JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 1, No. 3
Special issue: TOOLS USA 2002 proceedings

Testing Polymorphic Behavior

Neelam Soundarajan and Benjamin Tyler
Computer and Information Science, Ohio State University, Columbus, OH 43210

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 with the *polymorphic* behavior of classes. 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. In other words, we want to use a *black-box* approach [1] to testing. 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; the software vendor will typically not provide source code for these components, hence a black-box approach is necessary in such a situation. Second, the testing approach should enable us to suitably deal with classes that exploit *polymorphism*[1]. This is important because, as is widely recognized, see for example [10], the OO approach derives much of its power from the mechanism of polymorphism. Therefore it is clearly important in a specification-based testing approach, to include testing of polymorphic behavior.

[1]Throughout this paper by 'polymorphism', we mean *inclusion polymorphism* [2].

Unfortunately, however, there is a potential for conflict between these two requirements. Suppose $p()$ is a polymorphic method of C. As we will see in the next section, specification of $p()$'s polymorphic behavior will have to include information about the methods that $p()$ invokes during its execution. Testing to see if such a specification is satisfied cannot, it would seem, be done by waiting until $p()$ finishes execution and then checking the state of the object. Rather, we would have to be concerned with the details of the body of $p()$; indeed, as we will see, the obvious approach would seem to be to *add* some additional code to $p()$'s body to record the information that $p()$'s specification refers to. But this is clearly incompatible with the black-box approach. Remarkably, as we will see later in the paper, polymorphism itself provides a satisfactory solution to the problem. We will show how to construct a test class *Test_C* that exploits polymorphism to 'intercept' the execution of $p()$ at key moments and record the needed information about the behavior of $p()$, without having any access to the body of $p()$ as defined in C.

A particularly compelling example of the need for testing against specifications that characterize the sequence of method calls that polymorphic methods make during their execution is provided by OO *frameworks*. A framework F typically consists of a number of key polymorphic methods that invoke appropriate other methods at key points. A complete application can be built by suitably redefining, in derived classes of some of F's classes, the methods invoked by the polymorphic methods. If F has been well-designed, a whole range of applications can be built with just the effort of redefining these methods in the derived classes, while reusing all the polymorphic methods defined in F. Clearly, the specification of F must provide information about the sequence of method calls that its polymorphic methods make since this (in conjunction with the behavior of the methods redefined in the derived classes) is what will determine the behavior that the application will exhibit. Therefore, testing F requires us to test against such specifications of its polymorphic methods. We will return to this in the final section of the paper.

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

- It identifies a key problem in black-box testing of polymorphic behavior.
- It develops a solution to the problem that exploits polymorphism in such a way that it enables us to test that a given method exhibits the (polymorphic) behavior required by its specification, without needing any access to the body of the method under test.
- It illustrates the approach by applying it to a simple example.

The rest of the paper is organized as follows. In Section 2 we consider what type of information has to be included in specifications of polymorphic behavior and the problem we face in black-box testing of such behavior. In Section 3 we use a simple example to show how polymorphic behavior can be formally specified. In Section 4 we present our approach to testing such specifications without accessing or modifying the body of the method under test, and illustrate the solution on the example from the previous section. In Section 5 we briefly discuss related work on testing of OO systems. In the final section, we summarize our approach to testing polymorphic behavior and consider future work.

2 BACKGROUND AND MOTIVATION

Suppose C is a class that consists of a polymorphic method $p()$ that invokes two other methods $h1()$ and $h2()$ of C. Suppose also that in a derived class D of C, we redefine $h1()$ and $h2()$ (and inherit $p()$ from the base class). If $p()$ is applied to an object that is an instance of D, polymorphism requires, and the run-time dispatch system used to implement polymorphism ensures, that the $h1()$ and $h2()$ that are invoked during this execution of $p()$ are the ones defined in D rather than the ones defined in the base class. Thus the resulting behavior of $p()$ will be modified –we will use the term 'enriched'– because of the richer behavior implemented in $D.h1()$ and $D.h2()$ although $p()$ itself was inherited without change from the base class; this, of course, is exactly what makes polymorphism such a powerful tool.

The precise enrichment that $p()$'s behavior acquires as a result of the redefinitions of $h1()$ and $h2()$ will depend critically on the particular methods $p()$ invokes, the number of times it invokes each, the order it performs these invocations in, the argument values it passes in each invocation, etc. It is for this reason that polymorphic methods such as $p()$ are called *template methods* and the methods such as $h1()$ that it invokes are called *hook methods* in the *design patterns* literature [5]; the template method provides the 'template' of invocations of the hook methods, and the hook methods provide 'hooks' that the derived class designer can use to implement behavior appropriate to his or her derived class with the result that the template method's behavior is also appropriately enriched. From the point of view of specification, this means that a specification of $C.p()$ cannot just give us information on what the values of C's member variables will be when the method finishes execution. Rather, it must include suitable information about the calls $p()$ makes to the hook methods. Given such a specification, we will be able, when reasoning about the derived class, to 'plug-in' the richer behavior of $D.h1()$ and $D.h2()$ into the base class specification of $p()$ to arrive at the enriched behavior that $p()$ will exhibit in D, i.e., when it is applied to objects of type D.

In the next section we will see some details of such a specification of $p()$. In our specification, we will use a *trace* variable corresponding to each template method, as *auxiliary variables* [13]; the hook method calls that the template method $p()$ makes will be represented by recording suitable information about these calls on the trace variable, which we will generally denote by the symbol τ, of $p()$. As a result, the post-condition of $p()$ will include conditions that τ must satisfy in the same manner as it imposes conditions that the other variables, in particular the member variables of C, must satisfy when $p()$ finishes execution. Such a specification, as we will see, will enable us to arrive at the derived class behavior of $p()$, resulting from the behaviors of the hook methods as defined in the derived class, without having to reconsider the details of the code body of $p()$.

Now consider how we can *test* whether the methods of a class C satisfy their specifications. Since we want to use black-box testing, a natural approach [17] would be to introduce a *test class Test_C* with a test method *test_m*() corresponding to each method $m()$ of C. *Test_C* would contain an object tc of type C that would serve as the 'test object'. The individual test methods, *test_m*() would work as follows: First check that the state of tc satisfies the pre-condition of $m()$; if the pre-condition is not satisfied, we do not continue with the test since a basic tenet of the *design-by-contract* principle is that nothing is guaranteed about $m()$'s behavior if its pre-condition is not satisfied at the time of its invocation so there is nothing to test in this case[2]; next invoke the method $m()$ on tc; finally, when $m()$ returns, check whether the current state of tc (and any other result that $m()$ returns) satisfy the post-condition of $m()$; if the post-condition is not satisfied, then we have found an instance where the method does not meet its specification.

If the method in question is a polymorphic method $p()$, its post-condition, as we saw, will include conditions on the trace variable τ associated with that method. But there is nothing in the actual code body of $p()$ that will add the needed information to τ when $p()$ makes its hook method calls since τ is not an actual variable that the code of $p()$ manipulates in any way. Rather it is an auxiliary variable introduced *by the specification and reasoning system*. So how do we check if $p()$ meets such a specification? One possible solution would be introduce τ as a new variable in *Test_C*. But this, by itself, is not sufficient. We also need to ensure that τ is updated appropriately each time $p()$ makes a hook method call to add information, to τ, about this particular call; as we will see, information has to be added to τ also when the hook method call finishes and returns to $p()$. This clearly requires additional code but where would we put this code? Putting this additional code in *test_p*() will not work since it is $p()$ that makes these calls, so control will be in $p()$ immediately before and after each of these calls. In fact, according to the structure of the test methods we saw above, control does not return to *test_p*() until $p()$ *completes* execution. By that point, details about which hook methods $p()$ called during its entire execution, the argument values it passed in those calls, etc., are no longer available. For example, suppose in $p()$ there was a loop and there was a hook method call inside that loop; how can we tell, once $p()$ finishes, how many times it executed the loop, i.e., how many times it called the hook method? The clear alternative would seem to be to put the code for updating τ in $p()$, immediately before (and immediately after) the hook method calls, but this would violate our basic requirement of black-box testing. In black-box testing we do not even have access to the code of the methods under test, let alone trying to insert new code into them.

This is the fundamental conflict between our two requirements: On the one hand, the requirement that we be able to test polymorphic behavior means that we need

[2]In a complete testing methodology, we would have to find ways to perhaps modify the state of tc so that it satisfies the pre-condition of $m()$, rather than simply abandoning the test.

to be able to record, on the trace variable, information about each hook method call that the polymorphic method makes as it executes which seems to require us to insert code for this purpose into the body of the polymorphic method under test. On the other hand, the requirement of black-box testing forbids any access to the method under test. It turns out, as we will see in Section 4, that polymorphism itself comes to the rescue. We will see how we can use polymorphism in the test class *Test_C* to ensure that each time $p()$ (as defined in C) makes a hook method call, the call is intercepted, and appropriate information about the call is added to the trace and then the call proceeds normally; similarly, when the call finishes, the return will be intercepted, appropriate information about the results returned are added to the trace, and then control returns to $p()$; and none of this will require any change in (or even access to) the code of $C.p()$. This will allow us to test the polymorphic behavior of $p()$ while staying within the confines of black-box testing.

3 SPECIFYING POLYMORPHIC BEHAVIOR

Let us consider a simple example starting with the base class Eater. Objects of this class live fairly simple lives eating donuts and hamburgers, increasing their caloric intake in the process. The definition of the Eater class (using a *C++/C#*-like syntax) appears in Figure 1.

```
class Eater {
  protected int Cals_Eaten = 0;
  public virtual void Eat_Donuts(int n)
    { Cals_Eaten = Cals_Eaten + 200 * n; }
  public virtual void Eat_Burgers(int n)
    { Cals_Eaten = Cals_Eaten + 400 * n; }
  public void Eat_1200_Cals()
    { Eat_Donuts(2);  Eat_Burgers(2); }
}
```

Figure 1: Base class Eater

As their names suggest, the member variable `Cals_Eaten` maintains the number of calories consumed by the Eater object, while `Eat_Donuts()` and `Eat_Burgers()` are methods that increase it. These methods' specifications[3] are easily given:

$$\begin{aligned}\mathsf{pre.Eater.Eat\_Donuts(n)} &\equiv (\mathrm{n} > 0)\\ \mathsf{post.Eater.Eat\_Donuts(n)} &\equiv (\mathsf{Cals\_Eaten} = \mathsf{Cals\_Eaten@pre} + 200 * \mathrm{n})\end{aligned}$$

[3]Throughout this paper, we will use *concrete* specifications, i.e., specifications in terms of the member variables of the class in question. When dealing with more complex classes, we would of course have to introduce a suitable conceptual model of the class, provide a mapping from the concrete to conceptual model, and express the specifications in terms of the conceptual model [9]. 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 `Eater` class.

$$\begin{aligned} \mathsf{pre.Eater.Eat\_Burgers(n)} &\equiv (\mathrm{n} > 0) \\ \mathsf{post.Eater.Eat\_Burgers(n)} &\equiv (\mathsf{Cals\_Eaten} = \mathsf{Cals\_Eaten@pre} + 400 * \mathrm{n}) \end{aligned} \qquad (1)$$

In the post-conditions we use the *OCL*-notation [18] "@pre" to refer to the value of the variable at the start of the execution of the method.

Eat_1200_Cals() is a polymorphic method and will be the one of most interest to us. It does its job by calling the hook methods Eat_Donuts() and Eat_Burgers() (rather than by direct manipulation of Cals_Eaten). One possible specification for this method would be as follows:

$$\begin{aligned} \mathsf{pre.Eater.Eat\_1200\_Cals()} &\equiv \mathit{true} \\ \mathsf{post.Eater.Eat\_1200\_Cals()} &\equiv (\mathsf{Cals\_Eaten} = \mathsf{Cals\_Eaten@pre} + 1200) \end{aligned} \qquad (2)$$

What is missing from (2) 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 for reasoning about the enriched behavior that this method will exhibit in the derived class, to which we turn next.

```
class Eater_Jogger :  Eater {
  protected int Cals_Burned = 0;
  public void Jog() { Cals_Burned = Cals_Burned + 100; }
  public override void Eat_Donuts(int n)
    { Cals_Eaten = Cals_Eaten + 200 * n; Cals_Burned = Cals_Burned + 5 * n;}
  public override void Eat_Burgers(int n)
    { Cals_Eaten = Cals_Eaten + 400 * n; Cals_Burned = Cals_Burned + 5 * n;}
}
```

Figure 2: Derived class Eater_Jogger

Objects of the derived class Eater_Jogger in Figure 2 exhibit slightly healthier lifestyles than those of the base class Eater due to the enrichments provided by the additional data member Cals_Burned, the entirely new method Jog(), and the slight redefinitions of Eat_Donuts() and Eat_Burgers(). As we would expect, Cals_Burned keeps track of the calories expended through the act of jogging as well as eating hamburgers and donuts. Of course, we still take in calories through the act of eating, so we must increase Cals_Eaten when calling Eat_Donuts() and Eat_Burgers() as we did in the base class[4]. The specifications for the methods redefined in Eater_Jogger and for its new method are straightforward:

$$\begin{aligned} \mathsf{pre.Eater\_Jogger.Jog()} &\equiv \mathit{true} \\ \mathsf{post.Eater\_Jogger.Jog()} &\equiv (\mathsf{Cals\_Eaten} = \mathsf{Cals\_Eaten@pre} \wedge \\ &\quad\ \mathsf{Cals\_Burned} = \mathsf{Cals\_Burned@pre} + 100) \\ \mathsf{pre.Eater\_Jogger.Eat\_Donuts(n)} &\equiv (\mathsf{n} > 0) \\ \mathsf{post.Eater\_Jogger.Eat\_Donuts(n)} &\equiv (\mathsf{Cals\_Eaten} = \mathsf{Cals\_Eaten@pre} + 200 * \mathsf{n} \wedge \\ &\quad\ \mathsf{Cals\_Burned} = \mathsf{Cals\_Burned@pre} + 5 * \mathsf{n}) \end{aligned}$$

[4]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 Cals_Eaten, so we have just repeated the code.

$$\begin{aligned}\mathsf{pre.Eater\_Jogger.Eat\_Burgers(n)} &\equiv (\mathsf{n} > 0)\\ \mathsf{post.Eater\_Jogger.Eat\_Burgers(n)} &\equiv (\mathsf{Cals\_Eaten} = \mathsf{Cals\_Eaten@pre} + 400{*}\mathsf{n} \wedge\\ &\qquad \mathsf{Cals\_Burned} = \mathsf{Cals\_Burned@pre} + 5{*}\mathsf{n}) \qquad (3)\end{aligned}$$

Let us now turn to the behavior of the template method `Eater_Jogger.Eat_1200_Cals()` inherited from the `Eater` class. It is clear from the body of this method, as defined in Figure 1, that during its execution, the value of `Cals_Burned` will increase by 20 since the calls to `Eat_Burgers()` and `Eat_Donuts()` will each be dispatched to the methods defined in the `Eater_Jogger` class, and each of these calls will increment `Cals_Burned` by 10. But we cannot arrive at this conclusion just given the specifications in (2); in particular, that there is no way to appeal to the specifications in (3) for the behavior of the redefined hook methods that `Eat_1200_Cals()` invokes. The problem is that there is nothing in (2) that tells us that `Eat_1200_Cals()`, in fact, invokes `Eat_Donuts()` or `Eat_Burgers()`. Indeed, if we rewrote the body of `Eat_1200_Cals()` (in the base class) so that it ate 6 donuts, i.e., invoked only `Eat_Donuts()` with 6 as the argument, or ate 4 donuts and a hamburger, or ate 3 hamburgers, or just directly incremented `Cals_Eaten` by 1200, it would still satisfy the specification (2). But for each of these different implementations, the final value we will have in `Cals_Burned` when `Eater_Jogger.Eat_1200_Cals()` finishes, will be different (and in none of these cases would it be equal to ($\mathsf{Cals\_Burned@pre} + 20$)).

Consider the following more informative specification:

$$\begin{aligned}\mathsf{pre.Eater.Eat\_1200\_Cals()} &\equiv (\tau = \varepsilon)\\ \mathsf{post.Eater.Eat\_1200\_Cals()} &\equiv [\,(\mathsf{Cals\_Eaten} = \mathsf{Cals\_Eaten@pre} + 1200) \wedge\\ &\quad (|\tau| = 2) \wedge (\tau[1].m = \text{“}\mathsf{Eat\_Donuts}\text{”}) \wedge (\tau[1].p = 2) \wedge\\ &\quad (\tau[2].m = \text{“}\mathsf{Eat\_Burgers}\text{”}) \wedge (\tau[2].p = 2)\,] \qquad (4)\end{aligned}$$

τ denotes the *trace* of hook method calls that `Eat_1200_Cals()` makes during its execution. At the start of `Eat_1200_Cals()`, τ is ε, the empty sequence, since `Eat_1200_Cals()` has not called any hook methods yet. 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 and object values immediately before the call, the parameter and object values after the call, etc.; for full details, we refer the reader to [16]. Here we are interested only in the identity of the hook method and the initial value of the parameter value for each call. Thus the post-condition in (4), in particular the clause ($|\tau| = 2$) where $|\tau|$ is the length, i.e., the number of elements, of τ, tells us that when `Eat_1200_Cals()` finishes, it would have made exactly two hook method calls. The next clause tells us that the identity of the method called in the first of these two calls is “Eat_Donuts”, and the clause following that tells us that the parameter value passed in this call is 2. Similarly the next two clauses tell us that the next method called is “Eat_Burgers” and the parameter value passed in this call is also 2.

This specification can then be combined, using the *enrichment rule* of [16], with the specification in (3) of `Eater_Jogger.Eat_Donuts()` and

Eater_Jogger.Eat_Burgers(), to arrive at the following:

$$\begin{aligned}\mathsf{pre.Eater\_Jogger.Eat\_1200\_Cals()} &\equiv \mathit{true}\\ \mathsf{post.Eater\_Jogger.Eat\_1200\_Cals()} &\equiv [\,(\mathsf{Cals\_Eaten} = \mathsf{Cals\_Eaten@pre} + 1200) \wedge\\ &\qquad (\mathsf{Cals\_Burned} = \mathsf{Cals\_Burned@pre} + 20)\,]\end{aligned} \quad (5)$$

This asserts, as expected, that Eater_Jogger.Eat_1200_Cals() increases Cals_Eaten by 1200 and Cals_Burned by 20. Informally speaking, what we have done here is '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. Since our focus here is on testing, we will not go into the formal details of the rule corresponding to this 'plugging-in' process, referring the interested reader to [16]. But even without these formal details, it should be clear that the information provided in (4) is essential if we want to be able to understand the enriched behavior that Eater_Jogger.Eat_1200_Cals() acquires as a result of the enriched behavior implemented in the hook methods Eater_Jogger.Eat_Donuts() and Eater_Jogger.Eat_Burgers(), that it invokes. This means that if we are to satisfactorily test the behavior of the body of Eat_1200_Cals() as defined in Figure 1, it must be against a specification such as (4), rather than the simpler specification such as (2) which does not include information about the hook methods that this template method calls. In the next section we will see how to define a test class Test_Eater that will allow us to do this.

4 TESTING THE SPEC

Let us now see how we can build a test class, Test_Eater to test if the methods of Eater behave according to their specifications. A first attempt at Test_Eater, consisting only of a method to test the behavior of Eat_Donuts(), appears in Figure 3. t_Eater is the test object, rg is an object of type Random which is to be used

```
class Test_Eater_try1 {
  private Eater t_Eater; // test object
  private Random rg = new Random();
  public void test_Eat_Donuts() {
    int n = rg.Next(99);
    if( n > 0 ) {
      int old_n = n;
      int old_Cals_Eaten = t_Eater.Cals_Eaten; // problem!
      t_Eater.Eat_Donuts(n);
      assert(t_Eater.Cals_Eaten == old_Cals_Eaten + 200*old_n); }
  }
}
```

Figure 3: Test class Test_Eater_try1

for generating random parameter values for the tests, and test_Eat_Donuts() the test method corresponding to Eat_Donuts(). We generate a random integer n to represent the number of donuts t_Eater is to consume, and if the pre-condition of Eat_Donuts() (as specified in (1)) is satisfied, we invoke Eat_Donuts() on t_Eater. However, before we invoke the method, we must first save the state (i.e., the values of n and t_Eater.Cals_Eaten) so it can be used when asserting the post-condition after the call has completed. Saving t_Eater.Cals_Eaten is a bit of a problem since Cals_Eaten is a protected member of the Eater class (Fig. 1) and so is not directly accessible in Test_Eater. Let us ignore this problem for now. Once Eat_Donuts() finishes execution, its post-condition must be satisfied, using appropriate substitutions such as replacing Cals_Eaten with t_Eater.Cals_Eaten. Also, we replace occurrences of "@pre" by references to the starting values of the corresponding variables, which we saved in the "old" variables[5]. The assert that appears following the call to Eat_Donuts() requires the method's post-condition, with the substitutions just described, to be satisfied. If it is not, we will get an appropriate error message alerting us to the problem, and we have found a mistake in the Eat_Donuts() method.

The test method for test_Eat_Burgers() is similarly written, so will we omit it. Consider next the template method Eat_1200_Cals(). If we were only interested in the specification (2) which gives us information only about the effect that Eater.Eat_1200_Cals() has on the data members of the Eater class, this too would be straightforward. But as the analysis in the last two sections showed, this is insufficient; we also need to test that Eater.Eat_1200_Cals() meets its polymorphic behavior as specified in (4).

However, as we saw in Section 2, there are some key problems in doing this. First, the trace τ is not a member variable of the Eater class. This can be addressed by simply introducing it as a member of the test class. The more serious problem is that τ has to record appropriate information about the hook method calls that Eat_1200_Cals() makes *during* its execution; this cannot be done in the test method test_Eat_1200_Cals() before it calls Eat_1200_Cals() because at this point we have no way of knowing which particular hook methods Eat_1200_Cals() will call during its execution, nor can we do it after after Eat_1200_Cals() returns because again at this point we have no way of knowing which particular hook methods Eat_1200_Cals() did call during its execution. Instead, we need to 'track' Eat_1200_Cals() *as it executes*; whenever 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, we again need to intervene and record information about the results returned and the (current) state of the object before allowing the rest of Eat_1200_Cals() to continue. One way to do this would be to insert the appropriate statements

[5]If the parameters and data members in question are more complex or are references to objects, the task of saving the 'old values' is much more involved; in general we will need suitable *clone* [10] operations for this purpose. We will not concern ourselves with this issue here since this is orthogonal to the polymorphism question.

to update the value of τ before and after each hook method call in the body of Eater.Eat_1200_Cals(); but this would not only require access to the source code of that method, it will require us to *modify* that source code, and this is clearly incompatible with black-box testing.

Our solution is to define the testing class as a *derived class* of Eater and exploit polymorphism in it. This test class, Test_Eater, appears in Figure 4. Test_Eater.tau is the trace variable in which we will record information about the

```
class Test_Eater :  Eater {
  protected trace tau; // trace variable
  public override void Eat_Donuts(int n) {
    // add element to tau to record info such as name of method
    // called (Eat_Donuts), parameter value (n) etc., about this call;
    base.Eat_Donuts(n);
    // add info to tau about the result returned and current state.
  }
  // Eat_Burgers() will be similarly defined.
  public void test_Eat_1200_Cals() {
    if (true) {
      // Ok, since Test_Eater extends Eater
      int old_Cals_Eaten = this.Cals_Eaten;
      tau = ε;
      this.Eat_1200_Cals();
      // assert trace-based post-condition
    }; }
}
```

Figure 4: The improved test class Test_Eater

sequence of hook method calls that Eat_1200_Cals() makes during its execution.

Let us see how Test_Eater.test_Eat_1200_Cals() as defined in Fig. 4 works by using the *sequence call diagram* in Fig. 5 to explain how Test_Eater.test_Eat_1200_Cals() functions. The six vertical lines in the figure, each labeled at the top with the name of a method (the three on the left being from the Test_Eater class, the three on the right from the Eater class), represent time-lines for the respective methods. The initial call to the method is represented by the solid arrow at the top-left of the figure. The method starts by checking –this is represented by the point labeled with a diamond with a single question mark inside it– the pre-condition (which is just *true* in this case). Next it initializes tau to ε and saves the initial state in the old variable; this point is labeled (1) in the figure. Next, it calls Eat_1200_Cals() (on the this object). Since Eat_1200_Cals() is not overridden in Test_Eater, this is a call to Eater.Eat_1200_Cals() (Fig. 1). This call is represented by the solid arrow from the time-line corresponding to Test_Eater.test_Eat_1200_Cals() to the line corresponding to Eater.Eat_1200_Cals().

Consider what happens when this method executes. First, it invokes

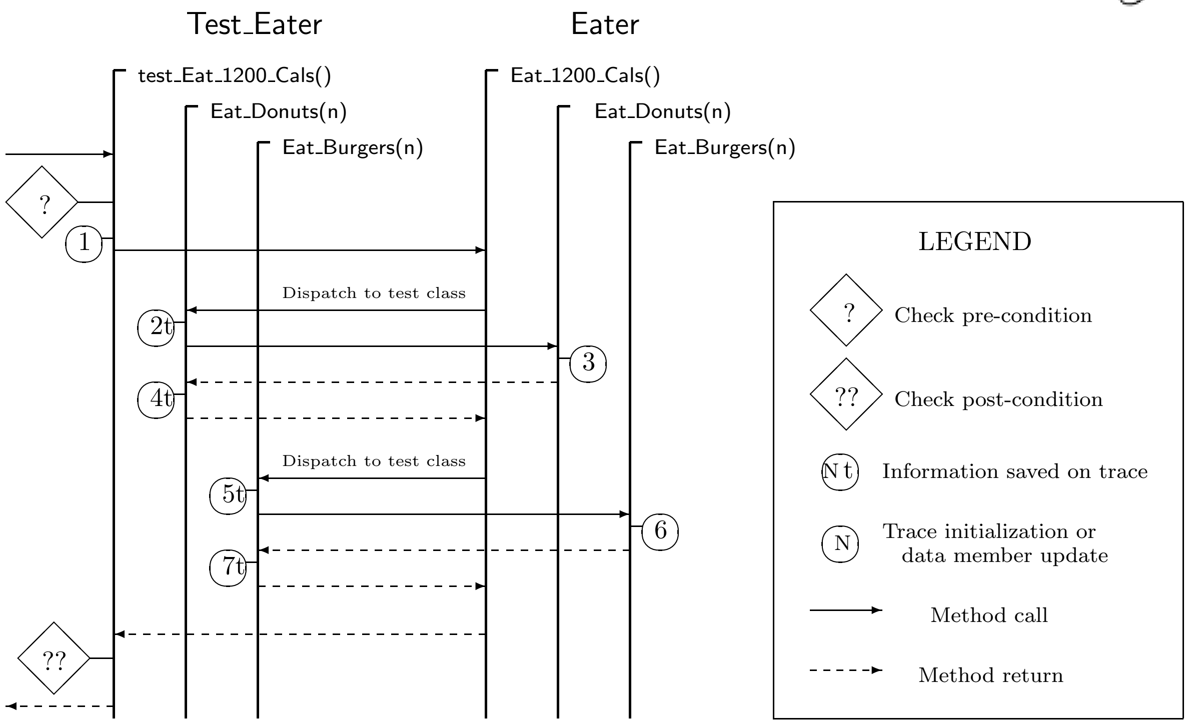

Figure 5: Sequence Call Diagram for Test_Eater.Eat_1200_Cals()

Eat_Donuts() which we *have* overridden in Test_Eater, so this call will be dispatched to Test_Eater.Eat_Donuts() since the object that Eat_1200_Cals() is being applied to is of type Test_Eater. This dispatch is represented by the solid arrow from the time-line for Eat_1200_Cals() to that for Test_Eater.Eat_Donuts() Now Test_Eater.Eat_Donuts() is simply going to delegate the call to Eater.Eat_Donuts() (represented by the arrow from Test_Eater.Eat_Donuts() to Eater.Eat_Donuts()); but before it does so it records appropriate information about this call, such as the name of the hook method called (‘Eat_Donuts’), the parameter value (n), etc., on the trace variable tau; this action is labeled by (2t) in the figure. Once Eater.Eat_Donuts() finishes (after performing its action consisting of updating Eater.Cals_Eaten, represented by the point labeled (3)), control returns to Test_Eater.Eat_Donuts(), represented by the dotted arrow from Eater.Eat_Donuts() to Test_Eater.Eat_Donuts(). Test_Eater.Eat_Donuts() now records additional information (about the result returned, current state of the object, etc.) on tau (represented by the point labeled (4t)), and finishes, so control returns to Eater.Eat_1200_Cals(); the return is indicated by the dotted arrow from Test_Eater.Eat_Donuts() to Eater.Eat_1200_Cals(). That method next calls Eat_Burgers() and this call is again dispatched to Test_Eater.Eat_Burgers(), represented by the solid arrow from Eater.Eat_1200_Cals() to Test_Eater.Eat_Burgers().

The process of saving initial information, delegating the call to the corresponding method in Eater, and then saving the results in the trace is repeated; these are represented respectively by the point labeled (5t), the solid arrow from `Test_Eater.Eat_Burgers()` to `Eater.Eat_Burgers()`, and the point labeled (7t) (after `Eater.Eat_Burgers()` performs its update –represented by (6)– and returns –labeled by the dotted arrow from `Eater.Eat_Burgers()` to `Test_Eater.Eat_Burgers()`). At this point `Test_Eater.Eat_Burgers()` finishes, so it returns –represented by the dotted arrow– to `Eater.Eat_1200_Cals()`. That method is also done now so it returns to `Test_Eater.test_Eat_1200_Cals()`. The final action, the one that we have been building up towards, is to check if the post-condition specified in (4) (with the appropriate substitutions, i.e., `tau` for τ and `old_Cals_Eaten` for `Cals_Eaten@pre`) is satisfied. This is represented by the point labeled with the diamond with a double question mark in it.

The key point is that by defining `Test_Eater` as a derived class of `Eater`, the class under test, we are able to exploit polymorphism to intercept the calls to (and returns from) the hook methods. This allows us to record information about these calls (and returns) without having to make any changes to the methods being tested. This allows us to achieve our goal of black-box testing of the polymorphic behavior of the template method of `Eater`.

It is worth noting that `Test_Eater.Eat_Donuts()` is not the test method corresponding to `Eater.Eat_Donuts()`. That test method will be essentially like the one defined in Fig. 3 with some minor changes: Both occurrences of `t_Eater.Cals_Eaten` should be replaced by `Cals_Eaten` since there is no separate `t_Eater` object now. More important, the call `t_Eater.Eat_Donuts()` should be replaced by `base.Eat_Donuts()` this being the method under test here.

If there is more than one template method, we could introduce more than one 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 [16], things would be more complex. We can simplify matters considerably by only recording the identities of the hook methods called and the parameter values and results returned. This is a topic for further work.

Let us now consider the derived class `Eater_Jogger`. How do we construct the test class `Test_Eater_Jogger`? It should *not* be a derived class of `Test_Eater` because then the redefinitions of the hook methods in `Eater_Jogger` would not be used by the test methods in `Test_Eater_Jogger`. In general, in fact, test classes should be *sealed* in C#-terminology or *final* in Java-terminology. A given test class *Test_C* 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, `Test_Eater_Jogger` would be quite similar to `Test_Eater`. The only differences would be that its base class would be `Eater_Jogger` not `Eater`, and the pre- and post-conditions would be the ones from the specifications of the `Eater_Jogger` class. This suggests that much of the work of defining test classes can be mechanized. We will return to this point in the final section.

Before concluding this section, we should note one other point. An important requirement that derived classes should satisfy is that of behavioral subtyping [9]. Behavioral subtyping requires that 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 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, we should test not just against the derived class specification of the methods, but also against their base class specifications. Alternately, the behavioral requirements specified in the base class for a hook method should be included as part of its derived class specification as is done, for example, in the Eiffel reasoning system [10].

5 RELATED WORK

Perry and Kaiser [14] and Smith and Robson [15] 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 from the base class, this code may not work in cooperation with the methods (re-)defined in the derived class; suitable tests appropriate to the derived class will have to be designed to check that this is indeed so. Harrold *et al.* [7] propose an algorithm that identifies which tests of the base class are applicable to the derived class. Their point is that if $m()$ 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 $m()$ as part of the base class, can continue to be used in the derived class. Hsia *et al.*'s [8] work is similar but deals not just with base and derived classes but also clients of classes; they use the notion of *class firewall* [19] to identify the parts of a system that are affected by a change in a given class.

None of these authors deal with specification-based testing. They are more concerned with identifying suitable test cases based on the internal structure of the classes. Doong and Frankl [3] present a specification based approach to testing

OO programs. One important difference with our work is that whereas we use concrete specifications, Doong and Frankl use abstract (algebraic) specifications. They require the user to supply an equivalence testing algorithm that can be used to test if two states are, from an abstract point of view, equivalent. We do not need such an algorithm since we can simply check whether a given concrete state satisfies a given pre- or post-condition, rather than having to compare it with an abstract state that satisfies the abstract specification.

More important, neither Doong and Frankl, nor the other authors, address the question of testing the polymorphic behavior of template methods. Indeed, as far as we know, our work is the first one to explicitly deal with this question.

6 DISCUSSION

Much of the power of the object-oriented approach derives from the mechanism of polymorphism. It allows a derived class designer to enrich the behavior of a template method by simply redefining some of the hook methods it invokes. Therefore, it is essential when testing the behavior of OO classes, not just to test their functional behavior but also the polymorphic behavior, reflected in the patterns of calls that the template methods of the class make to the hook methods. This has been the main motivation behind our work. As we noted, testing this behavior, while remaining within the confines of black-box testing is challenging since the behavior in question is, in a sense, *transient*. It manifests only during the execution of the template method and, by its very definition, black-box testing forbids 'peeking-inside' the method to observe it as it executes.

As we saw, polymorphism itself provided the solution. By defining the test class as a derived class of the class C under test and by redefining, in the test class, the hook methods of C so that they record appropriate information about the state of the test object as well as the call before delegating the call to the corresponding methods in C and similarly recording information once that delegated call finishes and returns, we were able to get the necessary information to test whether the template method's pattern of calls to the hook methods does indeed meet the expectations expressed by its specification for its polymorphic behavior. This means that we do not need any access to the source code of C; indeed, we may not even *have* that source code. This is important since if C was purchased from a software vendor, the vendor is unlikely to have supplied, given proprietary consideration, its source code.

We will conclude with a couple of pointers to future work. Consider again the case of a class whose source code we do not have. An important recent development [6] in the C#/.NET platform is the use of *meta-data* to capture important information about a class without the need for access to the source code. Meyer [11] mentions a *contract wizard* that exploits this information to allow us to add class invariants and pre- and post-conditions to existing classes. But the types of assertions considered here are in terms of just the member variables of the class in question. As we saw, to capture polymorphic behavior, we also need to be able to use traces. Indeed, polymorphic behavior is often documented informally, along the lines of "template method $t()$ will, during its execution, invoke hook method $h1()$ and $h2()$ in that order if during its earlier call to $h3()$, it received a particular response, else it will call only $h2()$". Since developers will often enrich the behaviors of such methods by defining derived classes in which methods such as $h3()$ are redefined. Hence it is important to formalize such documentations, and it would be interesting to extend the *contract wizard* to accommodate specifications involving traces. Once that is done, the next step would be to mechanize, as far as possible, the generation of test classes that can use such specifications. This would be particularly useful in building systems using OO application frameworks [4] since, as we noted in Section 1, these frameworks typically make extensive use of template methods and usually come with informal documentation of the type we just described.

REFERENCES

[1] B. Beizer. *Black-box testing.* John-Wiley, 1995.

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

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

[4] M. Fayad, D. Schmidt, and R. Johnson. *Building application frameworks.* Wiley, 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software.* Addison-Wesley, 1995.

[6] E. Gunnerson. *A programmer's introduction to C#.* Apress, 2001.

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

[8] P Hsia, X Li, DC Kung, CT Hsu, L Li, Y Toyoshima, and C Chen. A technique for the selective revalidation of OO software. *Software Maintainence: Research and Practice*, 9:217–233, 1997.

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

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

[11] B Meyer. .NET is coming. *IEEE Computer*, 34(8):92–97, 2001.

[12] G. Myers. *The Art of Software Testing.* John Wiley, 1979.

[13] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(1):319–340, 1976.

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

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

[16] N. Soundarajan and S. Fridella. Framework-based applications: From incremental development to incremental reasoning. In W. Frakes, editor, *Proc. of Sixth Int. Conf. on Software Reuse: Advances in Software Reusability*, LNCS 1844, pages 100–116. Springer, 2000.

[17] N. Soundarajan and B. Tyler. Specification-based incremental testing of object-oriented systems. In Q. Li, D. Firesmith, R. Riehle, G. Pour, and B. Meyer, editors, *Technology of Object Oriented Languages and Systems 39*, pages 35–44. IEEE Computer Society Press, 2001.

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

[19] 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.

ABOUT THE AUTHORS

Neelam Soundarajan is an Associate Professor in the Computer and Information Science Dept. at the Ohio State University. His primary interests are in Software Engineering, specifically in reasoning about program behavior.

Benjamin Tyler is a graduate student in the Computer and Information Science Dept. at the Ohio State University. Ben is interested in various aspects of Software Engineering; his Ph.D. work deals with developing techniques and tools for testing OO programs.