p ($\lambda(p)$) as well as the initial states of the object (σ^l) and parameter p ($\lambda(p^l)$). We also need to assert P immediately after the completion of the method since any properties that it expresses about variables which are not affected by the call must still be true. Of course, P will not say anything about the current value of the object or p, but instead it will talk about σ^l and p^l (the notation $\lambda[x/y]$ refers to the mapping λ but with the value at y replacing the original value at x). Also, any properties expressed by P about τ before the call, can be asserted after the call only about $head(\tau)$ (which is all of the elements of τ except the last). Finally, we can conclude that the new last element of τ contains a call to g, the state immediately before the call, and the state immediately following the completion of the method.

Using the $(VCMR)$ a designer can analyze the code of a polymorphic method f and keep track of the value of the virtual method call trace at all times. Standard techniques for sequential composition of statements, while-loops etc, will then allow the designer to derive the appropriate post-condition of f.

3.2 Establishing a Derived Class Specification

Our next task is to introduce rules that will allow a derived class designer to establish an enriched specification for an inherited polymorphic method f, without reanalyzing the code of the method. Let us consider how such rules can be derived. Suppose we would like to verify a new post-condition for f in a derived class D of C. We will refer to this new post-condition as $post.s_f^D$. Note that we are considering how to verify the new specialization post-condition first since the functional post-condition can be obtained from it. Let us assume that the pre-condition of f holds over the initial state of the object upon which f is invoked as well as for the initial values of f's parameters (p_f). In other words assume that

$$pre_f(\#\sigma, \#\lambda(p_f)) \tag{1}$$

is true. What then can we say about the behavior of f in D? That of course depends on the specifications in D of each virtual method $g_1 \ldots g_n$ invoked by f. We have required that the pre-conditions of these methods should be the same as in C, and the post-conditions must be stronger. If these requirements are met, then the reasoning that C's designer performed to establish $post.s_f$ is still valid and we can assume this post condition holds over the base class portions of the initial and final states and τ:

$$post.s_f(\#\sigma, \#\lambda(p_f), \sigma, \lambda(p_f), \tau) \tag{2}$$

This is an important point because it means that we do not have to repeat the verification of the properties expressed in the above post-condition.

But the properties expressed by the post-condition will only talk about the parts of the state present in C. What about the parts of the state introduced in D? (We will refer to the elements of the state present in C as $C(\sigma)$ and those added in D as $D(\sigma)$.) First we know that the elements of $D(\sigma)$ will not be changed until the first time a virtual method is invoked. We also know that these elements will not be changed after the last time a virtual method is invoked. We can therefore conclude the following:

$$D(\#\sigma) = D(\tau.is[1]) \tag{3a}$$
$$D(\sigma) = D(\tau.fs[|\tau|]) \tag{3b}$$

Here the notation $\tau.is[i]$ refers to the initial state recorded in τ immediately before the call to the virtual method at the i^th position. Similarly $\tau.fs[i]$ refers to the final state after that same method returns. We will refer to the above assertions collectively as the *initial/final state* conditions. They will allow us to relate the states that are recorded in τ to the initial and final states that are constrained by the pre- and post-conditions of f. But what can we say about the derived class portions of the other states recorded in τ? We know that the derived class state cannot be changed between calls to virtual methods, and we know that the states immediately before and after a call to a virtual method will be related by the derived class post-condition for that particular virtual method. This gives us the following two conclusions:

$$\forall i, 0 < i < |\tau|, D(\tau.fs[i]) = D(\tau.is[i+1]) \tag{4a}$$
$$\begin{aligned}&\forall i, 0 < i \leq |\tau|, (\tau.m[i] = g) \Rightarrow \\ &\qquad (pre^D_g(\tau.is[i]) \Rightarrow post^D_g(\tau.is[i], \tau.fs[i]))\end{aligned} \tag{4b}$$

Here the notation $|\tau|$ represents the length of τ, and $\tau.m[i]$ refers to the name of the method at position i in τ. Note that we actually have a separate instance of the second conclusion for each virtual method of C, since each of these will have different pre- and post-conditions. We will refer to these two assertions collectively as the *trace consistency* conditions. They allow us to reason about the contents of each state which is recorded in τ and combined with $(3a)$ and $(3b)$ above, allow us to reason about the total effect f has on the derived class state.

Now we are ready to specify how a designer can verify a new specification for f in D. Essentially, if the new post condition follows from the conclusions we have enumerated, then the new post condition is valid. This is stated formally as our *derived class post-condition rule*:

$$\begin{aligned}&[(1) \wedge (2) \wedge (3a) \wedge (3b) \wedge (4a) \wedge (4b)] \Rightarrow \\ &\quad post.s^D_f(\#\sigma, \#\lambda(p_f), \sigma, \lambda(p_f), \tau)\end{aligned} \tag{DCPR}$$

A derived class designer must verify that the above implication holds in order to conclude that $post.s^D_f$ is a valid post-condition for f in the derived class.

4 Case Study

In this section we will apply our approach to a simple example. The base class for our example is Circle. It has member variables which specify the coordinates of the center of the circle and its radius. The class has two member functions, Double and BlowUp. Double is a virtual function; it receives no parameters, and is defined (in the base class) to double the radius of the circle. BlowUp is a polymorphic function that invokes Double several times to increase the radius of the circle appropriately. It receives an integer parameter that specifies how many times the radius of the circle should be doubled and calls Double the appropriate number of times. Since BlowUp is a polymorphic function, it will have a functional as well as a specialization specification. When specifying the latter, we of course will deal with its trace[6] which records all the calls it makes to Double.

In Figure 1 the specification of Double asserts that when this (virtual) method finishes execution radius will be twice what it was at the time of the call. Note that the post-condition does not explicitly say that Double will not modify the coordinates of the center of the circle. We could have added the appropriate clauses to the post-condition to assert this but have instead provided a *modifies* clause which provides this information. In other words, if there is a *modifies* clause, then only the values of member variables mentioned in this clause may be modified by this function; all the other variables (if any) of the class will be left unchanged by the function. But note that this clause does not prevent us from introducing new member variables in a derived class, and redefining Double to manipulate some of those. All it says is that the values of member variables

[6] If the class involved more than one polymorphic function, we will need to use separate trace variables for each polymorphic function.

```
class Circle {
    protected:
        Real centerx, centery, radius;

    public:
        ...
        void virtual Double();
        // pre_D ≡ true
        // modifies radius
        // post_D ≡ radius = 2 * #radius
        { radius *= 2; }

        void BlowUp( int i );
        // pre_B ≡ i ≥ 0
        // modifies radius
        // post.f_B ≡ radius = #radius * 2^i
        // post.s_B ≡ [(radius = #radius * 2^i) ∧ (|τ| = i) ∧ (τ.m[k] = Double)]
        { while (i > 0) { Double(); i– } }
}
```

Figure 1: Partial definition of class Circle

of the current class (and those inherited from the base class, if any, of the current class) not mentioned in the *modifies* clause will not be changed by this function.

The functional specification of BlowUp says that it multiplies the radius by the appropriate factor (2^i). Its specialization specification, $post.s_B$, says that it will call the Double method i times. Note that there is an implicit quantification over k that we have left out; in other words, this clause should be read to mean that it applies to all the elements of τ, i.e., that each element of τ corresponds to a call to Double.

Note also that this clause does not say what the state will be at the time of each of these calls, nor at the time of each return. This may seem like an oversight. If the value of the radius is allowed to be anything at these points during the execution of BlowUp, how can we know that the value of radius when this function finishes will be the value at the start times 2^i as both $post.f_B$ and $post.s_B$ claim? The answer is that when verifying the behavior of BlowUp at each call we will apply the $(VMCR)$ and make use of the specification of Double, which assures us that it will double the value of radius; this will allow us to establish that when BlowUp finishes, the value of the radius will indeed be what its specification claims. The program specifier is not, however, obliged to retain this information (i.e. that the value in radius following each call to Double will be twice what it was at the start of the call) in the specification about the trace. What if a derived class designer were to redefine Double so that it *triples* the value of radius since this would not contradict what $post.s_B$ of BlowUp says? He is not allowed to do that since such a redefinition would not satisfy the specification of Double. In other words, it is not the (specialization) specification of BlowUp that imposes restrictions on how we can redefine Double in the derived class. That is done by the specification of Double. Any redefinition that still satisfies the base class specification of Double is legitimate.

Given the specification of Double in Figure 1, this means that the derived class designer has no freedom at all in what his redefined Double may do to the member variables of the Circle class: it must double the radius and leave the coordinates of the center unchanged. What the derived class designer *can* do, and this is what we will do in the BorderClass below (see Figure 2), is to introduce new member variables in the derived class and have the redefined Double manipulate these as he chooses. And the specialization specification $post.s_B$ of BlowUp will let us establish the effect that a call to BlowUp will have on these new variables.[7]

[7] In Figure 2 we use the superscript d to distinguish the derived class pre and post-conditions from their base class counterparts in Figure 1.

```
class BorderCircle : public Circle {

    protected:

        Real thickness;

    public:

        void virtual Double();
        // pre^d_D ≡ true
        // modifies radius, thickness
        // post^d_D ≡ (radius = 2 * #radius) ∧ (thickness = 2 * #thickness)
        { self.Circle::Double(); thickness *= 2 }
}
```

Figure 2: Definition of class BorderCircle

A BorderCircle object is different from a Circle object in that it has a *border* whose thickness also doubles each time the radius is doubled; in other words, the larger we make BorderCircle, the fatter it gets. This is specified in $post^d_D$ which also asserts that the radius is doubled, as it should be, given the base class specification.

At this point it is reasonable to ask the question: what is the specification of BlowUp in the class BorderCircle? Since Double still satisfies its base class specification, and BlowUp was verified in the base class on the basis of that specification, we can conclude that the base class functional specification for BlowUp is still valid. But we would like to be able to say more than this, in particular, we would like to know what effect BlowUp has on the derived class member thickness. To answer this question we need to look at the specialization specification of BlowUp, specifically the post-condition $post.s_B$, which tells us that Double is called i times during the execution of BlowUp. Since thickness is not even defined in the base class, these invocations of Double are the only statements in BlowUp which can affect its value. From $post^d_D$ we know that Double will have the effect of doubling the thickness, and thus we can conclude that BlowUp will double the thickness i times. This means that we can give the following functional specification for BlowUp:

$$post.f^d_B \equiv (radius = \#radius * 2^i) \wedge (thickness = \#thickness * 2^i)$$

What about the specialization specification? $post.s_B$ tells us that the trace of virtual function calls made by BlowUp depends only on the parameter passed to the method, not on the behavior of Double. Therefore, whatever we know about the trace from the base class specification will continue to hold in the derived class. This gives us:

$$post.s^d_B \equiv (radius = \#radius * 2^i) \wedge (thickness = \#thickness * 2^i) \wedge$$
$$(|\tau| = i) \wedge (\tau.m[k] = Double)$$

This specification can be formally verified using the $(DCPR)$ we introduced in Section 3. The *traceconsistency* conditions will allow us to conclude that τ in the derived class takes the form of i calls to Double, and that at each call the initial and final states are related by $post^d_D$. From this we will see that in each successive element of τ the value of thickness is doubled and that this happens i times. The $(DCPR)$ is designed so that we can combine this new reasoning with the conclusions already verified for $post.s_B$, namely that the radius is doubled the appropriate number of times, to end up with the final form of $post.s^d_B$ above.

Alternatively, we could establish the richer functional specification of BlowUp by looking at the body of the method from the base class and seeing exactly when and how it invokes the Double method. But this is exactly the kind of reanalysis that we are trying to avoid. By providing information about τ in the specialization specification of BlowUp we allow the derived class designer to establish new properties of the method *without ever appealing to the code of the method in the base class.* While it might be argued that in this particular situation, reasoning with respect to the trace of virtual function calls is at least as complicated as looking at the (very simple) body of BlowUp, this will not be true in more complicated examples where the polymorphic function contains large sections of code devoted to manipulating various base class members. In these situations τ will distill the exact information critical to the derived class designer's reasoning and elide the rest. Moreover, if the base class is purchased from a software vendor, the source code of polymorphic functions such as BlowUp may not even be available, which would make reanalysis not just undesirable, but impossible.

Now let us return to a point we considered earlier. In the above example we did not need to provide any information about the states immediately before, and immediately after the calls to the virtual methods recorded in the trace. Under what conditions would this information be needed? In general, if the redefined virtual method is going to manipulate the new member variables in ways that depend on the values of the member variables of the base class, information must be provided about the state immediately before each virtual method is called. For instance, the designer of BorderCircle may decide that if the radius of the circle is less than one, then Double should not double the thickness after all (since the border might become so thick that the circle appears as a dot!) In this case the final effect of BlowUp depends on the value of the radius immediately before each call to Double. Clearly, based on the *code* of BlowUp, if the initial radius is greater than one, then BlowUp will double the thickness i times. But $post.s_B$, as it is written, does *not* allow us to conclude this. Suppose the code of BlowUp first saved the value of the radius in a temporary variable, then set the value of the radius to zero, called Double i times, restored the value of radius, and finally multiplied it by two, i times. Such a body would satisfy $post.s_B$, but its effect in the derived class we are considering would be to always leave the thickness unchanged. $post.s_B$ must provide more information to allow a derived class designer to determine the effect of BlowUp on the thickness properly in this case. In particular, $post.s_B$ must state that for every j less than or equal to i, in the state recorded immediately before the jth call to Double in τ, the value of the radius must be equal to the initial value times 2^j.

On the other hand, there are also situations where the post-condition of a polymorphic method will need to talk about the state immediately *after* a virtual method call returns. Suppose we have a class Figure which contains only an anchor point and a flag indicating whether or not the figure is displayed on the screen. A possible virtual method might be Intersect(Line l, Boolean in). This method is designed to test whether the given line segment intersects the figure and to set in to true if so (and false otherwise). Of course, since Figure itself is a rather unconstrained base class (after all, it only contains an anchor point), it is expected that the derived classes of Figure (Square, Triangle, etc..) will provide meaningful implementations of Intersect. In the base class the post-condition for Intersect will simply say that the value of in may be set to true or to false. The base class *can* provide a useful method which uses Intersect however. Clip(Line l) is a method which tests whether a figure intersects with a given line and if so, sets the displayed flag to false. Clip of course is implemented using a call to the virtual method Intersect. The functional post-condition for Clip will simply say that the displayed flag of the figure may be set to false if it is not already. Obviously in the base class no indication of the circumstances which would cause the displayed flag to be turned off can be established, because of the vague specification of Intersect. In a derived class such as Circle, however, when presumably the specification of Intersect has been sufficiently tightened, we would like the specification of Clip to state under what circumstances the displayed flag is turned off. The functional specification of Clip will be little help to us here. Instead we need the specialization specification of Clip to state that if the value returned from the virtual call of Intersect is true, then the flag is turned off, and it is left alone otherwise. The only way for the specialization specification to express such a condition is if the state immediately following the call to Intersect is recorded in τ.

5 Conclusions

In the hands of a talented class designer, OO polymorphism is an extremely powerful tool. It allows him to design a base class B that provides a host of polymorphic functions and a carefully selected set of virtual functions, so that a derived class designer can build an entire new class D precisely tailored to the needs of the particular kinds of objects that his application calls for, by just providing appropriate definitions of the virtual functions of B. All the polymorphic functions that D inherits from the base class will, without any apparent effort on his part, exhibit richer behavior appropriate to the needs of (instances of) D. It is this remarkable ability to not just reuse the base class functions but enrich their behavior that has been instrumental in OO's wide spread acceptance.

But any powerful tool can lead to problems if it not used properly. The derived class designer needs to be able to not just construct his class, but convincingly show that the inherited polymorphic functions will exhibit the enriched behavior that he claims for them. One approach would simply be to reanalyze the code of each polymorphic function, given the behaviors of the redefined virtual functions. But this would almost entirely nullify the advantage of OO polymorphism. What use code reuse without reasoning reuse? Rather we need an approach that will avoid the need for reverifying the (base class) code of the polymorphic functions. That has been the fundamental motivation behind our approach. And we do not just want the derived class designer to be able to check that the inherited functions will continue to exhibit their original behavior. If that is all we cared about, why bother to define D in the first place? The point of OO polymorphism is that it allows the derived class designer to build an enriched class. The reasoning formalism needs to be able to establish the richer behavior (of the inherited functions).

To enable this, we introduced the idea of *specialization specification*; thus the specification of a polymorphic function p in the base class not only specified p's functional behavior, i.e., its effects on the various member variables of B[8], but also, in the specialization specification, additional information that the derived class designer will need to establish the richer behavior of p in his class. In simple examples of the kind that we were necessarily limited to in this paper, the specialization specification for p looks as complex as the code for p. That is because the polymorphic function in these examples did little more than call the virtual functions. But in more realistic examples, the polymorphic functions will execute large segments of code, with a few calls to key virtual functions inserted at key points. For such functions, the specialization specification will only be slightly more complex than the functional specification. Given the specialization specification of p from the base class, the derived class designer can indeed establish, using the formalism of section 3, that the inherited p will exhibit the richer behavior that he claims for it. One question that a base class designer writing down a specialization specification for a polymorphic function has to answer, as we saw in sections 2 and 4, is how much information to include in it. Making it complete, in other words specifying as completely as possible the states that are recorded in the virtual method trace, as well as how the final values of the function depend on these states, will lead to rather complex specifications. But this complexity will enable the derived class designer to pin down the behavior that the polymorphic function will acquire in the derived class more precisely. Typically, the base class designer probably has a good idea of which data members in the base class are likely to be used by virtual methods when they are redefined in the derived class, and will include complete information about only such members.

Some other authors who have considered the problem of polymorphic functions are Stata and Guttag [9], Kiczales and Lamping [2], Lamping [3], and Lamping and Abadi [4]. [4] focuses on typing issues, whereas our interest is in behavioral issues. [3] proposes specifying for each polymorphic function the set of virtual functions that it invokes. This will allow a derived class designer to know which functions might potentially be affected –enriched in our terminology– by redefinitions of specific virtual functions. The idea seems to be that the designer can then go back and study the code of the function in the base class to see how it will be affected. [9] extends this idea to cover behavioral issues; i.e., they propose identifying for each polymorphic function the behavior that a given virtual function must exhibit in order for the polymorphic function to continue exhibiting its original behavior. So long as the derived class designer's redefinition of the virtual function exhibits this required behavior, the polymorphic function will continue to meet its original specification. But these approaches do not seem quite in tune with the intent behind OO polymorphism. It is supposed to *enrich* the base class behavior, not just preserve it. That is why we have designed our formalism to arrive at this richer behavior, not just check that the base class behavior continues to hold. Not surprisingly, our formalism is more complex than those of [9, 3], but we believe this added complexity is a price well worth paying. Kiczales and Lamping [2] perhaps comes closest in spirit to our approach. They propose providing information about which virtual functions the polymorphic function will call and the order in which it will call them. But they don't talk about establishing behavioral specifications of the

[8] In a complete formalism we would also have abstract versions of this specification for use by clients of the class.

kind we are interested in.

One of the most interesting applications of polymorphism is in the building of OO frameworks. An OO framework consists of a collection of classes each of which contains several polymorphic as well as virtual functions. The entire collection is intended to be used as a unit. An application builder would define derived classes of each of the classes in the framework and provide definitions for all the virtual functions. The approach seems to make it possible to develop entire applications with modest effort. Is our approach applicable to understanding the behavior of frameworks and applications built on top of them? We believe it is, and a reasoning technique such as ours that allows us to avoid re-verification of polymorphic functions and enables us to arrive at their richer behavior, is even more critical for dealing with frameworks since their sheer size argues against reanalyzing them each time we build a new application using them.

References

[1] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[2] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. In *OOPSLA '92*, pages 435–451, 1992.

[3] J. Lamping. Typing the specialization interface. In *OOPSLA*, pages 201–214, 1993.

[4] J. Lamping and M. Abadi. Methods as assertions. In *ECOOP*, pages 60–80, 1994.

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

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

[7] N. Soundarajan and S. Fridella. Inheriting and modifying behavior. In Meyer Ege, Singh, editor, *Proceedings of TOOLS '97*, pages 148–163. IEEE Computer Society Press, 1997.

[8] N. Soundarajan and S. Fridella. Reasoning about polymorphic behavior. In *TOOLS 98 (submitted)*, 1998.

[9] R. Stata and J.V. Guttag. Modular reasoning in the presence of subclassing. In *OOPSLA*, pages 200–214. ACM Press, 1995.