Specification-Based Incremental Testing of Object Oriented Systems

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

March 30, 2000

Abstract

Inheritance is an important feature of the OO approach that allows a designer to easily build a new *derived* class that extends the functionality of an existing *base* class. In order to exploit the full potential of inheritance to build systems incrementally, the designer must also be able to test and reason about the behavior of the derived class in an incremental manner. In this paper we develop a specification-based technique for testing both base and derived classes, with the specification and tests for the derived class being obtained incrementally from those of the base class. Using our approach, given the *concrete specification* of a method $m()$ of a class, we can mechanically generate a simple test method for $m()$. We show how our approach can be integrated with a standard testing framework. We illustrate the approach by applying it to a simple example.

1 Introduction

The Object-Oriented approach to developing large systems is extremely powerful. Much of this power derives from the key notion of *inheritance*. Given an existing *base class*, a system designer can use inheritance to build a new *derived class* that extends, or that slightly differs from the base class. The effort involved in building a new class using inheritance from an existing base class is often significantly less than if the designer had started from scratch. But in order to exploit the full potential of inheritance to build systems incrementally, the system designer must not only be able to reuse code from the base class in building the derived class, but also be able to test and reason about the behavior of the derived class in an incremental manner. The goal of this paper is to investigate specification-based techniques for testing both base and derived classes, with the specification and tests for the derived class being obtained incrementally from those of the base class.

Abstraction, a basic tenet of the OO approach, causes, as noted by Berard [4], an important difficulty for the testing of OO systems, especially for specification-based testing. The problem may be seen as follows: Since abstraction dictates that clients of a class should not be concerned with, or even aware of, the internal details of the class, the specification of the class is usually given in an abstract manner, either using algebraic axioms [7, 9], or as pre- and post-conditions of the methods of the class that are written not in terms of the actual member variables of the class but in terms of an abstract model of the class [12, 13]. Since during the testing of the class we would presumably analyze how the values of the member variables of the class change as various member operations of the class are invoked, we have a problem of matching these values to the abstract specification of the class. The problem is even more acute when we take account of inheritance since the set of member variables and member operations (and what the operations do) in the derived class is a complex mix of items defined in the base class and those defined in the derived class.

We can try to address the problem in one of two different ways. First, we could try to recast the testing so that it too works at an abstract level; this is the approach of several authors [1, 5, 6], and we will discuss some of these in the next section. The second approach, the one we use in this paper, would be to work with a *concrete* specification of the class. We will see the complete details later in the paper, but in essence a concrete specification of a class B characterizes the behavior of each method of B in terms of pre- and post-conditions that are assertions on the member variables of B. Given such a specification, we can, as we will see, use standard techniques such as equivalence partitioning [17] to design suitable test cases for B. Next, suppose D is a derived class of B. How do we arrive at a test suite for D? Some methods may be defined (or redefined) in D while others may be inherited from B; even some of the inherited methods may exhibit behavior that is different from their behavior in the base class because of calls to methods that are redefined in D. In our approach, we proceed as follows to arrive at the test suite for D: Based on the concrete specification of B, incrementally arrive at the concrete specification of D; next, based on this concrete specification, design suitable test cases for D; some of these, given the close correspondence between the specifications for B and D, will be based on or identical to the test cases for B. Thus, just as inheritance allows us to build the code for the derived class incrementally from that of the base class, our approach allows us to incrementally specify the derived class and design a suitable test suite for it.

Designing the test suite is only part of the problem. In order for the approach to be usable practically, we need to automate as much as possible of the running of the tests and of the analysis of the results. The *JUnit* testing framework of Beck and Gamma [2] is well suited for this task. This framework makes it easy to run the tests and, because of the nature of the tests –tests typically contain `assert` statements to check the correctness of the results–, makes the analysis of the results automatic at least as long as the tests do not reveal any problems. Although JUnit was not designed with specification-based testing in mind, `assert` statements provide an ideal method of testing against specifications. As we will see later in the paper, our work also suggests some useful extensions to JUnit.

Let us return briefly to the question of abstraction. While it is true that clients of a class should not concern themselves with the internals of a class, designers of derived classes are indeed very concerned with the internals of the appropriate base classes. As we discuss in detail in [21, 22], in the absence of a concrete specification for the base class B, the designer of the derived class D that inherits from B would be forced to analyze the code of the methods defined in B in order to ensure that the methods he[1] defines (or redefines) in the derived class use the member variables inherited from B in ways that are consistent with how these variables are used by the methods defined in the base class. It is for the same reason that in designing tests (for the base class as well as for the derived class), it is extremely useful, as we have argued above, to have available, the concrete specification of the class. This is not to suggest that when testing the *client code*, we should rely on the concrete specification of the class; for that purpose, we should of course rely only on the abstract view of the class perhaps using approaches such as those of [1]; we will return to this discussion in the final section of the paper.

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

- It makes the case that a natural approach to specification-based testing of OO systems, especially when dealing with derived classes, is to base the tests on concrete specifications of the classes.
- It presents an incremental method of generating test suites using such concrete specifications for systems of base and derived classes; and applies the method to a simple example.
- It discusses how the approach can be integrated into a standard testing framework, thereby automating the running of and analysis of the tests; and considers possible extensions, suggested by the approach, to the framework.

The rest of the paper is organized as follows: In Section 2 we briefly survey related work. Section 3 is the heart of the paper. Here we show what concrete specifications of base classes look like and how we can test them on the basis of such specifications; we show how specifications and tests for derived classes can be generated incrementally from those of the base class; and we briefly explain the JUnit framework and show how we can use this framework to apply our testing technique. Section 4 illustrates the approach on a simple example. The final section summarizes the approach and provides pointers for future work.

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

2 Related Work

Perry and Kaiser [18] and Smith and Robson [19] were the among earliest to identify the problems involved in testing of OO systems. In particular, they pointed out the difficulties caused by inheritance: While inheritance allows the derived class designer to inherit the code of a method from the base class, it does not provide any guarantee at all that this code will work in cooperation with the methods he (re)defines in the derived class; suitable tests appropriate to the derived class will have to be designed to check that this is indeed so. The main goal of the current paper is of course to investigate ways of doing this.

Harrold *et al* [11] propose an algorithm that identifies which tests of the base class are applicable to the derived class. The key idea behind their work was that if $f()$ is a method of the base class and it is neither redefined in the derived class, nor does it invoke (directly or via other methods) any methods that are redefined in the derived class, then any test designed to validate this method as part of the base class, can continue to be used in the derived class[2]. Hsia *et al*'s [15] work is similar but deals not just with base and derived classes but also clients of classes; they use the notion of *class firewall* [26] to identify the parts of a system that are affected by a change in a given class. But neither [11] nor [15] discusses the question of designing the tests on the basis of specifications of the base or derived classes. Indeed, the type of tests that the results of [11, 15] apply to do not even have to be specification based.

Doong and Frankl [6] present a specification based approach to testing OO programs. Unlike in our approach, the tests that they design are based on an *abstract* specification of the class being tested. Given an algebraic specification of a class, as well as an algorithm for testing equivalence (at the abstract level) of two instances of the class, their system generates calls to operations of the class to see if two sequences of calls that are supposed to be equivalent according to the specification do indeed result in instances that are equivalent (as determined by the equivalence algorithm). An advantage of this system is that the tests are produced automatically based on the specification. But it does require the user to supply an implementation of the equivalence checking algorithm. Antoy and Hamlet [1] propose a related approach. Given an algebraic specification for an ADT A and an implementation B for it, automatically create a second implementation AA for A using a *term rewriting* approach as suggested by [10]; require a piece of (user-provided) code R that maps a concrete state of B to an abstract instance of A; as the system executes, at various points, use R to map the result produced by B to the corresponding abstract instance, and compare it with the result produced by AA (scaffolding code to invoke the right sequence of operations of AA mirroring the sequence of operations of B that are actually invoked would also have to be produced). While there are important differences between the systems of [1] and [6], one important common feature, from our point of view, of these two systems is that

[2]In *C++* [24], if $f()$ only invoked *non-virtual* methods, then even if these invoked methods were redefined in the derived class, that would not affect the behavior of $f()$ since it would continue to call the methods defined in the base class; thus the idea underlying [11] is more accurately stated as: a base class test of $f()$ can be used in the derived class if neither $f()$ nor any *virtual* method that it invokes is redefined in the derived class. In *Java* [8] and in *Eiffel* [16], methods are virtual by default.

neither addresses inheritance which is an important aspect of practical OO systems.

Derrick and Boiten [5] present a different approach. They too consider tests at the abstract level but the problem they are interested in is that of designing tests for a *refinement* R of a system S given the tests for S; specifically, they address the question of how to arrive at a finite state machine that can generate a suitable test suite for R given the corresponding machine for S. While there is some similarity between refinement and inheritance, there are also important differences: Inheritance is a language mechanism provided by OO languages to enable us to construct a derived class based on an existing base class, whereas refinement is more a relation between the *abstract specifications* of classes or ADTs; thus even if R is a refinement of S, they maybe implemented entirely independently of each other and the concrete states in these two implementations may have nothing in common with each other, indeed it may well be that only one of R and S is in fact implemented; conversely, if B is a derived class of D, there is no guarantee that the abstract specification of D is a refinement of that of B. Thus while the approach of [5] works at the level of abstract specifications of systems one of which is a refinement of the other, our work is concerned with classes one of which inherits from the other. We will return to this in the final section of the paper.

3 Specification and Testing of Base and Derived Classes

3.1 Testing Framework

How do we test a class C? Using the *JUnit* framework of Beck and Gamma [3] this may be done as follows: Design a parallel *test class*, call it TC. TC must be a derived class of the class TestCase defined in the framework. The key methods of TC will be the actual tests, and these must be named test??; for example we may have two test methods, respectively testa() and testb(). TC must include a constructor (essentially as a way to give a name to the suite of tests defined in TC so the framework can print out this name as necessary). TC must also define a setUp() method which will typically create one or more instances of C; it is these instances that the methods of C will be invoked on, by the test methods; there may also be a corresponding tearDown() method which releases any resources that may have been allocated during the test.[3]

The last method in TC is main() which (typically) works as follows: Creates an instance of the TestSuite class (another class defined in the framework), passing to the constructor function of this class the name[4] TC of our test class; this constructor function essentially constructs, using the *reflective* capabilities of the language, a *test suite* object that contains all the test methods defined in TC; next, main() passes this object to the run method of

[3]In a language like *Java*, garbage collection takes care of releasing memory, in particular memory occupied by the instances of C that were created in setUp(), so the only time we need to define tearDown() is if it is necessary to release other resources such as files; in *C++*, tearDown() would have to take care of releasing memory as well.

[4]This is not strictly correct but it is close enough without getting into too many *Java*-specific details.

the TestRunner class which is another class defined in the framework. This method will run each of the tests in the test suite, i.e., each of the test methods of TC, invoking setUp() and tearDown() respectively prior to and following the call to each test method. If all the test methods run without problems, the run() method prints a message to that effect and finishes; else, it prints messages that indicate which tests had problems. The framework also makes it possible to create a *master tester class* that makes it possible to create a suite of suite of ... tests, so that all classes can be tested in one fell swoop by invoking the main() method of this class.

We will see examples of individual test methods (defined in TC) in the next section. Typically these methods invoke appropriate methods of C, the class being tested, then compare the results with expected results; this is done using the assert mechanism; if the asserted condition is satisfied, the method will just proceed; if it is not, an appropriate message will be printed. This is well suited to our needs because we can base arguments to the assert statements on the actual specifications of the class under test.

3.2 Base Classes

Consider a base class C. C will consist of a number of member variables and a number of methods. The concrete specification of C will consist of two components. The first component is an *invariant* I, an assertion over the values of the member variables; I is expected to be satisfied at the start and end of each call to a method of C. The second component is, for each method $m()$, a specification of the form $\langle m.pre,\ m.post\rangle$. $m.pre$, the pre-condition of $m()$, is an assertion over the member variables of C as well as any parameters expected by $m()$; as usual this assertion must be satisfied by the values of these variables immediately prior to any call to $m()$. $m.post$ is an assertion over the values of these same variables at the time that $m()$ finishes, as well as their values at the time of the call; we will use the x.@pre notation of *OCL* [25] for referring, in the post-condition, to the value of the variable x at the time of the call.[5] C will have one or more constructor functions. The pre-condition of the constructor function will not impose any conditions on the values of the member variables since these values do not exist before the object is constructed; for the same reason, the invariant I will not be required to be satisfied at the start of a constructor function nor will the post-condition refer to the values that the member variables had at the start of the function.

Let us now consider the corresponding testing class TC. As we saw in Section 3.1, in TC we need to define setUp(), tearDown(), and whatever test methods are needed. Let us start with the test methods. A simple approach that will nevertheless test all the methods of C would be to have, for each method m() of C, a corresponding test method testm() in TC; but to do this, we need an instance, call it tc, of C so that we can invoke m() on it; let us assume that setUp() will create tc. We can now write testm() to simply consist of a call to m() followed by an assert statement in which we require that the post-condition $m.post$ is

[5] It is not strictly necessary to allow the post-condition to refer to the values of the variables at the time of the call but doing so often simplifies the specifications. Also, for simplicity, we assume that any additional parameters of m() are of simple types such as integers and booleans, and are passed by value-result.

satisfied. Now m() is guaranteed to work, that is ensure that its post-condition is satisfied when it finishes, only if its pre-condition was satisfied at the time of the call to it. Thus it would seem that the body of testm() should look like:

assert(*m.pre*);
tc.m();
assert(*m.post*);

But there are a number of problems with this. First, it ignores the class invariant; this is easily fixed by conjuncting IC, the invariant from the concrete specification of C, to both the asserts. Second, *m.pre*, *m.post*, and IC are all assertions involving the member variables of C; but which instance of C? Clearly, we mean the instance tc; thus we need to replace references, in *m.pre*, *m.post*, and IC to the member variables of C, by references to the corresponding variables of tc; i.e., x in any of these assertions would be replaced by tc.x. Third, are these references to the components of tc legal, i.e., will they be allowed by the compiler? The problem here is that member variables are typically declared to be protected, and are accessible only within the class –C in the current instance– that they are members of, and in derived classes of C; and TC is not a derived class of C. The solution, in *Java*, would be to put the TC class in the same *package* with C so that from within TC we would indeed have access to protected members of C; in *C++*, we could solve the problem by declaring TC to be a *friend* of C. Fourth, what if m() expects additional parameters? In that case, we would have to declare additional variables of the appropriate type and initialize them to suitable values at the start of testm(), and pass these as arguments to tc.m(). Fifth, the post-condition may contain references not only to the values that the components of tc (and additional arguments of m() if any) have when control returns from tc.m() but also their values immediately prior to the call. To deal with this, we will have to *save* the values that these variables have prior to the call, and replace references in *post* to x@pre by xold, xold being the variable in which we saved the value that tc.x had prior to the call. This assumes that member variables of C are simple types; for the more general case, we will need to make use of a *cloning* operation [16] that can make a copy tcc of tc prior to the call and replace references to x@pre by tcc.x. And finally, is it legitimate to pass to the assert statement the invariant of C and the pre- and post-conditions of m()? That depends on what these assertions look like. Clearly, if existential or universal quantifiers are involved, no implementation of assert can handle them. In actual practice, most class invariants and most method pre- and post-conditions tend to be relatively simple and are appropriate for passing to the assert statement; we will return to this point briefly in the final section.

There is one other issue we need to address. Suppose tc does not satisfy *m.pre* (let us ignore for now any additional arguments to m()). The way we have written the code for testm(), this will lead to an error indication when the test is executed since the argument to the assert statement prior to the call to m() will evaluate to *false*; but there is no reason to assume that there is a problem here. It may be that m() is not designed to be invoked on an instance of C immediately after the instance is created. Thus we need to rewrite the code for testm() so that rather than asserting that *m.pre* is satisfied prior to the call to m(), we only call m() *if* the pre-condition is satisfied. But the issue is a bit more subtle: Suppose the pre-condition is not satisfied, so we do not call m(); the way we have defined our test suite, this is the *only* test of m(), so in this case m() will actually not be tested at all! This is clearly not satisfactory.

But it should also not be surprising. Everything we have described above can be mechanized (and added to the JUnit framework); in order to solve the problem at hand, what we need to do is to somehow generate instances that would satisfy the pre-condition of m() and there is no easy way for a mechanical system to do this for an arbitrary pre-condition. Clearly designer intervention is needed to address the problem. A reasonable approach would be to use the above approach as the default[6] (in which case, as we just noted, the whole process can be entirely mechanized). Or the designer could provide an additional test method, call it testDm(), for m(); this method would be similar to testm() as we have described it above except that it would include a series of calls to some of the other methods of C to change the state of ic so that it reaches a state that does satisfy the requirements imposed by $m.pre$, and *then* calls m(). We will see an example in Section 4.

Before moving on to derived classes, let us briefly consider the setUp() method. When this method finishes, the newly created instance must satisfy the class invariant. Thus the method looks like:

$$\begin{array}{l} \mathsf{tc} = \mathsf{new}\ C(); \\ \mathsf{ttc} = \mathsf{clone(\ tc\ )}; \\ \mathsf{assert(}\ IC^{\bar{\mathsf{x}}}_{\overline{\mathsf{tc.x}}}\ \mathsf{)}; \end{array}$$

where we have used $\bar{\mathsf{x}}$ to denote all the variables of C and the superscript/subscript notation denotes variable substitution as usual. We have included the cloning operation of tc here so it does not have to be done in the individual test methods.

3.3 Derived Classes

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

Let us now consider the specification of D. First, we will have a class invariant, ID. Typically this will be a strengthening of IC, the invariant of the base class, and will consist of all the clauses of IC, conjuncted with some additional clauses concerning requirements that must be satisfied by the new member variables. But in general, ID can be quite different from IC; and our approach to testing D does not depend on any particular relation between the two invariants.

[6]It would probably be useful to provide an indication, if the pre-condition turns out not to be satisfied, that the test of the method was not in fact carried out, so the designer does not place unwarranted faith in his creation.

Next, let us consider the methods of D. If m() is a new method defined in D, we deal with it in the same way that we dealt with methods of the base class. So we will arrive at an appropriate *m.pre* and *m.post* to describe the behavior of m(). Note that in general these assertions will refer to *all* the member variables of D, those that are inherited from C as well as those newly declared in D. Given such a specification for m() and the class invariant for D, we can proceed to construct the corresponding test method testm() in the same way as in the base class. If m() is defined in C and redefined in D, then again we proceed in essentially the same fashion since the definition in C of m() has no impact on the m() of D. In such redefinitions it is often the case that what is needed is not a complete redefinition of what m() does or how it works but a relatively minor extension; in such a case, the programmer will invoke, from within the body of the m() defined in D, the m() defined in the base class. This makes no real difference as far as our approach to specification or testing is concerned.

Finally let us consider the methods inherited from C. Let us first consider the case of a non-polymorphic method m(), that is m() does not invoke any other method of C; more precisely, and getting language-specific for a moment, for *C++*, only virtual methods count here; similarly for *Java* only non-final methods and for *Eiffel* only non-frozen methods count, since final/frozen methods cannot be redefined in the derived class; in the rest of the paper, we will use the term *nfv* (non-final/non-frozen/virtual) for such methods. For a method m() that does not invoke any nfv-methods, the specification is identical to that in C, except that *m.post* can include an additional set of clauses asserting that the member variables introduced in D have the same value when m() finishes as when it starts. As far as the test is concerned, it will be essentially identical to the base class test; the only change will be to include, in the assert statement following the call to m(), the requirement that the values of the member variables introduced in D are unchanged due to the call to m(); even this is not strictly necessary since it is impossible for these variables to change since the m() defined in the base class does not have access to them.

Before we consider the last kind of method of D, the polymorphic method inherited from C, we need to briefly address one important aspect of derived classes that we have not mentioned so far. Much work on reasoning about OO systems requires that D be a *behavioral subtype* [14] of C; essentially behavioral subtyping requires, for each method, that the method as specified in the derived class, also satisfy its base class specification. The reason this is important is that if this condition is satisfied, then any reasoning that we may have performed concerning the behavior of a piece of client code CC will continue to be valid if we were to replace, in CC, instances of the base class by instances of the derived class; this is because the methods applied to these instances, although they will be the ones from the derived class, also satisfy their base class specifications, and it is these specifications we must have appealed to in our original reasoning concerning the behavior of CC. Similarly, any conclusions we may have arrived at concerning the behavior of a polymorphic method pm() in the base class will continue to be accurate in the derived class since although the nfv-methods invoked by pm() will be the ones defined in the derived class, these definitions will also satisfy their base class specifications (and it is these specifications that must have been used in our original reasoning about the behavior of pm()). But note that this ignores the fundamental reason for defining D in the first place, this being that we want to extend or enrich the behaviors provided by the methods as defined in the base class C. Inheritance allows us to inherit those methods of C that are satisfactory and redefine those that we wish to extend; and polymorphism, via run-time binding of the calls to nfv-methods, allows even those methods that are inherited unchanged to exhibit richer behavior by invoking the redefined methods. This is the source of the real power of the OO approach but behavioral subtyping by itself does not allow us to account for this richer behavior.

In [20, 23] we have proposed a technique for accounting for this richer behavior. The key idea is to include, in the specification of the polymorphic methods, information about the sequence of calls to nfv-methods that they make. This is provided in the form of a *trace* of calls; each element of τ_{pm}, the trace associated with the method pm(), corresponds to a single call to and return from an nfv-method and provides the following information: the name of the method called; the values of the parameters passed to the method; the state of the object, i.e., the values of all the member variables of the class, at the time of the call; the result values of the parameters when the method returns; and the state of the object at the time of the return. Including information about τ_{pm} in the specification of pm() allows us to 'plug-in', when reasoning about the derived class, the derived class behavior of the called methods, to arrive at the richer behavior that pm() exhibits as a result of the redefinitions of the methods it invokes. In particular, as we will see in an example in the next section, this can include changes to the values of member variables introduced in D.

Let us now consider the testpm() method. First consider the base class. The key point is that since *pm.post* is an assertion not just over the member variables of C but also over the τ_{pm}, we need to have access to the value of this variable. So far our test methods have simply invoked the method they are responsible for testing, waited for this invocation to finish, and then checked whether the state at that point satisfies the appropriate post-condition. But τ_{pm} is not part of the state; nor is it possible to calculate the final value of τ_{pm} when pm() finishes from the state at that point (or the state prior to the call to pm()). In fact, the only way to obtain τ_{pm} is to 'shadow' the execution of pm(); i.e., we introduce an extra sequence variable τ_{pm}; during its execution, whenever pm() makes a call to an nfv-method, we add an element to τ_{pm} containing the required information (the components of this element that concern the parameter values and the state when the nfv-method returns will, of course be added only when the method returns to pm()).

How do we achieve this shadowing? One possibility would be to rewrite our testpm() as follows: At the start of testpm(), initialize τ_{pm} to the empty sequence; replace the call to pm() by the entire body of that method, and within this body add, immediately prior to and immediately after each call to an nfv-method, the appropriate statements that will append, to τ_{pm}, the element corresponding to this call/return. While this technique works, and gives us the most direct test of the specification of pm(), it requires that we have the body of pm() available to us; this is not always the case, in the case for example where the method was purchased from a software vendor (OO frameworks often include such methods) and the vendor is unwilling to provide a copy of the source code. A less demanding testpm() would be our original version with its call to pm() with one important change: the argument to the assert statement following this call would not be *pm.post* since this assertion involves τ_{pm}. Instead, we will have to use a simpler assertion that refers only to the regular member variables of C and the parameters of pm(), and is *implied* by *pm.post*. In the rest of the paper, we will assume that testpm() is indeed of this simpler form.

What about the derived class? Although pm() has not been redefined in D, we cannot use the same testpm() in the test class TD since in the derived class we need to test that the *richer* specification of pm(), resulting from the redefined nfv-methods that pm() invokes. All we need to do though is to replace the argument to the assert statement following the call to pm() by the specification appropriate to D. As far as the test method is concerned, it of course makes no difference whether this specification was arrived at informally on the basis of our intuitive understanding of pm() and the nfv-methods it invokes or via the more formal approach described in [20, 23].

Before concluding this section, there is one final issue we need to consider. The derived class D has access to the protected members of the base class C as per the standard rules of inheritance. But the test class TD is not a derived class of C; so methods such as the testpm() that we just discussed that are members of this class will not have access to the protected members of C unless all these classes are in the same package. While this maybe a possiblity for small systems, it is unreasonable or even impossible for large systems, as well as for situations where C was obtained from a software vendor. One possible solution would be to introduce, into OO languages, a privileged type of access that testing software should be allowed to have.

4 Example

We will consider a simple bank account example in this section to illustrate our approach to specification-based testing of base and derived classes. The definition (in *Java*-syntax) of the BankAccount class appears in Figure 1. Deposit and Withdraw, as their names suggest, are used to deposit money into and withdraw money from the account. It is expected that in derived classes, these methods might be redefined, which is why they are not flagged as final. GetBalance returns the current balance in the account and is not expected to be redefined.

monthlies[] is an array of transactions that are expected to be performed once every month; new transactions may be added to this array by calling AddMonthly(). NewMonth() is the method that will perform all the transactions in this array and is expected to be invoked once a month. When NewMonth() is invoked, it calls Deposit and Withdraw to perform the transactions; so if these methods are redefined in the derived class, that will also affect the behavior of NewMonth().

Let us consider specifications of some of these methods:

$$\begin{aligned}
&\mathsf{Deposit(amt)}.\mathit{pre} \equiv (\mathsf{amt} > 0)\\
&\mathsf{Deposit(amt)}.\mathit{post} \equiv\\
&\quad (\mathsf{bal} = \mathsf{bal@pre} + \mathsf{amt}) \land !\{\mathsf{nmonthlies}, \mathsf{monthlies}, \mathsf{amt}\}
\end{aligned} \tag{1}$$

The @pre notation in the post-condition refers to the value of the variable in question at the

```
class BankAccount {
      protected int bal;
                  // current account balance
      protected nmonthly;
                  // no. of monthly transactions
      protected int monthlies[];
                  // transactions which happen every month

      BankAccount( )
            { bal = 0; monthlies = new int[100]; nmonthlies = 0; }
      public void Deposit(int amt)
            { bal = bal + amt; }
      public void Withdraw(int amt)
            { bal = bal − amt; }
      public final int GetBalance() { return bal; }
      public final void AddMonthly(int amt)
            {     if (monthlies.size() <= nmonthlies)
                        { monthlies.grow(5); }
                  monthlies[nmonthlies++] = amt; }
      public final void NewMonth()
            {     for (int i = 0; i < nmonthlies; i++) {
                        if (monthlies[i] > 0)
                              { this.Deposit(monthlies[i]); }
                        else { this.Withdraw(−monthlies[i]); } }
            }
}
```

Figure 1: Class *BankAccount*

time of the call to the method. The "!" notation[7] states that the variables included inside the braces have the same values as at the start of the method; thus !{x,y} is equivalent to $((x = x@pre) \wedge (y = y@pre))$. Thus the specification (1) states that the amt to be deposited must be positive, and that when Deposit finishes execution, it would have appropriately updated bal and that the other variables, nmonthlies etc., remain unchanged.

The testDeposit() method is easily written:

```
if( Deposit(x).pre' ) {
      tc.Deposit(x);
      assert( Deposit(x).post' );
}
```

where tc is the test BankAccount object created by the setUp() method; x is an arbitrary amount that is passed as the argument value in this call to Deposit(); we have used the

[7]This is not part of the OCL[25] standard but it is useful.

notations *pre'* and *post'* to indicate that the substitutions we discussed in Section 3.2 have been performed; so, for example, all occurrences of member variables such as bal in these assertions must have been replaced by tc.bal; all occurrences of bal@pre in *post* must have been replaced by tcc.bal, tcc being a copy of tc that setUp() created.

Since x is an additional argument to this method, it will have to be initialized prior to the if statement. If it is initialized to 0 or less, the test will be skipped (since the pre-condition will not be satisfied). This is similar to the problem we mentioned near the end of Section 3.2 with regard to the initial state of tc being such that the pre-condition of the method was not satisfied. The solution is similar; that is, rather than generating an arbitrary value for x (which is what a completely automated test framework is likely to do), the designer will have to initialize x to a positive value to ensure that the test is actually carried out.

What happens if the implementation of Deposit() does not meet its specification, say, for example, we write the body of the method so that a service charge for carrying out the transaction is deducted from bal? In that case, the condition $(\mathsf{tc.bal} = \mathsf{tcc.bal} + amt)$ will not be satisfied, the testing framework will print a message stating that this particular assert statement in the testDeposit() method failed. This will be our indication that there is a problem in the implementation of Deposit().

Let us now consider the method NewMonth().

$$\mathsf{NewMonth}().pre \equiv true$$
$$\mathsf{NewMonth}().post \equiv$$
$$(\mathsf{bal} = \mathsf{bal@pre} + \Sigma_{i=0}^{\mathsf{nmonthlies}-1}\mathsf{monthlies}[i]) \land !\{\mathsf{nmonthlies}, \mathsf{monthlies}\} \tag{2}$$

This specification asserts that the NewMonth() method updates bal as per the various deposit and withdraw transactions stored in the monthlies[] array, and does not change any of the other member variables.

The testNewMonth() method can be written in the same way as the testDeposit() method. In one respect it is, in fact a bit simpler since the pre-condition is *true* so we do not need the initial if. On the other hand, the post-condition is more complex since it involves the Σ operation that sums all the elements of the monthlies array. This operator is (generally) not included in the language, so we will have to provide a function that can return the sum of the elements of a given array and express the argument to the final assert in terms of this function.

The more interesting question is what happens if we redefine the Deposit() and Withdraw() methods in the derived class. Consider the DBankAccount defined in Figure 2. tCount maintains a count of the transactions that the account has gone through. We have redefined Deposit() and Withdraw() to update this count (using the base class methods for carrying out the rest of the task). The revised specification of Deposit() will be:

$$\mathsf{Deposit(amt)}.pre \equiv (\mathsf{amt} > 0)$$
$$\mathsf{Deposit(amt)}.post \equiv$$
$$(\mathsf{bal} = \mathsf{bal@pre} + \mathsf{amt}) \land (\mathsf{tCount} = \mathsf{tCount@pre} + 1) \land !\{\mathsf{nmonthlies}, \mathsf{monthlies}, \mathsf{amt}\} \tag{3}$$

We can write the corresponding testDeposit() method (this will be a member of TDBankAccount, the corresponding test class) in a straightforward way. Let us consider NewMonth().

```
public class DBankAccount extends BankAccount {
      protected int tCount; // Transaction count
      DBankAccount()
            {     tCount = 0; }
      public void Deposit(int amt)
            {     super.Deposit(amt); tCount++; }
      public void Withdraw(int amt)
            {     super.Withdraw(amt); tCount++; }
}
```

Figure 2: Class *DBankAccount*

From the specification (2) of this method, we cannot conclude anything about what it will do to tCount. Consider the following specification of NewMonth():

$\mathsf{NewMonth}().pre \equiv true$
$\mathsf{NewMonth}().post \equiv$
$(\mathsf{bal} = \mathsf{bal@pre} + \Sigma_{i=0}^{\mathsf{nmonthlies}-1}\mathsf{monthlies}[i]) \wedge !\{\mathsf{nmonthlies}, \mathsf{monthlies}\} \wedge$
$(\#\tau = \mathsf{nmonthlies} \wedge (\forall k \leq \#\tau.\tau[k].n \in \mathsf{Deposit}, \mathsf{Withdraw}))$ (4)

τ is the trace associated with NewMonth (since this is the only polymorphic method, there is no need to decorate the τ with the name of the method). $\#\tau$ is the length, that is the number of elements in τ; $\tau[k]$ is the k^{th} element of τ; $\tau[k].n$ is the name of the method called in the invocation represented by this element.

The specification (4) tells us, in addition to the information in (2), that during its execution, NewMonth makes as many calls to nfv-methods as the value of nmonthlies and that each call is to either Deposit or to Withdraw. Note that (4) does imply (2), so given the above specification of NewMonth() in the BankAccount class, the testNewMonth() that we described as part of that class is appropriate according to the approach we saw in Section 3.3. What about the testNewMonth() method for the TDBankAccount class? A simple analysis of (4) and (3) (and similar specification of Withdraw()) tells us that the NewMonth() in the class DBankAccount will increment tCount by nmonthlies. We could arrive at this conclusion either informally by inspection of (4) and (3), or via the formal rules described in [20, 23]. In either case, we have the following specification for NewMonth() of the derived class:

$\mathsf{NewMonth}().pre \equiv true$
$\mathsf{NewMonth}().post \equiv$
$(\mathsf{bal} = \mathsf{bal@pre} + \Sigma_{i=0}^{\mathsf{nmonthlies}-1}\mathsf{monthlies}[i]) \wedge !\{\mathsf{nmonthlies}, \mathsf{monthlies}\} \wedge$
$(\mathsf{tCount} = \mathsf{tCount@pre} + \mathsf{nmonthlies}$ (5)

We can write the corresponding testNewMonth() method of TDBankAccount class in a straightforward way.

5 Discussion

Our main goal in this paper was to develop a specification-based method for testing the methods of both base and derived classes in an OO system. A key point of departure for our approach compared to other specification-based testing methods such as those of [1, 5, 6] was that whereas these approaches are based on abstract specifications whereas our approach is based on the concrete specifications of the classes being tested. We did this for two reasons: First, our interest was in testing implementations of the classes in question, rather than client code that uses the classes. Since the implementation of the class is of course in terms of the concrete state rather than an abstract model of the class, our testing procedure also had to work at the level of the concrete specification. But there is need for working with both abstract and concrete specifications. If, for example, some of the components of the class C we are testing are, in fact, instances of other classes C', C'', (rather than being simple entities such as integers), then in testing C, we would of course rely on the abstract specifications of C', C''.

The second reason for basing the approach on concrete specifications was our interest in inheritance; for many designers, inheritance and the polymorphism it enables via dynamic binding, is the embodiment of the OO approach. Many systems make heavy use of inheritance to define hierarchies of classes, each building on the classes above it. We tailored our approach to be able to deal with specifying and testing derived classes on the base of the specification and testing of the base class; and since inheritance works at the level of the concrete state, this meant working with concrete specifications.

As it turned out, the JUnit framework of Beck and Gamma [2] was well suited for our approach since the test methods in that framework utilize the `assert` facility of *Java/C++* which meant we could simply use our specifications as arguments to the `assert` statements. But `assert` is of course a limited facility; if specifications involved complications such as quantifiers, we can only assert their simpler counterparts. There was one other problem that we faced in writing our test methods. This had to do with the fact that our post-conditions referred to the state at the end of the method as well as at the start. This meant that we had to *save* the starting state so that could check that the post-condition was indeed satisfied when the method finished. This requirement too fitted in well with the JUnit framework since the `setUp()` method of the framework provides the ideal point for doing this.

A more difficult problem had to do with our use of traces to specify the behaviors of polymorphic methods. The problem here was that unlike in the case of the initial state, the value of the trace had to be updated at every call to a nfv-method; and since these calls may be scattered throughout the polymorphic method, this cannot be done at one point such as in `setUp()`. Our solution was to not try to check for the entire specification but only the parts that dealt with the other variables. Note that the specification still gives us information about the trace; that is what allows us, in the derived class, to arrive at the richer behavior of the polymorphic method in the derived class, resulting from the redefined nfv-methods that it invokes. Once we have written down the specification corresponding to this richer behavior, the corresponding test method is obtained in a straightforward manner.

But one point must be stressed. The test methods that we wrote down were extremely simple. What was good about them was that they could be produced essentially mechanically. But they test each method independently and they test each method on a newly created object. If there are problems that would manifest themselves following a sequence of method calls on a given object, such problems would not be caught by our simple tests. But there is nothing inherent in the approach which requires us to use only these simple tests. A designer who wishes to do a more complete job would be easily able to alter the tests to do so; all that would be needed would be to make several calls, in the test methods, to the various methods of the class, rather than just the single call that we now have. Indeed, it may be appropriate to write a test method that contains what the designer might consider a 'typical' sequence of calls that might be expected in practice. Following each such call would be an `assert` statement requiring that the post-condition of the method that just completed is satisfied. Alternately, the designer may wish to use standard techniques such as *equivalence partitioning* to produce an appropriate collection of calls to test the method in question adequately.

We conclude with a couple of pointers to future work. In Section 3 we noted the problem that the test class TD corresponding to a derived class D is not itself a derived class of B, the base class of D. Thus in TD we would not normally have access to the protected member variables of B. We need to address this problem; the solution of putting all the classes in one package is rather unsatisfactory. Second, it would be interesting to extend the JUnit framework to provide for automated support for dealing with the trace of method calls of polymorphic methods to nfv-methods. As we have said, much of the power of the OO approach derives from this feature of OO languages; a good testing technique should be able to directly deal with the assertions concerning this key aspect of OO systems.

References

[1] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Trans. on Software Eng.*, 26:55–69, 2000.

[2] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), 1998.

[3] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[4] E. Berard. *Essays on object oriented software engineering.* Prentice-Hall, 1993.

[5] J. Derrick and B. Boiten. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability*, 9:27–50, 1999.

[6] R. Doong and P. Frankl. The astoot approach to testing object-oriented programs. *ACM Trans. on Software Eng. and Methodology*, 3:101–130, 1994.

[7] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T. Yeh, editor, *Programming Methodology, Vol. 4*, pages 80–149. Prentice Hall, 1978.

[8] J. Gosling, B. Joy, and G. Steele. *Java Language Specification.* Addison-Wesley, 1996.

[9] J. Guttag and J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[10] J. Guttag, E. Horowitz, and D. Musser. Abstract data types and software validation. *Comm. ACM*, 21:1048–1064, 1978.

[11] M. Harrold, J. McGregor, and K. Fitzpatrick. Incremental testing of oo class structures. In *14th ICSE*, pages 68–80, 1992.

[12] C.A.R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

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

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

[15] P. Hsia *et al.* A technique for the selective revalidation of oo software. *Software Maintainence: Research and Practice*, 9:217–233, 1997.

[16] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[17] G. Myers. *The art of software testing.* John Wiley, 1979.

[18] D. Perry and G. Kaiser. Adequate testing and oo programming. *Journal of Object Oriented Programming*, 2:13–19, 1990.

[19] M. Smith and D. Robson. Oo programming – the problems of validation. In *Int. Conf. on Software Maintainence*, pages 272–281, 1990.

[20] N. Soundarajan. Understanding frameworks. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Object-Oriented Application Frameworks*, pages 289–308. John-Wiley, 1999.

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

[22] N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Fifth International Conf. on Software Reuse*, pages 206–216. IEEE Computer Society, 1998.

[23] N. Soundarajan and S. Fridella. Framework-based applications: From incremental development to incremental reasoning. In B. Frakes, editor, *Sixth International Conf. on Software Reuse*, LNCS. Springer-Verlag, 2000.

[24] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.

[25] J. Warmer and A. Kleppe. *The Object Constraint Langauge.* Addison-Wesley, 1999.

[26] L. White and H. Leung. A firewall concept for both control-flow and data-flow in regression and regression integration testing. In *Int. Conf. on Software Maintainence*, pages 262–271, 1992.