Testing Components

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

Abstract

Our goal is to investigate specification-based approaches to testing OO components. That is, given a class C and its specification, how do we test C to see if it meets its specification? Two important requirements that we impose on the testing approach are that it must not require access to the source code of the class under test; and that it should enable us to deal incrementally with derived classes, including derived classes that exploit polymorphism to extend the behavior of the base class. In this paper, we report on our work towards developing such a testing approach.

1. INTRODUCTION

Our goal is to investigate specification-based approaches to testing OO components. Suppose we are given an implementation of a class C and the specifications of its methods in the form of pre- and post-conditions (and possibly a class invariant). How do we test the implementation of C to see if it meets its specifications? We are not specifically interested in the question of how to choose a broad enough range of test cases [12] although that would, of course, have to be an important part of a complete testing methodology for OO systems. Rather, we want to develop a general approach that can be used to test that C meets its specifications. Once we do this, we should be able to combine it with an appropriate methodology for choosing test cases.

We impose two important requirements on the testing approach. First, as far as possible it must not require access to the source code of the class under test. This is important if we are to be able to test not just components we designed and implemented but components that we may have purchased from a software vendor. Second, the testing approach should enable us to deal incrementally with derived classes, including derived classes that exploit polymorphism to extend the behavior of the base class. Much of the power of the OO approach derives from the ability to develop systems incrementally, using inheritance to implement derived classes that extend the behavior of their base classes. To best exploit this incremental nature of OO, our approach to reasoning about and testing the behavior of such classes should also be correspondingly incremental. In this paper, we report on our work towards developing such a testing approach.

In the next section, we provide a more detailed statement of the problem. In Section 3, we outline how the behavior of derived classes that use polymorphism to enrich base class behavior may be established in a verification system. In Section 4 we show how our testing approach can work with the kind of specifications used in the verification system. In Section 5, we briefly consider some problems related to testing classes that have components that may themselves exploit polymorphism.

2. BACKGROUND AND MOTIVATION

An important tenet of the OO approach is *abstraction*. Thus a client of a class should have an abstract view of the class, rather than thinking in terms of the concrete structure, i.e., the member variables, of the class. Correspondingly, the specification of a class C usually consists of pre- and post-conditions of the methods of C, in terms of an abstract or conceptual model of C. But abstraction causes an important difficulty [2] for specification-based testing[1]. When testing, we have to analyze how the values of the member variables of the class change as various member operations are invoked, so we have the problem of matching these values to the abstract specification. Inheritance exacerbates the problem since the set of variables and operations in the derived class is a complex mix of items defined in the base and derived classes.

Given this, in our approach to testing, we work with *concrete* specifications for the classes. This is not to suggest that abstract specifications are not important. It is just that when considering and testing the behavior of the *implementation* of C, the concrete state of the class has to play an important role since that is what the implementation works with. Similarly, when considering the behavior of the derived class, we (the designer of the derived class as well as the tester) must keep in mind the concrete state of both the base and derived classes. When dealing with the behavior of some *client* code *cc* that uses C, we should of course not think in terms of the concrete state of C; later in the paper, we will see how abstract specifications (of C) enter the picture when considering testing of *cc*.

[1]In this paper, by 'component', we will generally mean 'class' as in a typical OO language.

OOPSLA 2001 Workshop on Specification and Verification of Component Based Systems Oct. 2001 Tampa, FL, USA

The concrete specification of C characterizes the behavior of each method of C in terms of pre- and post-conditions that are assertions on the member variables of C. The specification may also include an invariant, although for simplicity we will usually ignore it in our discussion. Our goal is to create a *testing class* TC corresponding to C that will allow us to test the class C against this concrete specification. We note that the word 'testing' in the title of the paper may be considered a verb since we are interested in testing the behavior of C; it may also be considered an adjective since our approach to testing is to construct the testing component TC.

In [16], we had suggested the following simple approach to the construction of TC: For each method m() of C, include a corresponding test method test_m() in TC that will invoke m(). To do this we need an instance object, call it tc, of type C. More precisely tc is a member variable of TC of type C, and its value will be (a reference to) an object of type C. Let us assume that the constructor of TC has initialized tc to such a value. We can now write test_m() to simply consist of a call to m() followed by an assert statement in which we require that the post-condition *post.m* is satisfied. Now m() is required 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 a natural definition of the body of test_m() is:

```
if ( pre.m ) {  tc.m(); assert( post.m );  }
```

Since the object here is tc, references to a variable x of C in *pre.m*, *post.m* should be replaced by to tc.x. Are these references legal? Member variables are typically protected, and accessible only within C (and derived classes). In *Java* [1], we could put TC in the same *package* with C and give data members package scope. It is not clear how to address this in other languages; we had suggested in [16] that it may be useful to introduce the privileged notion of *test class* into the language, with the methods of the test class being given access to the members of the class[2]. Another point is that *post.m* may contain references to the value of tc.x at the time of the *call*. So we need to *save* this value in, say, xold, and replace (in *post.m*) x@pre by xold; in general, we need to use a *cloning* operation [11] for this purpose. Yet another issue has to do with the form of the assertions. Given that we want the assertions to be machine checkable, they have to have a somewhat restricted form [5, 8]. One possibility [11] would be to require that the assertions be legal boolean expressions allowed by the language. Here we will just assume that simple assertions, including quantifiers over finite domains, are allowed.

In this paper, we want to focus on a different issue. Suppose again that D is a derived class of C. Some methods may be defined (or redefined) in D while others may be inherited from C. Most importantly, 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. Following the design patterns literature [6], we will call such methods *template* methods, the methods they invoke that may be redefined in D being called *hook* methods. Let $t()$ be a template method of C, and $h()$ a hook method that $t()$ invokes. As we just noted, redefining $h()$ in the derived class enriches also the behavior of $t()$. When reasoning about $t()$ in the base class, we would have appealed to the base class specification of $h()$ to account for the effects of the calls that $t()$ makes to $h()$. In order to be sure that the conclusions we have reached about the behavior of $t()$ apply also to its behavior in the derived class despite the redefinition of $h()$, we have to require that the redefined $h()$ satisfies its base class specification; this requirement is the essence of *behavioral subtyping* [10, 4]. But ensuring that $t()$ continues to behave in a way that is consistent with it its base class specification is only part of our concern. The reason that we redefined $h()$ in the derived class was to thereby enrich (as we will see even in the simple example later in the paper) the behavior of $t()$. Therefore we need to be able to reason incrementally about this enriched behavior and, more to the point of this paper, we need to be able to test the enriched behavior that $t()$ exhibits (or is expected to exhibit) in the derived class as a result of the redefinition of $h()$.

First let us consider how the behavior of $t()$ may be specified in the base class so that we can reason incrementally about it in the derived class. The approach used in [3, 15] to specify the behavior of $t()$ in the base class is to include suitable information, in its specification, about the sequence of hook method calls $t()$ makes during its execution. This information is in the form of conditions on the value of the *trace* variable τ associated with $t()$ that records information about these calls. This information can then be used [15] to arrive at the richer behavior to $t()$ in the derived class[3] by combining it with the derived class specification of $h()$. In this paper we will see how we can test the implementation of C and D, in particular the code of $t()$ (and $h()$), to check whether it satisfies this richer specification about its behavior.

This is a challenging task because we need to keep track of the value of τ. Every time $t()$ makes a call to $h()$ (or another hook method), τ has to be updated to record information about this call (and return) but, of course, there is nothing in the code of $t()$ to do so. After all, τ is a variable introduced by us in order to help reason about the behavior of $t()$, not something included by the designer of C. One possible solution to this problem would be to *modify* the code of $t()$ to include suitable (assignment) statements, immediately before and after each hook method call, that would update τ appropriately. But this would violate our requirement that we not assume access to the body of C, and certainly not modify it. As we will see, it turns out that we can, in fact, exploit polymorphism in the same way that template methods do, to address this problem.

[2]In *C++*, we could simply declare TC a friend of C but, as is widely recognized, the friend mechanism is subject to serious abuse.
It is also worth noting that if the state of tc is such that *pre.m* is not satisfied, the body of test_m() would be entirely skipped; this may be considered a truly extreme instance of poor test-case-choice!

[3]Ruby and Leavens [14] (see also earlier work by Kiczales and Lamping [7, 9]) present a formalism where *some* additional information about a method beyond its functional behavior is provided; this may include, for example, information about the variables the given method accesses, the hook methods it invokes, etc. While this is not as complete as the information we can provide using traces, it has the important advantage that it is relatively easy to build tools that can exploit this information, or indeed even mechanically extract this type of information from the code, rather than having to be specified by the designer.

3. INCREMENTAL REASONING

Let us consider a simple example consisting of a bank account class as the base class (and a derived class we will define shortly). The definition (in *Java*-like syntax) of the Account class appears in Figure 1. The member variable bal

```
class Account {
  protected int bal; // current balance
  protected int nautos; // no. of 'automatic' transactions
  protected int autos[]; // array of automatic transactions
  public Account() { bal = 0; nautos = 0 ; }
  public int getBalance() { return bal; }
  public void deposit(int a) { bal += a; }
  public void withdraw(int a) { bal −= a; }
  public final void addAuto(int a) {
    autos[nautos] = a; nautos++; }
  public final void doAutos( ) {
    for (int i=0; i < nautos; i++) {
      if (autos[i] > 0) { deposit(autos[i]); }
      else { withdraw(autos[i]); } } }
}
```

Figure 1: Base class Account

maintains the current balance in the account. The methods deposit(), withdraw(), and getBal() are defined in the expected manner. Their concrete specifications[4] are easily given:

$$\begin{aligned}
&\text{pre.Account.getBalance}() \equiv \mathit{true}\\
&\text{post.Account.getBalance}() \equiv\\
&\quad [\ !\{\text{nautos, autos, bal}\} \wedge (\text{result} = \text{bal})\ ]\\
&\text{pre.Account.deposit(a)} \equiv (\text{a} > 0)\\
&\text{post.Account.deposit(a)} \equiv\\
&\quad [\ !\{\text{nautos, autos, a}\} \wedge (\text{bal} = \#\text{bal+a})\ ]\\
&\text{pre.Account.withdraw(a)} \equiv (\text{a} > 0)\\
&\text{post.Account.withdraw(a)} \equiv\\
&\quad [\ !\{\text{nautos, autos, a}\} \wedge (\text{bal} = \#\text{bal−a})\ ] \qquad (1)
\end{aligned}$$

In the post-conditions we use the notation "!S" to denote that the value of each of the variables that appears in the set S is the same as it was at the start of the method in question. The "#"notation, also in the post-condition, is used to refer to the value of the variable at the start of the execution of the method. Thus these specifications simply tell us that deposit() and withdraw() update the value of bal appropriately and leave the other variables unchanged; and getBalance() returns the balance in the account and leaves all variables unchanged. The notation result [11] in the post-condition refers to the value returned by the function in question.

More interesting are the 'automatic transactions'. The autos[] array maintains the current set of automatic transactions, nautos being a count of the number of these transactions. doAutos() is the (only) template method of this class. Whenever it is invoked, it performs each of the transactions in the autos[] array by invoking the hook methods deposit() and withdraw(). A positive value for an array element denotes a deposit, a negative value denotes a withdrawal. Thus doAutos() iterates through the elements of this array, invoking deposit() if the element in question is positive and withdraw() if it is negative. addAuto() allows us to add another transaction to the autos[] array. We will leave the precise specification of addAuto() to the interested reader; its pre-condition would require the parameter value to be not equal to 0, the post-condition would say that autos[] array is updated to include this value at the end of the array (and nautos is incremented by 1).

Let us now consider the specification of doAutos(). An obvious specification for this method would be:

$$\begin{aligned}
&\text{pre.Account.doAutos}() \equiv \mathit{true}\\
&\text{post.Account.doAutos}() \equiv\\
&\quad [\ !\{\text{nautos, autos}\} \wedge\\
&\quad (\text{bal} = \#\text{bal}+(\Sigma(k = 0 \ldots \text{nautos} - 1).\, \text{autos}[k])\ ] \qquad (2)
\end{aligned}$$

This specifies that doAutos() updates bal appropriately. What is missing is information about the hook method calls that it makes during execution. As a result, although (2) is correct in what it specifies, it proves inadequate in allowing us to reason about the enriched behavior that this method will exhibit in the derived class, to which we turn next.

```
class NIAccount extends Account {
  protected int tCount; // transaction count
  public NIAccount() { tCount := 0; }
  public void deposit(int a) { bal += a; tCount++; }
  public void withdraw(int a) { bal −= a; tCount++; }
  public int getTC() { return tCount; }
}
```

Figure 2: Derived class NIAccount

The enrichment provided by NIAccount (for 'New and Improved account'!) is fairly simple: it keeps a count of the number of transactions (deposits and withdrawals) performed on the account. This is achieved by redefining deposit() and withdraw() appropriately[5]. The newly defined method, getTC() allows us to find the value of the transaction count. The specifications of these methods are straightforward modifications of (1). We will only write down the specs for getTC() and deposit():

$$\begin{aligned}
&\text{pre.NIAccount.getTC}() \equiv \mathit{true}\\
&\text{post.NIAccount.getTC}() \equiv\\
&\quad [\ !\{\text{nautos, autos, bal, tCount}\} \wedge (\text{result} = \text{tCount})\ ]\\
&\text{pre.NIAccount.deposit(a)} \equiv (\text{a} > 0)\\
&\text{post.NIAccount.deposit(a)} \equiv\\
&\quad [\ !\{\text{nautos, autos, a}\} \wedge (\text{bal} = \#\text{bal+a})\\
&\quad \wedge (\text{tCount} = \#\text{tCount+1})\ ] \qquad (3)
\end{aligned}$$

Let us now turn to the behavior of doAutos() in the NIAccount class. It is clear from the body of this template method, as defined in the base class, that during its execution, the value of tCount will be incremented by the number of transactions in the autos[] array, i.e., by the value of nautos, since doAutos() carries out each of these transactions by invoking deposit() or withdraw(). But we cannot arrive at this conclusion from its specification (2), not even given the specification (3) for the behavior of the redefined

[4]This class is so simple that its abstract specification would essentially be the same as its concrete specification. Note also that we have included the name of the class in the specs since we will also consider the behavior of these methods in the derived class. Thus, (1) specifies the behavior of these methods when applied to an instance of the Account class.

[5]If these methods were at all complex, it would have been appropriate to invoke the base class methods in their definitions; here, the only task to be performed by the base class portion is to update bal, so we have just repeated the code.

hook methods that doAutos() invokes. The problem is that there is nothing in (2) that in fact tells us that doAutos() invokes deposit() or withdraw(). Indeed, if we rewrote the body of doAutos() so that it directly added each element of the autos[] array to bal, instead of invoking deposit() and withdraw() to perform the transactions, it would still satisfy the specification (2) but, of course, this rewritten method, in the NIAccount class (i.e., when applied to a NIAccount object) would *not* change the value of tCount.

Consider the following more informative specification:

$$\begin{aligned}&\text{pre.Account.doAutos( )} \equiv (\tau = \varepsilon)\\&\text{post.Account.doAutos()} \equiv\\&\quad [\ !\{\text{nautos, autos}\} \wedge (|\tau| = \text{nautos})\\&\quad \wedge (\text{bal} = \#\text{bal} + (\Sigma(k = 0 \ldots \text{nautos} - 1).\ \text{autos}[k]))\\&\quad \wedge (\forall k : (1 \leq k \leq |\tau|) :\\&\qquad \tau[k].m \in \{\text{deposit, withdraw}\})\ ] \qquad (4)\end{aligned}$$

τ denotes the *trace* of hook method calls that doAutos() makes during its execution. At its start, doAutos() has not made any hook method calls, so τ is ε, the empty sequence. Each hook method call (and corresponding return) is recorded by appending a single element to τ. This element consists of a number of components, including the name of the method in question, the parameter values passed in the call, the returned results, etc.; for full details, we refer the reader to [15]. Here we are interested only in the identity of the method; $\tau[k].m$ gives us the identity of the method invoked in the call recorded in the k^{th} element of τ. Thus the post-condition in (4) states that when doAutos() finishes, it would have made as many hook method calls as nautos, the number of automatic transactions in the autos[] array, and that each of these calls will be to either deposit() or withdraw(). This specification can, using the *enrichment rule* of [15], then be combined with the specification (3) to arrive at the following:

$$\begin{aligned}&\text{post.NIAccount.doAutos()} \quad \equiv\\&\quad [\ !\{\text{nautos, autos}\} \wedge (|\tau| = \text{nautos})\\&\quad \wedge (\text{bal} = \#\text{bal} + (\Sigma(k = 0 \ldots \text{nautos} - 1).\ \text{autos}[k]))\\&\quad \wedge (\forall k.(1 \leq k \leq |\tau|).\ \tau[k].m \in \{\text{deposit, withdraw}\})\\&\quad \wedge (\text{tCount} = \#\text{tCount} + \text{nautos})\ ] \qquad (5)\end{aligned}$$

This asserts, as expected, that doAutos() increments the transaction count appropriately. Informally speaking, what we have done here is to 'plug-in' the additional information provided by the derived class specs (3) of the hook methods, into the specification (4) of the template method, to arrive at the enriched behavior of the template method in the derived class.

4. TESTING POLYMORPHIC BEHAVIOR

Suppose we wanted to test the class Account to ensure that it behaves as expected, i.e., according to its specifications. We could use the approach outlined in Section 2 to define the corresponding test class, TAccount shown partially in Figure 3. tAccount is the test account object. rg as an object of type Random, to be used for generating random values (for use as parameter values). t_deposit() is the test method corresponding to deposit(). We generate a random amount rd to deposit into tAccount, and if the pre-condition of deposit() (as specified in (1)) is satisfied, we invoke deposit(rd) on tAccount, and then assert that the post-condition of deposit() must be satisfied, with appropriate substitutions such as replacing bal by tAccount.bal being made. Note that we also need the save the starting values of the data members of tAccount since the post-condition refers to these values. We have shown only one of these in the figure, oldbal being the variable in which the starting balance in tAccount is saved. Of course, when the data member in question is more complex, such as the array autos[], this becomes somewhat more involved; and if the member is an object (of a type defined by the user), this will require, as we noted in Section 2, that the corresponding class provide a *cloning* operation.

```
class TAccount {
  protected Account tAccount; // test object
  Random rg;
  public void t_deposit( ) {
    int rd = rg.nextInt(); int oldbal = tAccount.bal; ...
    if( rd > 0 ) { tAccount.deposit(rd);
      assert( (tAccount.bal = oldbal+rd) ∧ ... ); }
}
```

Figure 3: Test class TAccount

The test methods t_withdraw() and t_getBal() are similarly written, and we will omit them. Let us consider the template method doAutos(). If we were only interested in the specification (2) which gives us information only about the functional effect that doAutos() has on the data members of the Account class, this too would be straightforward[6]. But a key aspect of the behavior of doAutos(), indeed the aspect that qualifies it as a template method and makes it possible to define derived classes that enrich its behavior by simply redefining deposit() and/or withdraw(), is of course the calls it makes to these hook methods. Thus if we are to really test the implementation of doAutos() against its expected behavior, the testing must be against the trace-based specification (4).

However, we face an important difficulty in doing this. The problem is that the trace variable τ which plays a key role in this specification is not an actual member variable of the Account class. We could, of course, introduce such a variable in the test class TAccount but this won't serve our purpose. The problem is that τ has to record appropriate information about the hook method calls that doAutos() makes *during* its execution; this cannot be done in the test method t_doAutos() before it calls doAutos() or after doAutos() returns. In other words, what we need to do is to 'track' doAutos() *as it executes*; whenever it it gets ready to make a hook method call, we have to 'intervene', record appropriate information about the call – in particular, the name of the method called, the parameter values, the state of the object at the time of the call – and then let the call proceed; once the hook method finishes execution and returns control to doAutos(), we again need to intervene and record information about the results returned and the (current) state of the object. One possible way to do this would be to insert the appropriate statements to update the value of τ before and after each hook method call in the body of doAutos(); but this would not only require access to the source code of doAutos(), it will require us to *modify* that source code, and

[6]One question here would be that of generating a random value in the tAccount.autos[] array; indeed, in general, the test object should be in a random (reachable) state, rather than being initialized to some 'standard' state; but this question is independent of inheritance and polymorphism, so we will ignore it here.

this is clearly undesirable.

The solution turns out to be provided by polymorphism itself. The key is to define TAccount not as a class that includes a member variable of type Account but rather to have TAccount as a derived class of Account. We call this new test class T2Account in order to distinguish it from the original test class TAccount. T2Account appears in Figure 4. The variable tau of T2Account is the trace variable in which

```
class T2Account extends Account {
  protected trace tau; // trace variable
  public void deposit(int aa) {
    // add element to tau to record info such as
    // name of method called (deposit),
    // parameter value (aa) etc., about this call;
    super.deposit(aa);
    // add info to tau about the result returned
    // and current state.
  }
  // withdraw() will be similarly defined.
  public void t_doAutos( ) {
    tau = ε;
    // check pre-condition, then call doAutos(),
    // assert post-condition.
  }
}
```

Figure 4: Test class T2Account

we record information about the sequence of hook method calls that doAutos() will make during its execution.

The t_doAutos() method starts by initializing tau to ε, then calls doAutos() (on the *self* object). Let us consider what happens when doAutos() executes, in particular when it invokes the deposit() method (withdraw() is, of course, similar, so we won't discuss it). We have redefined deposit() in T2Account, so this call in doAutos() to deposit() will be dispatched to T2Account.deposit() since the object that doAutos() is being applied to is of type T2Account. Now T2Account.deposit() is simply going to delegate the call to the Account.deposit() but before it does so, it records appropriate information, such as the name of the hook method called ('deposit'), the parameter value (aa), etc., about this call on tau. Next, T2Account.deposit() calls the deposit() defined in Account; when Account.deposit() finishes, control comes back to T2Account.deposit(); T2Account.deposit() now records additional information (about the result returned, current state of the object, etc.), and finishes, so control returns to Account.doAutos(). The net effect is that the original code, Account.deposit(), of the hook method invoked has been executed but, in addition, information about this call has been recorded on the trace. And to do this, we did not have to modify the code of any of the methods of Account, indeed we did not even need to be able to see that code.

One point might be worth stressing: T2Account.deposit() is *not* the test method corresponding to deposit(); rather, it is a redefinition of the hook method Account.deposit() in order to record information about calls that template methods might make to this hook method, the information being recorded on the trace of the template method. If there is more than one template method, we might consider introducing more than one trace variable, and yet another variable to keep track of which template method is currently being tested so that the redefined hook methods can record the information on the correct trace variable. This is in fact not necessary since only one template test method will be executing at a time, and it starts by initializing tau to ε. Of course we have assumed that we can declare tau to be of type "trace". If we really wanted to record all the information that tau has to contain in order to ensure completeness of the reasoning system [15], things would be quite complex. We can simplify matters somewhat by only recording the identities of the hook methods called and the parameter values and results returned. This is a topic for further work.

This approach can also be used for testing *abstract* classes, i.e., classes in which one or more of the hook methods may be abstract (in *Java* terminology; pure virtual in *C++*, deferred in *Eiffel*). The only change we have to make is that in T2Account.deposit(), we cannot invoke super.deposit(); instead, we would just record information in tau and return to doAutos(). Note that the specifications (2) and (4) would also be quite different. For one thing, we cannot really establish (2) because, if Account.deposit() (and, presumably, Account.withdraw() as well) is abstract, there is no way to tell what effect doAutos() will have on bal, etc. Nevertheless, the portion of (4) that refers to the hook methods invoked can still be specified since the basis for this can be seen from the body of the template method, so the designer of the Account class could have written this down as part of the specification of doAutos(). The t_doAutos() method will then test that doAutos() does indeed satisfy the expectation about the hook methods it will call[7].

Let us now consider the derived class NIAccount. How do we construct the test class TNIAccount? We cannot define it as a derived class of T2Account because then the redefinitions of the hook methods in NIAccount would not be used by the test methods in TNIAccount. In fact, in general, test classes should be *final*; i.e., a given test class TC is only intended to test that the methods of the corresponding class C meet their specs. A different class D, even if D is a derived class of C, would have to have its own test class defined for it. Of course, TNIAccount would be quite similar to T2Account. The important differences would be that we would have test methods corresponding to any new methods defined in NIAccount, and pre- and post-conditions would be the ones from the specifications (such as (3) and (5)) of this class.

Before concluding this section, we should note one other point. An important assumption we have made is that hook methods obey behavioral subtyping [10], i.e., any redefinitions of hook methods in the derived class must continue to satisfy their base class specifications. If this were not the case, the reasoning that we have performed in the base class about the behavior of the template method, including the trace-based specification of that method, may no longer be valid. For example, suppose a template method $t()$ first calls the hook method $h1()$; if the value returned by $h1()$ is positive, $t()$ then calls $h2()$, else it calls $h3()$. Suppose

[7]In fact, we would not only want to be assured about the identity of the hook methods called or the number of times they are called (which are the pieces of information provided by (4)) but also the parameter values passed in these calls as well as the state just before the calls, etc.; this is particularly important if the hook method in question is abstract. This additional information can be provided using our traces although the resulting specs are naturally much more involved [15].

also that the base class specification of $h1()$ asserts that it will return a positive value. When reasoning about the base class, we might then establish, on the basis of this specification of $h1()$, a specification for $t()$ which asserts that the identity of the first hook method that $t()$ calls (as recorded in the first element of the trace τ of $t()$) is $h1()$, and the identity of the second method called is $h2()$. Suppose now we redefine $h1()$ in the derived class so that it returns a negative value. Then, in the derived class, $t()$ will not satisfy its specification, and the problem is not with $t()$ but with the way that $h1()$ was redefined. The redefined $h1()$ does not satisfy its base class specification, i.e., it violates behavioral subtyping. Hence, when testing the behavior of the hook methods in the derived class, it may be useful not just to test against the derived class specification of the method, but also against its base class specification to ensure that the redefined hook method still satisfies that specification.

5. DISCUSSION

Let us briefly consider a class C that has a member variable acc of type Account. In reasoning about the behavior of the methods of C, we will of course depend upon the specifications of the Account class. Do we have to worry about the specifications of the NIAccount class? Yes, indeed. The point is that for a particular object that is an instance of C, the acc component may well be of type NIAccount[8]. In fact, one reason for defining classes such as NIAccount is precisely that client classes such as C can take advantage of the enrichment provided by this class. What are the issues that we have to consider in reasoning about and testing the behavior of C?

One possibility would be that in reasoning about C, we only take account of the specification of Account. And in testing C, we only create instances of C that have an acc component of type Account. But this is clearly insufficient. We need to test the behavior of C for instances that have an acc component of type NIAccount. In fact, whenever a new derived class of a base class such as Account is defined, the behavior of any client code of Account has to be re-tested [13]. While this may seem undesirable, it is to be expected. After all, by defining a new derived class of Account, we are enriching the behaviors that a client class, such as C, of Account can exhibit; so naturally we have to test for such richer behaviors. The techniques for reasoning about such richer behaviors of C, as well as the corresponding techniques for testing them, are topics for further work.

[8]Of course, in languages like $C++$ for this to happen, acc would have to be a pointer to Account but this is a language detail which we can ignore.

6. REFERENCES

[1] K. Arnold, J. Gosling, and D. Holmes. *Java Programming Language, Third Edition*, 2000.

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

[3] M. Buchi and W. Weck. The greybox approach: when blackbox specifications hide too much. Turku Centre for Computer Science TR No. 297, 1999, http://www.tucs.abo.fi/.

[4] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proc. of 18th Int. Conf. on Softw. Eng.*, pages 258–267. IEEE Computer Soc., 1996.

[5] S. Edwards, G. Shakir, M. Sitaraman, B. Weide, and J. Hollingsworth. A framework for detecting interface violations. In *Proc. of 5th Int. Conf. on Softw. Reuse*. IEEE, 1998.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable OO software*. Addison-Wesley, 1995.

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

[8] P. Krishnamurthy and P. Sivilotti. The specification and testing of quantified progress properties in distributed systems. In *23rd Int. Conf. of Software Eng.* ACM, 2001.

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

[10] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. and Systems*, 16:1811–1841, 1994.

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

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

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

[14] C. Ruby and G. Leavens. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000*, pages 208–228. ACM, 2000.

[15] N. Soundarajan and S. Fridella. Framework-based applications: Incremental development to incremental reasoning. *Proc. of 6th Int. Conf. on Softw. Reuse*, pp. 100–116, Springer, 2000.

[16] N. Soundarajan and B. Tyler. Specification-based incremental testing of object-oriented systems. In *TOOLS 39*, pp. 35–44, IEEE CS Press, 2001.