# Enriching Behavioral Subtyping

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

December 1, 1997

## Abstract

The standard approach to dealing with OO polymorphism is to require subclasses to be behavioral subtypes of the base class. This ensures that reasoning that has been done about any client code that operates on base class objects will continue to be valid if instances of the subclasses are used in place of the base class objects. But often we are interested in stronger properties of the client code, in particular that its behavior will be appropriate to the *specific* subclass objects that are used, rather than just generic behavior that ignores the differences between the different subclasses. We present some examples to illustrate the problem, and propose a formal system that allows us to establish stronger properties of the client code on the basis of the richer behavior provided by the appropriate derived classes.

## 1 Introduction and Motivation

One of the most important ideas underlying the Object Oriented approach is *polymorphism*.[1] Polymorphism allows us to use an object that is an instance of a *derived class* in place of an object of the *base class*, with the run-time dispatch system ensuring that the methods applied to this object are the ones defined in the derived class. This enables us to write client code that can deal with objects that may be instances of any of the derived classes of a given base class, but treat them uniformly as if they were all instances of the base class. Indeed, it is even possible to define *new* derived classes of the same base class, and invoke the original client code on instances of this new class since the run-time system will ensure that the functions that are actually applied to these objects are the ones defined in the new derived class. Much has been written about the power of OO polymorphism; see Meyer [12], for instance.

The goal of this paper is to address some questions in the specification and verification of systems that use OO polymorphism. The standard approach to the problem is in terms of *behavioral subtyping* (see, for instance, [1, 10]). In effect, given two classes $C$ and $D$, and given a specification of $C$, the class $D$ is considered a behavioral subtype of $C$ if the behaviors exhibited by the methods of $D$ are compatible with the specification of $C$; in other words, the methods of $D$ should not exhibit any behaviors that are not allowed by the specification of $C$. The importance of behavioral subtyping in the context of OO polymorphism arises because of the following consideration: Suppose we have a class $C$ and some client code $CC$ that operates on objects of type $C$ and that we have verified, using the given specification of $C$, that this code exhibits a certain behavior $\mathcal{B}$. Suppose next that we have a number of derived classes $D_1, \ldots, D_n$ of $C$. Then we

[1] Throughout this paper by 'polymorphism', or occasionally 'OO polymorphism', we mean the *inclusion polymorphism* of Cardelli and Wegner [2].

can be sure that $CC$ will continue to exhibit the behavior $\mathcal{B}$ even if objects of type $D_i, i = 1, \ldots, n$ are used in place of the objects of type $C$ that $CC$ expects, *provided* each $D_i$ is a behavioral subtype of $C$. While this is certainly correct, it overlooks an important point. The derived classes typically have a richer conceptual model than does the base class, the methods defined in these derived classes being tailored to provide behavior appropriate for these richer models; and often the main reason for using OO polymorphism is to exploit this richer behavior. But the behavioral subtyping-based approach focuses on just ensuring that the behavior $\mathcal{B}$ already established for $CC$ using the base class specifications for the operations in question will continue to be valid even though the operations actually invoked will be the ones defined in the derived classes, rather than on establishing the richer behavior that will result from the use of these derived class operations. Our goal in this paper is to set up a formalism that can be used to establish richer properties of the client code by appealing to the behaviors provided by the relevant derived classes, instead of just basing our reasoning about the client code on the base class behavior.

The problem has been noticed by other authors see, for instance, Dhara and Leavens [4]. They consider an example consisting of a base class BankAccount and a derived class PlusAccount. Conceptually, the key difference between these two classes is that BankAccount objects have a single balance called credit, while PlusAccount objects have two, one corresponding to the balance in the checking portion of the account, the other corresponding to the savings portion. The problem they note is that when reasoning about client code that treats bank accounts polymorphically (as BankAccount objects), the behavioral subtyping approaches do not allow us to say anything about what happens individually to the savings and checking balances. Dhara and Leavens do not propose a solution to this problem and indeed they do not claim to, their interest being in defining a somewhat weaker notion of behavioral subtyping rather than solving this problem. In section 3 we will briefly show how our approach can deal with this example.

The main advantage of the behavioral subtyping approach is that client code that has been verified to be correct once does not have to be reverified when it is invoked on objects that are instances of the derived classes rather than of the base class. But this is reasonable only if the behavior that we want to verify remains the same. As we have argued, the reason for using OO polymorphism is often to exploit the richer behavior provided by the derived class(es). Hence it is likely that the behavior we want to verify for the client code will also be similarly richer. Correspondingly, in the system proposed in this paper, when we invoke some client code on newly defined derived class objects, we will have to do additional work in verifying this richer behavior.

There are two important ideas in our system that allow us to reason about this richer behavior. First, the conceptual model of the derived class will be an extension of the model of the base class; and the specifications of the various methods of the derived class specify the effects of the methods on this extended model. Second, we propose a set of axioms that allow us, when reasoning about client code, to arrive at results that depend upon the effect that the methods invoked have upon the *extended model*, rather than just upon the model of the base class. The combination of these two features enables us, as we demonstrate with an example in section 3, to establish interesting results about client code that cannot be established using the other approaches.

The rest of the paper is organized as follows: In the next section we describe our formalism. The first part of the section explains how base classes and derived classes are specified, the second part of the section explains how these specifications can be used to reason about client code. The third section applies our approach to a fairly typical example of OO polymorphism consisting of a base class Figure that corresponds to generic graphical objects that can be displayed on a screen, derived classes that correspond to specific kinds of figures like circles, and a simple piece of polymorphic client code that uses these classes. We also briefly discuss how our approach can be used for the example from [4]. In section 4 we summarize the motivation behind our approach, and reiterate the underlying ideas. We also talk about possible extensions to the system proposed in this paper.

## 2 Specializing Behavior

In section 2.1 we introduce our notation for specifying the behavior of base and derived classes. Our notation is designed to make it easy to specify the richer behavior provided by the derived classes. Section 2.2 considers how to use such specifications in establishing the behavior of client code, in particular the part that depends upon polymorphically exploiting the richer model and behavior provided by the derived classes

Before presenting our formalism we should note that throughout much of this paper we assume that OO polymorphism is implemented using the inheritance mechanism of languages such as *Simula, Eiffel, C++*[2]. It is also possible, in these languages, to use inheritance to define new classes reusing part of the work that has gone into defining existing classes, with the derived class not being conceptually closely related to the base class. Such derived classes of course are not meant to be used polymorphically and will not obey the restrictions imposed in this paper. We will not consider such use of inheritance in this paper; in [14] we show how the designer of such derived classes can, in validating his[3] class, make use of much of the reasoning that the base class designer went through in validating the behavior of the base class. Finally, OO polymorphism does not require, for its implementation, a *C++*-like inheritance mechanism; for instance, *Java*'s interface inheritance allows us to achieve OO polymorphism although the base class internals are not inherited into the derived class. Our approach can also be used for dealing with such implementations, although we will not address this directly in this paper.

### 2.1 Specifying base and derived classes

Suppose $B$ is a base class and $D$ is a derived class of $B$.[4] Let us first consider how to specify $B$. As usual, each class will have a *concrete* specification and an *abstract* specification. The concrete specification of $B$ is needed not only when establishing the correctness of $B$'s implementation, but also, as we discuss in some detail in [14], when establishing the correctness of $D$'s implementation. Since our focus in this paper is on reasoning about the behavior of client code rather than on establishing that classes meet their specifications, we will not worry about the concrete specifications of the classes, but instead concentrate on their abstract specifications.

The abstract specification of $B$ will consist of three components: a conceptual model of the class; an invariant over this model; and the specification of each (public) method[5] of the class in the form of a pre- and post-condition over the conceptual model. We will use $B.\mathcal{M}$ to denote the conceptual model of $B$, $B.Inv$ to denote its invariant, and $B.pre.f$, $B.post.f$ for the pre- and post-conditions of the method $f$. The pre-condition of $f$ will be an assertion on the conceptual states of the object and the parameters of the method that must be satisfied when $f$ is invoked. The post-condition is an assertion on the states of the object and the method parameters when $f$ finishes execution and the corresponding states immediately before $f$ starts execution. Having a post-condition that refers to both the initial and final states when $f$ starts and finishes makes it easier to specify these methods. The pre- and post-conditions are written in terms of *self*, the (conceptual) object on which the given function is being applied.

Next consider the abstract specification of the derived class $D$. This will be similar to that of $B$ but we will impose certain conditions on it. First we will require the conceptual model $D.\mathcal{M}$ of $D$ to be an *extension* of $B.\mathcal{M}$; more precisely, $D.\mathcal{M}$ is a cartesian product of $B.\mathcal{M}$ and $ext_B^D.\mathcal{M}$, the latter being the (conceptual model corresponding to the) extension provided by the derived class. Thus an element of $D.\mathcal{M}$ will be of the form $\langle b, d\rangle$ where $b$ is an element of $B.\mathcal{M}$, and $d$ is the extension. The invariant $D.Inv$ for $D$ will, in general, refer to both components of $D.\mathcal{M}$. We will require this invariant to be stronger than $B.Inv$.

[2]Indeed we often use *C++*-like terminology, although the approach is applicable any language in this class.

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

[4]It would be more appropriate to say '$B$ is not a derived class' rather than to say '$B$ is a base class', because by looking at the definition of $B$, at least in standard languages like *C++*, we can only tell whether or not $B$ inherits from other classes, not whether $B$ itself will serve as a base class. It is also, of course, possible that a derived class serves as a base class for other classes; we will briefly consider this question later in the paper.

[5]Again since our interest in this paper is on establishing client code behavior, we will ignore protected and private methods since they are only of concern to the base and derived class implementors.

The relation between the specifications in $D$ and $B$ of the individual methods is more complex. If we want the client code to be able to exploit the richer behavior provided by $D$, then clearly the specifications of the methods of $D$ must provide information about how these methods use the extra component $ext_B^D.\mathcal{M}$. But at the same time, in order to ensure that behavior of client code that has been established on the basis of the specification of the base class $B$ is not violated when instances of $D$ are used, we must be sure that these methods also satisfy whatever the specification of $B$ requires of them. Thus we will associate *two* specifications with each method of $D$. The first specification will be identical to the corresponding specification in the base class and can therefore be implicit, instead of being explicitly specified. Or rather, the derived class designer will have to, as part of the task of validating his class, check that the methods, as implemented in $D$, satisfy the corresponding specification in $B$.

The second specification of $D$'s method will provide information about how the method manipulates the additional component of the conceptual model of $D$. It is this second specification that the client programmer will have to rely upon in order to understand the richer behavior provided by $D$. A natural question would be whether the two specifications can be combined into one; in general the answer is no since preserving the behavior of client code that was established on the basis of the base class specification will require the pre-conditions of the methods not to refer to new component of the conceptual model, whereas properly specifying the effect of the function on the extended model will require us to impose conditions on the extension as well. We will discuss this in more detail in section 4.

All of this applies only if the method in question is also in the base class. If the method is an entirely new one that the derived class designer introduced, we will only have a single specification for it, one that describes its complete effect on both components of the model of $D$. All methods including such new methods must, of course, preserve the invariant of the class.

How do we verify that a class, base or derived, meets its (abstract) specification? Consider the base class. First we need to introduce a *concrete* specification for the class. This specification will describe the effects of the various methods in terms of how they affect the values of the various member variables of the class rather than in terms of the conceptual model. We also need an abstraction function that maps the concrete state of the class onto the conceptual state, and we will have to show that under this mapping, the concrete specification in some precisely defined sense implies the abstract specification. All of this is fairly standard, see for instance the text by Jones [6].

The derived class will similarly have a concrete specification. As discussed in [14], part of this specification can be inherited from the concrete specification of the base class. Since our interest in this paper is in verifying client code, we will omit these details, as well as the details of verifying that the client code does meet its abstract specification, referring the interested reader to [14]. One important difference with the system in [14] is that in the current situation, we have to deal with two abstract specifications for the derived class. This will, in general, require us to come up with two concrete specifications for the class the difference between the two being that one will describe the effects of the various methods on all the member variables of the class including those inherited from the base class, while the other will only describe the effects on the variables inherited from the base class.[6] While this means a bit more work for the derived class designer, conceptually nothing new is required for this purpose.

## 2.2 Behavior of client code

Recall that our main goal is to be able, in reasoning about client code, to bring out the richer behavior provided by the derived classes. Information about this richer behavior is, as we saw in 2.1, provided by the (abstract) specification of the derived classes but we need some additional formal machinery to be able to use this information. Specifically, when considering a call such as x.f() where x is a variable declared to be of type $B$, we want to be able to use the specification of the method f in the derived class $D$ if we know that

[6]This is not strictly correct; the abstraction function in the derived class will in general be different from that in the base class, and this function may be such that it uses some of the new member variables of the class to map to the base class portion of the conceptual model. In that case this second concrete specification will have to include information about these member variables.

the *object* that x refers to[7] is, in fact, of type $D$.

For any variable x, let dtype.x refer to the declared type of x, and otype.x refer to the type of the object that x currently refers to. dtype.x will be fixed when x is declared and does not change thereafter. otype.x can potentially change whenever x is made to refer to a new object, since this new object may be of a different type than the one x previously referred to. There are two possible ways in which x may be made to refer to a new object, First, we may use the constructor function of a class $D$ to create a new object; in *C++* this would be written as,

x := new D(...);

where the arguments expected by the constructor function are indicated by the '...'. Second, if we have another variable y (whose declared type need not be the same as that of x) which refers to an object of type $D$, then we can assign y to x with the result that x will refer to this object following the assignment:

x := y;

In both of these cases, otype.x will become $D$, since x now refers to an object of type $D$.

There is one more point to consider, the *particular* object that x refers to at any given point. The reason this is important is that the effect of a method call like x.f(...) is to modify the state of the object that x currently refers to. In other words, x continues to refer to the same object as before but the state of that object is in general different from what it was before the method call. We will use obj.x to refer to the object that x refers to at any given time; clearly obj.x will in general change only following the two types of assignments to x considered in the last paragraph.[8] For simplicity, we will assume that all objects are numbered 1, 2, .... Thus if at any given time x refers to the object numbered 25, the value of obj.x will be 25. Further, we will assume that ObIds is the set of numbers of all objects currently in existence. Finally, we will use ObSts to denote the states of all the objects currently in existence; ObSts is essentially a mapping from the numbers of the objects to their current (conceptual) state.

Let us now consider the assertions that we may use when reasoning about client code. As before, let x be a variable declared to be of type $B$. Our assertions will be written in terms of dtype.x, otype.x, and obj.x, as well as ObIds and ObSts. The effect of a declaration will be to assign a value to dtype.x; no value is assigned to otype.x or obj.x since x does not currently refer to any object.[9] This may be formalized with the following axiom:

$$\{ \text{ true } \} \text{ B x; } \{ \text{ dtype.x} = \text{B } \} \tag{1}$$

This says that following the declaration of x, its dtype has the appropriate value.

Next consider creating a new object of type $D$ (a derived class of $B$ as usual) and assigning it to x. Let us assume, for simplicity, that there is only one constructor function in $D$ and that the specification of $D$ contains the following, $p$ being an assertion involving the argument y of the constructor function, cons.D denoting the constructor function, and the post-condition $q$ being an assertion involving *self* and y:

$\{ p \}$ cons.D(y) $\{ q \}$

The effects of using this constructor function to create an object of type $D$ and having x refer to it are the following: the otype.x will become $D$; a new object, with its own identity $i$ (an integer not used to identify any other object) is created, and the identity of this object is recorded by adding $i$ to ObIds; the state of this object, as specified by the value of ObSts[$i$] as well as the states of any objects passed as parameters, will satisfy the assertion $q$, the post-condition of the constructor function; all the other objects will have the same state as before. We can capture all of this with the following axiom (# denotes the original value of the entity in question before the statement began execution; since our post-conditions are assertions over the initial and final states, # will appear frequently):

---

[7]In *C++*, we would be required to declare x to be of type *pointer* to $B$, but this is a language detail that we will ignore.

[8]Since in this paper we will not address problems of aliasing –two or more variables referring to the same object– we could have avoided introducing obj.x and instead treated objects as values, with a new value being generated following a call like x.f(...); but including it in the formalism will, we hope, allow us to deal in the future with aliasing; we will return to this briefly in the final section.

[9]Alternately, we could introduce a distinguished '*unassigned*' value and assign this to otype.x and obj.x.

$$\{\ \mathsf{dtype.x}\ \preceq\ \mathsf{D} \wedge p^{\mathsf{y}}_{\mathsf{ObSts[obj.z]}}\ \}$$
$$\mathsf{x} := \mathsf{newD(z)}$$
$$\{\ \mathsf{dtype.x} = \#\mathsf{dtype.x} \wedge \mathsf{otype.x} = \mathsf{D}$$
$$\wedge\ \mathsf{ObIds} = \#\mathsf{ObIds} \cup \mathsf{newobid}(\#\mathsf{ObIds}) \wedge \mathsf{obj.x} = \mathsf{newobid}(\#\mathsf{ObIds})$$
$$\wedge\ q^{self,\mathsf{y}}_{\mathsf{ObSts[newobid(\#ObIds)],ObSts[obj.z]}}$$
$$\wedge\ \forall k \in \mathsf{ObIds}.[k \neq \mathsf{newobid}(\#\mathsf{ObIds}) \Rightarrow \mathsf{ObSts[k]} = \#\mathsf{ObSts[k]}]\ \} \tag{2}$$

The first clause in the pre-condition uses the relation '$\preceq$' to check that the declared type of x is a base class of D; this condition is not really required since it is the job of the compiler to check this. The second clause in the pre-condition checks that the pre-condition of the constructor function being invoked is satisfied (with the formal parameter y being replaced by the actual object to which the argument z refers). In turn, the last but one clause in the post-condition asserts that the state of the newly created object satisfies the post-condition of the constructor function (with the appropriate substitutions). We have assumed that there is a function newobid that, given the current set of object identifiers, gives us the 'next available' object identity. Thus the newly created object has the identity newobid(#ObIds). The last clause in the post-condition above says that the states of all the other objects are unchanged.

The effect of an assignment statement is similar but somewhat simpler since no new object is being created:

$$\{\ \mathsf{dtype.x} \preceq \mathsf{dtype.v}\ \}$$
$$\mathsf{x} := \mathsf{v}$$
$$\{\ \mathsf{dtype.x} = \#\mathsf{dtype.x} \wedge \mathsf{otype.x} = \mathsf{otype.v} \wedge \mathsf{obj.x} = \mathsf{obj.v}\ \} \tag{3}$$

We should note here that we have not explicitly stated that nothing other than x, for instance the value of ObIds or ObSts, changes; that is implicit and we will feel free in the discussion of the examples in the next sections to include those clauses.

Finally, and most important, let us consider the effect of a call to a method of a class. There are two possible situations. The first, and more general one, is when we only know about the declared type $B$ of a variable; in this case, we can only make use of the specification of the class $B$. This means the result we can establish will only talk about a portion of the (conceptual) state of the object that the variable refers to, the portion that corresponds to the class $B$; if, in fact, the object is an instance of a derived class $D$ of $B$, then we will not be able to say anything about the part of the state that corresponds to the derived class. The second case is when we also know the type of the object $D$ that the variable currently refers to, in which case we can use the specification of the class $D$. Since $D$ must be a derived class of $B$, the relation between the specifications of $B$ and $D$ guarantee that the result we can establish in the latter case is stronger, which is of course the whole point of the formalism. We have two axioms, one corresponding to each of these two cases:

$$\{\ \mathsf{dtype.x} = B \wedge B.pre.\mathsf{f}^{\mathsf{y},self}_{\mathsf{ObSts[obj.z],ObSts[obj.x]}(B)}\ \}$$
$$\mathsf{x.f(z)}$$
$$\{\ \mathsf{dtype.x} = B \wedge \mathsf{obj.x} = \#\mathsf{obj.x} \wedge B.post.\mathsf{f}^{\mathsf{y},\#\mathsf{y},self,\#self}_{\mathsf{ObSts[obj.z]},\#\mathsf{ObSts[obj.z]},\mathsf{ObSts[obj.x]}(B),\#\mathsf{ObSts[obj.x]}(B)}\ \} \tag{4}$$

The second clause in the post-condition says that x continues to refer to the same object that it previously did. The final clause says that final state of that object satisfies the post-condition of the function f as given by the specification of the class $B$. The notation ObSts[obj.x]$(B)$ refers to the $B$ component of the state of this object; since the object may be of type $D$ (a derived class of $B$), it may have additional components; the pre-condition of f in the specification of $B$ will not, naturally, impose any conditions on these additional components, so we only need to refer to the $B$ component of this state. And in turn, in the post-condition we can provide no guarantees about such additional components of the state of this object since the post-condition of f in the class $B$ will say nothing about them.

This means that if at the time of the call to f we do not know the actual type of the object that x refers to, we can only arrive at conclusions based on the *declared type* of x which is precisely what we would expect. On the other hand, if we do know the type of the object that x refers to to be $D$, the following axiom allows us to make assertions about the additional components of the state:

$$
\begin{array}{l}
\{\ \mathsf{dtype.x} = B \wedge \mathsf{otype.x} = D \wedge D.pre.\mathsf{f}^{\mathsf{y},self}_{\mathsf{ObSts[obj.z]},\mathsf{ObSts[obj.x]}}\ \} \\
\mathsf{x.f(z)} \\
\{\ \mathsf{dtype.x} = B \wedge \mathsf{otype.x} = D \wedge \mathsf{obj.x} = \#\mathsf{obj.x} \wedge D.post.\mathsf{f}^{\mathsf{y},\#\mathsf{y},self,\#self}_{\mathsf{ObSts[obj.z]},\#\mathsf{ObSts[obj.z]},\mathsf{ObSts[obj.x]},\#\mathsf{ObSts[obj.x]}}\ \}
\end{array}
\tag{5}
$$

We conclude this section with two points. The first is a fairly simple one. As we noted in connection with axiom (3), but this really applies to all our axioms, the post-conditions in the axioms have left out information about objects that are not affected by the statement under consideration. We could have written stronger axioms to include this but did not do so mainly to ensure that the important effects of the statements are easily seen. In the examples in the next section, we will assume that unaffected objects are unchanged.

The second point is more substantial. What about the case when we know that obj.x is either $D$ or $D'$ (both of which are derived classes of $B$)? In this situation, we would have consider the two cases separately, using (5) as necessary, and then combine the results using a standard disjunction axiom. In other words, if we want to consider the behavior of polymorphic code for two possible types of objects that x might refer to, and the behavior we are interested in depends upon the behavior provided by these two types and not by the base class, then we have to analyze the code separately for each of the two types. Of course, if all we care about is the behavior that depends on whatever the base class provides us, we can use (4) and that would of course be independent of the type of object x refers to.

# 3 Case Study: Drawing, Erasing, and Moving Figures

In this section we will use our approach to reason about client code which manipulates different types of graphical objects in a polymorphic manner using a hierarchy of figure classes. At the end of the section we will also briefly consider the bank account problem from [4].

Consider an application which allows users to define, draw, and manipulate many different types of graphical objects on the screen. These objects may include not only basic shapes such as circles, squares, and rectangles, but also more complicated screen entities such as splines, character strings etc. The challenge is that the program must manage many different types of objects, and do so in a uniform manner. After all, all of the objects will need to be displayed, edited, erased, and moved around the screen. Furthermore, it is unlikely that the program designers will be able to anticipate in advance all of the possible figures that the users of the application will eventually want to be able to create. Thus it is crucial that the code for the application be easily extensible. OO polymorphism is ideally suited for such an application.

The polymorphic approach for this application would be to first create an abstract (or deferred, we use both terms) base class, let us call it Figure, containing the interface, state, and behavior that is common to all graphical objects. Thus every Figure object must have an anchor point, methods for getting and moving that point, a color, and methods for drawing the object on the screen and erasing it from the screen. But while we can state in the Figure class that we have a method for drawing the figure, we cannot actually implement such a method in this class since the method for drawing a figure object will, of course, depend upon the type of the figure being drawn. In other words, Draw (as well as Erase) will be *pure virtual* functions in the Figure class. We can then proceed to implement a variety of derived classes such as Circle, Rectangle, etc. corresponding to different kinds of figures. Each of these classes would have to provide definitions appropriate to those particular classes for Draw and Erase. The application code can declare references to instances of the Figure class and manipulate them without regard to which derived classes of Figure are actually present in the program. Further new classes can be added to the Figure hierarchy and instances of these classes can used as they are needed without recompiling the existing application code. Our interest is of course in the question, how do we specify and verify the behavior of such application code?

Since our focus in this paper is on dealing with client code, rather than in showing that implementations of classes meet their specifications, we will only give the specifications of the classes; the interested reader may refer to [5] for one possible implementation of the various classes we consider. Next we will specify some client code that uses these classes polymorphically and use the approach of section 2 to reason about the behavior of this code; along the way we will contrast the application of our method with that of other approaches.

*class Figure*

*is modeled by : (anchor: point, color : integer)*

*post.Figure(Point a, int c)* $\equiv$ *self* $=$ (*a, c*)

*pre.MoveAnchor(int dx, dy)* $\equiv$ *(self.anchor.x + dx* $\geq$ *0)* $\wedge$ *(self.anchor.y + dy* $\geq$ *0)*
*post.MoveAnchor(int dx, dy)* $\equiv$ *( self* $=$ ( (*#self.anchor.x + dx , #self.anchor.y + dy*) ,*#self.color*) *)*

*post.GetAnchor()* $\equiv$ *(returns self.anchor)* $\wedge$ *(self* $=$ *#self)*

*post.Draw(Screen s)* $\equiv$ *(self* $=$ *#self)* $\wedge$ *(s.sizey* $=$ *#s.sizey)* $\wedge$ *(s.sizex* $=$ *#s.sizex)* $\wedge$ *(s.bcolor* $=$ *#s.bcolor)*
(*$\forall x \forall y$*( $(0 \leq x \leq$ *s.sizex*) $\wedge$ $(0 \leq y \leq$ *s.sizey*) ) $\Rightarrow$
( *s.pixels[x][y]* $\neq$ *#s.pixels[x][y]* $\Rightarrow$ *s.pixels[x][y]* $=$*self.color*) )

*post.Erase(Screen s)* $\equiv$ *(self* $=$ *#self)* $\wedge$ *(s.sizey* $=$ *#s.sizey)* $\wedge$ *(s.sizex* $=$ *#s.sizex)* $\wedge$ *(s.bcolor* $=$ *#s.bcolor)*
(*$\forall x \forall y$*( $(0 \leq x \leq$ *s.sizex*) $\wedge$ $(0 \leq y \leq$ *s.sizey*) ) $\Rightarrow$
( *s.pixels[x][y]* $\neq$ *#s.pixels[x][y]* $\Rightarrow$ *s.pixels[x][y]* $=$*s.bcolor*) )

Figure 1: Specification of base class Figure

Our first task is to give a specification of the Figure class. This appears in Figure 1.[10] An interesting feature of this specification is what it does *not* say. Note the post-conditions of the Draw and Erase methods. Presumably, the main purpose of these two methods is to cause a change to the state of the Screen object which is passed to them as a parameter. However, the post-condition of Draw says only that if a pixel on the screen is changed then it will be changed to the color of the figure which is being drawn. It does not say *which* pixels on the screen will be changed by the draw method. Similarly the post-condition of Erase ensures that pixels are only erased (set to the background color of the screen) and not drawn, but it does not mention *which* pixels are erased. This is because in the Figure class we don't really know how to draw or erase the figure; that is why we decided to make these functions pure virtual in this class. In other words, we do not know which pixels will be drawn (or erased); that is for the derived classes to worry about. What the specification of Draw in the Figure class does express is the base class designers *general* expectations of the Draw function (to be) defined in *all* derived classes, i.e., that Draw will not affect the anchor point, and that any pixels that are drawn on the screen have the color appropriate to the figure being drawn. The post-condition of Erase is motivated by similar considerations.

Our next task is to describe and specify some of the derived classes of Figure. Let us consider two natural classes of this kind, Circle and Rectangle. Both of these classes enrich the conceptual state of the base class: instances of Circle contain a *radius* along with the anchor point; instances of Rectangle contain another point in addition to the anchor point representing the (coordinate of the) rectangle's opposite corner from the anchor. Each class also, of course, enriches the behavior of the base class by providing specific implementations for the Draw and Erase functions. Again we will omit the implementations, and provide only the specifications of the classes. The specification of Circle appears in Figure 2. The formal specification for Rectangle would be structurally similar and we will leave it to the interested reader.

Recall that each derived class will have two specifications, one which is identical to the specification of

[10] Here and in the following specifications the class Screen is assumed to be modeled by an ordered 4-tuple containing the horizontal size of the screen, the vertical size, the background color of the screen, and an array of integers representing the current color of each screen pixel. Also the class Point is modeled by an ordered pair of integers, being the cartesian coordinates of the point.

*class Circle : Figure*

*is modeled by : (anchor: point, color : integer, radius : integer)*

*post.Circle(Point a, int c, int r)* $\equiv$ *self* = (*a*, *c*, *r*)

*pre.Draw(Screen s)* $\equiv$ (*self.anchor.x* $-$ *self.radius* $\geq 0$) $\wedge$ (*self.anchor.x* + *self.radius* $\leq$ *s.sizex*) $\wedge$
(*self.anchor.y* $-$ *self.radius* $\geq 0$) $\wedge$ (*self.anchor.y* + *self.radius* $\leq$ *s.sizey*)
*post.Draw(Screen s)* $\equiv$ (*Figure.post.Draw*) $\wedge$
$(\forall x \forall y(\ (0 \leq x \leq s.sizex) \wedge (0 \leq y \leq s.sizey)\ ) \Rightarrow$
$(\ trunc(sqrt((x - self.anchor.x)^2 + (y - self.anchor.y)^2)) \neq self.radius \Rightarrow$
$s.pixels[x][y] = \#s.pixels[x][y]\ ) \wedge$
$(\ trunc(sqrt((x - self.anchor.x)^2 + (y - self.anchor.y)^2)) = self.radius \Rightarrow$
$s.pixels[x][y] = self.color\ )\ )$

*post.Erase()* *similar to post.Draw*

Figure 2: Specifications of derived classes Circle and Rectangle

Figure, and one which talks about the behavior of the derived class in terms of the new conceptual state which is not present in the base class. For Circle we omit the first of these specifications and only include those clauses of the second which differ from the first. For instance, the post-condition for the method GetAnchor is the same as the post-condition for that method in the Figure class and is therefore omitted. Also, the Circle class should probably include methods to allow clients to set and test the value of the radius etc. Such methods, while easily specified, will not be necessary for our example –and cannot be used polymorphically since they are not common to all types of figures– so in the interest of brevity we have omitted them. Finally, we have left the specification of the Erase method, which closely follow that of Draw, for the interested reader to complete.

The specification of Draw in the class Circle contains two new assertions: a pre-condition and an extra clause in the post- condition. The pre-condition states that the Draw method cannot be invoked when the circle is in a state that would cause part of the circle to extend off the edge of the screen. It is important to note however, that this pre-condition is only part of the *second* specification of Circle. The class must also, of course, respect its first specification (the one for the class Figure), in which there is no pre-condition for the Draw method, otherwise we cannot be sure that results that have already been proven about existing client code will remain valid. In short, an implementation of the Draw method for Circle must exhibit the behavior defined by Figure.post.Draw so long as the circle object meets the condition expressed by Figure.pre.Draw, regardless of whether or not it meets this new pre-condition. But whenever the initial state meets the condition Circle.pre.Draw, the implementation must in addition exhibit the additional behavior defined by Circle.post.Draw. This post-condition uses the new state of the derived class, the radius, to define precisely *which* pixels on the screen are drawn by the Draw method. More specifically, the post condition states that unless the pixel is on the circle of radius self.radius centered around the point self.anchor, the pixel does not change. If the pixel is on that circle, then its final color matches that of the circle being drawn. As we have argued previously, it is precisely this kind of information about the behavior of the new classes *in terms of the conceptual state not present in the base class* that we are most interested in when we are reasoning about polymorphic client code that use these classes, and we turn to that task next.

Let us consider the following client code which moves a figure from one part of the screen s to another (we assume that fig has been declared as a reference to Figure and dx and dy are ints). The code functions by first erasing the figure object referred to by fig, then moving it to a new location on the screen, and finally redisplaying it:

*CC*:

fig.Erase(s);
fig.MoveAnchor(dx, dy);
fig.Draw(s);

The code is independent of the particular type of figure referred to by fig, relying instead on polymorphism to invoke the appropriate Erase and Draw functions.

Now we may ask the question: what results would we like to be able to establish about the behavior of *CC*? Our answer depends on what we know about the object that fig is referencing when this code is executed. It may be that we are reasoning about the behavior of *CC* without any knowledge of the various subtypes of Figure. This type of reasoning might serve to assure us that our application does not violate certain general properties. For instance, we may want to show that no red pixels are drawn on the screen, no matter what subtypes of Figure are used in the program. So we need to establish the following result:

$\{$ ObSts[obj.fig].color $\neq$ red $\wedge$ ObSts[obj.s].bcolor $\neq$ red
$\wedge$ $(\forall x \forall y($ $(0 \leq x \leq$ObSts[obj.s].sizex$)$ $\wedge$ $(0 \leq y \leq$ ObSts[obj.s].sizey$)$ $) \Rightarrow$
ObSts[obj.s].pixels$[x][y] \neq$ red $)$ $\}$
*CC*
$\{$ $(\forall x \forall y($ $(0 \leq x \leq$ObSts[obj.s].sizex$)$ $\wedge$ $(0 \leq y \leq$ObSts[obj.s].sizey$)$ $) \Rightarrow$
ObSts[obj.s].pixels$[x][y] \neq$ red $)$ $\}$ $(pc)$

We can in fact use the specification of Figure to see that the assertion $(pc)$ holds after each statement of *CC*. Since Figure.post.Erase assures us that the only pixels on the screen that are changed by Erase are changed to the screen background color, and we have the assumption that the background color of the screen is not red, $(pc)$ holds after the first statement. Figure.post.MoveAnchor asserts that the screen is unaffected by calls to MoveAnchor, so $(pc)$ holds after the second statement. Finally, Figure.post.Draw tells us the the only pixels changed by Draw are changed to the color of fig, which we know is not red. Thus $(pc)$ holds at the end of *CC*. To formally prove this result we would simply need three applications of axiom (4) from Section 2.2, one for each method call. This result can also be established using other approaches such as those of [9, 10] since it involves reasoning only about the base class conceptual state. Further, in our formalism as well as in the others, once this result is established, the restrictions imposed on the specifications of acceptable derived classes ensure that even if new derived classes are added to the system, the result will continue to hold.

What if we are aware of the different subtypes of Figure? Suppose we knew that when *CC* is executed, that fig will only refer to a Circle object. What kind of properties of *CC* would we like to be able to prove in this case? Obviously, we would be interested in results that talk about the state and behaviors that depend upon the behavior provided by the particular derived class, Circle in this instance. After all, if we were not interested in such properties, then why even design and implement these derived classes in the first place? Here is an example of a result that depends upon knowing that the figure object that fig refers to is a circle:

$\{$ (otype.fig = Circle) $\wedge$ (ObSts[obj.s].sizex - ObSts[obj.fig].anchor.x + dx $\geq$ ObSts[obj.fig].radius)
$\wedge$ (ObSts[obj.s].sizey - ObSts[obj.fig].anchor.y + dy $\geq$ ObSts[obj.fig].radius)
$\wedge$ (ObSts[obj.fig].anchor.x + dx $\geq$ ObSts[obj.fig].radius)
$\wedge$ (ObSts[obj.fig].anchor.y + dy $\geq$ ObSts[obj.fig].radius) $\}$
*CC*
$\{$ ( $\forall x \forall y$ $(0 \leq x \leq$ObSts[obj.s].sizex $\wedge$ $0 \leq y \leq$ObSts[obj.s].sizey$)$ $\Rightarrow$
( $trunc(sqrt((x-$ObSts[obj.fig].anchor.x$)^2 + (y-$ObSts[obj.fig.anchor.y$)^2$ ) ) =ObSts[obj.fig].radius
$\Rightarrow$ ObSts[obj.s].pixels$[x][y]$= ObSts[obj.fig].color ) $\}$

This result asserts that if fig refers to a Circle and if the anchor point of fig (after adding dx and dy) is sufficiently far from the edge of the screen, then *CC* will draw a circle of the appropriate radius on the screen.

Such a result cannot be established using other approaches because in these other approaches, we would be forced to reason about the three method calls in *CC* using *only* the specifications for these calls in the base class Figure. And, as we have already seen, those specifications say nothing about *which* pixels are erased or drawn on the screen so there is no way to use them to conclude that a circle is drawn. Indeed, the base class specifications are not even aware of the existence of the radius part of figure's state.

Using our formalism, however, it is easy to establish the above result. Axiom (5) from Section 2.2 allows us to use the method specifications found in the Circle class. Our assumption about the initial position of fig will allow us to show that the pre-condition for the Draw call is satisfied, and the post-condition for Draw will assert that a circle, not a rectangle or some other figure, is drawn on the screen. The formal proof of the above result will require three applications of axiom (5), one for each method call. If, on the other hand, we knew that *CC* was being applied to a Rectangle object, we would similarly be able to specify and verify the behavior appropriate to that case.

Let us now briefly consider the bank account example from [4]. The example consists of a base class BankAccount and a derived class PlusAccount. Conceptually BankAccount objects have a single balance called credit, while PlusAccount objects have two, one corresponding to the balance in the checking portion of the account, the other corresponding to the savings portion. Consider a fragment of client code that transfers money from one BankAccount object to another using standard operations such as deposit and withdraw provided by this class; this behavior can be expressed in terms of the credit in the two BankAccount objects. Now consider what will happen if this code is applied to PlusAccount objects. Using our approach it would be easy to establish how the checking and savings balances in the two accounts are affected, on the basis of the specification provided by the PlusAccount class. Indeed this would be very similar to what we went through above when considering the behavior of *CC* when applied to Circle objects (rather than arbitrary Figure objects). Dhara and Leavens observe that this is not possible using just behavioral subtyping; as we noted earlier though, this problem is not the focus of [4].

# 4 Discussion

Polymorphism is one of the most important ideas underlying the OO approach. The power of polymorphism arises because when a method is invoked on an object, the run-time system chooses the appropriate code to apply based on the class of the object, thus relieving the programmer of this burden. Indeed the programmer does not even have to make changes, or even recompile, when new derived classes are defined and the same code is applied on objects that are instances of these new derived classes. Meyer [13] calls this the 'open-closed' principle since the compiled code can be used as such (hence 'closed'), and can also be extended by adding new derived classes (hence 'open').

Other approaches such as those [1], and [10], to reasoning about the behavior of OO code require us to deal with polymorphism essentially by ignoring the differences between the different derived classes and using only the information (about the methods in question) obtainable from the base class. But often the reason for defining the derived classes is that these differences are important to the client; abstracting them away when reasoning about the client code would defeat the purpose of defining the derived classes. The solution we have proposed is to have the specification of each derived class provide information about the added functionality provided by that particular class, in the form of the extension it provides to the conceptual model (of the base class), and in the form of the effects the various methods will have on the extended portion of the conceptual model. Our formalism also requires that the derived class not violate anything that is contained in the base class specification, i.e., the effect of any method on the base class portion of the conceptual model must be consistent with the base class specification of the method; in this sense our work is an extension of the behavioral subtyping approach.[11]

[11]The formalisms of [10, 11], as well as that of [4], allow a slightly more general relation between the base class model and the derived class model. They require a mapping from the model of the derived class to that of the base class such that under this mapping the behavior of the various methods of the derived class match their specifications in the base class. We could have similarly allowed the derived class model to be an extension not of the base class model but of a model that can in turn be mapped to the base class model. More important, these other formalisms do not allow the reasoning about the client code to

But it is not sufficient to enrich the specifications of the derived classes. The formalism should also allow us, when reasoning about the client code, to make use of these richer specifications. Our axioms (4) and (5) in section 2.2 were designed for this purpose. (4) allows us to use the base class specification if we have no information about the specific derived class that the object in question is an instance of, and (5) allows us to appeal to the specific derived class specification if we do have such information. The formalism in section 2.2, in particular ObIds, the set of identities of all objects currently in existence, and ObSts, the states of these objects, etc., has also been designed to mesh well with how OO programs function and how people think about them intuitively; specifically with the fact that objects are long-lived entitites and have internal states that change. The alternative would have been to treat objects as *values* and create a new value each time the state of an object changes. While such an approach might have worked in simple examples such as the ones considered in the current paper, we believe our approach is more appropriate for realisitic situations, in particular when aliasing –two or more variables referring to the same object– is common. In such situations, if we change an object by accessing it via one of the variables that refers to it, the change will be visible when the object is accessed via another variable that also refers to it. This is exactly what will happen in our approach since the change to the object state is recorded in ObSts, not by creating a new object and associating it with the first variable.

Returning briefly to how derived classes are specified in our approach, it may seem that having two abstract specifications for each derived class method would require additional work on the part of the derived class designer but recall that one of these specifications is identical to the specification of the method in the base class. *Verifying* that the implementation of the derived class is correct will indeed require extra work, since the implementor has to show that both abstract specifications are satisfied. In general this will require two concrete specifications for each method in the derived class. This can be seen in the example of the Circle class. The problem is that a circle may be too big to fit in the given window. So the Circle class designer has decided to impose a pre-condition that this not be so. But client code that contains a call to the Draw() function that does not meet this pre-condition may already have been verified to satisfy some properties! That is because the verification may have only used the base class specification, and that specification does not impose conditions on the size of the circle, indeed the size is not part of the conceptual model of the base class. That is why the derived class designer/implementor is not at liberty to, say, flag an error if the circle is too big. Instead the derived class must in such a case take some default action (such as not making any changes in the screen) if this condition is not satisfied. Correspondingly, we have two concrete specifications for the Draw() function; one will will impose no pre-condition on the size of the circle and will guarantee that either no pixels will be changed or a complete circle will be drawn; the other will require a pre-condition that says the circle must not be too big, and will guarantee that the appropriate circle will be drawn on the screen. An alternative version, let us call it altCircle, class may draw a *clipped circle* if it is too big to fit in the screen. The abstract specification of altCircle would be different from that of Circle; it would impose no pre-condition on the radius of the circle, and would guarantee in its post-condition that either a full circle or a clipped one, as appropriate, will be drawn. Which of these derived classes is better is not particularly important; what is important is that the formalism allow, as ours does, the derived class designers to specify the added functionality provided by their classes.

One criticism that may be directed against our approach is that it forces us to do case analysis of the client code on the basis of the particular type of the object on which the code is being applied, whereas approaches like those of [10, 1] do not. But note that we have to do case analysis only to the extent that we are interested in establishing additional properties beyond the ones that can be arrived at using the base class specifications. These properties obviously depend upon the class of the object in question, indeed that is the reason for defining various derived classes. If we want to show that a given piece of code will draw a circle on the screen, we cannot expect to do that without appealing to the Circle class. An alternative that may be worth exploring would be to include in the base class specification an assertion expressing the 'expectations' that the base class designer has, of the derived classes, and use such expectations to establish stronger properties of the client code.[12] We are unaware of any systems that do this, and in any case the

make any use of the effects of the methods on the part of the derived class model that is the *extension*; that is the key difference with our approach.

[12]To a limited extent we can do this in our approach, and indeed did so in the Figure class; we required that the Draw function

approach would seem to present more problems than it solves. What, for instance, does the Figure class designer expect of the derived classes? The notion of 'drawing' seems intuitively well defined but codifying such intuition formally would seem to hamstring the derived class designer and go against the idea of 'open' systems. It has been argued that derived class designers often come up with new classes that are quite different from anything that the base class designer had in mind; if that is true, our approach is clearly preferable.

It is worth reiterating that although our approach is influenced by the model of inheritance-based polymorphism in languages like *Eiffel* and *C++*, the approach (like those of [10, 4] etc.) is applicable also in cases where polymorphism works on a different basis such as the *interface inheritance* of *Java*. Indeed most of our discussion has been in terms of abstract specifications whereas inheritance is one specific concrete mechanism for implementing polymorphism; since the details at the abstract level are independent of the implementation level mechanisms, nearly everything in this paper applies without change to these alternate mechanisms for implementing polymorphism. Of course, if the polymorphism is not class based but is instead done on the basis of, say, delegation by the object and an object can at run-time change who it delegates a particular method to, our approach will need considerable modifcation.[13] It may also be appropriate to mention here the work of Lamping and Abadi [8] in which they formalize polymorphism in a very general manner allowing method selection to depend on a variety of criteria such as on the types of all the operands (multi-methods as in *Cecil* [3] or *CLOS* [7]), or even real-time considerations (as in *DROL* [15]). But their focus is on formalizing how the method is chosen (by the run-time system) rather than on reasoning about behaviors. Nevertheless it would be interesting to see if some of their ideas can be used as a generalization of our ObIds, ObSts, etc. to arrive at a reasoning system that can handle these general polymorphism mechanisms.

We will conclude with a brief mention of two problems that may arise when dealing with class-based polymorphism that we have not addressed, and possible ways to extend our approach to deal with them. First, we have assumed that we have a base class $B$ and a set of derived classes $D_1, D_2, \ldots$. What about derived classes of derived classes? Suppose $B$ is a base class, $D1$ is a derived class of $B$ and $D2$ is a derived class of $D1$. First recall that an instance of class $D2$ may be used where an object of type $D1$ or an object of type $B$ is expected. Thus $D2$ is a derived class of both $D1$ and $B$. The conceptual model of $D2$ will, of course, be an extension of that of $D1$ which itself is an extension of $B$. Since $D2$ is a derived class of $D1$, we of course have to show that its methods satisfy $D1$'s (abstract) specification, in addition to satisfying the specification we have provided in $D2$. But we must also show that $D2$'s methods satisfy the (abstract) specification of $B$ since $D2$ objects may be used in place of $B$ objects. This seems an undesirable feature of our approach. After all, if we show that $D2$ objects satisfy $D1$'s specifications, shouldn't it follow that they also satisfy $B$'s specifications (since we have already shown that $D1$ objects satisfy $B$'s specifications)? That would indeed be true if we had explicitly included $B$'s specification as part of $D1$'s specification. Since instead we took $B$'s specification as being implicit when dealing with $D1$, checking that $D2$ objects satisfy $D1$'s (explicit) specification does not automatically guarantee that they also satisfy $B$'s specification. Apart from this detail, there is no difficulty in using our approach to deal with derived classes of derived classes.

Second, what about multiple inheritance? Suppose $B_1$ and $B_2$ are two base classes and $D$ inherits from both. Our approach can be easily extended to deal with this situation. First, the conceptual model of $D$ would be an extension of the models of both $B_1$ and $B_2$. Thus conceptually an instance $d$ of $D$ would consist of three components: $\langle b_1, b_2, e \rangle$, the first two components being instances of the models of $B_1$ and $B_2$ and $e$ is the extension provided by $D$. Next, the derived class designer would have to show that $D$ satisfies the abstract specifications of both $B_1$ and $B_2$. In addition, of course, $D$ would have its own specification that captures the behavior of the various methods in terms of all three components and the derived class

(of the derived classes) not change the color of any pixels unless the new color is the same as that of the Figure in question. This is a restriction that the base class designer can forsee must be satisfied by every conceivable derived class of Figure, and it is expressible in terms of the conceptual model of the base class. What we are talking about now is something much more; the question now is, can the specification of the Figure class dictate what kinds of figures –such as circles, etc.– may be drawn by the derived classes?

[13] *Smalltalk* does not fall in this category; although variables in *Smalltalk* are not typed, polymorphism is still on a class basis; hence our approach, with some changes given the lack of typing of variables, should apply here too. Indeed we expect ObIds and ObSts to play an even more important role here than in languages where variables are typed.

implementor would have to verify that this specification is also met. One notational extension that would be needed is to deal with the following: when an object of type $D$ is used in place of an object of type $B_1$, both the $e$ component as well as the $b_2$ component are new to the programmer of this code. Thus when understanding this code, as it functions on $D$ objects, we need to worry not just about the new effects as typically manifested by the $e$ component but also the effects on the $b_2$ component. Thus we will need to introduce appropriate formal notations that allow us to use the specification from the $B_2$ class, not just the $D$ class, when understanding such behavior, but conceptually no new ideas need to be introduced.

## 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] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 1985.

[3] C. Chambers. The cecil language: specification and rationale. Technical report, University of Washington, 1993.

[4] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE-18*, pages 27–51. Springer-Verlag.

[5] C. Horstmann. *Mastering Object-Oriented Design in C++*. Wiley, 1995.

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

[7] S. E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.

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

[9] G. Leavens and W. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32:705–778, 1995.

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

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

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

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

[14] N. Soundarajan and S. Fridella. Inheriting and modifying behavior. In *Proceedings of TOOLS, published by IEEE Computer Society (to appear)*, 1997.

[15] K. Takashio and M. Tokoro. Drol: An oo language for distributed real-time systems. In *OOPSLA*, pages 276–294, 1992.