Black-box Testing of Grey-box Behavior

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

Abstract

A key aspect of the Object-Oriented (OO) approach is that a designer can enrich an OO system by providing suitable (re-)definitions for some of the methods of the given system. *Application frameworks* provide good examples of such enrichment. An OO framework typically provides a number of *template* methods that implement specific patterns of calls to *hook* methods. An application developer can customize the framework by simply providing definitions, suited to the needs of the particular application, for the hook methods. The calls, made by the template methods of the framework, are dispatched at run-time to the hook methods defined in the application, thereby customizing the behavior of the template methods as well.

Testing such systems should include testing the hook method call-patterns of the template methods, that is their *grey-box* behavior. But software vendors often do not provide the source code of their systems. This poses the challenge, how can we test the grey-box behavior of a system without being able to access and instrument the code? In this paper, we develop an approach that allows us to do this, and demonstrate it on a simple case study.

1 Introduction

An important feature of object-oriented (OO) languages is the possibility of enriching or extending the functionality of an OO system [16] by providing, in derived classes, suitable definitions or re-definitions for some of the methods of some of the classes of the given system. *Application frameworks* [7, 11, 18] provide compelling examples of such enrichment. The framework includes a number of *hooks*, methods that are not (necessarily) defined in the framework but are invoked in specific, and often fairly involved, patterns by the *polymorphic* or *template* methods [9] defined in the framework. An application developer can build a complete customized application by simply providing appropriate (re-)definitions for the hook methods, suited to the needs of the particular application. The calls to the hook methods from the template methods are *dispatched* to the methods defined by the application developer, so that the template methods also exhibit behavior tailored to the particular application. Since the patterns of hook method calls implemented in the template methods are often among the most intricate part of the overall application, a well designed framework can be of great help in building applications, and maximizes the amount of reuse among the applications built on it. Our goal is to investigate approaches to perform specification-based testing of such frameworks.

Testing such systems should clearly include testing these patterns of hook method calls. That is, we are interested in testing what is called the *grey-box* behavior [2, 4, 8, 20] of OO systems, not just their *black-box* behavior. If we had access to the *source code* of the template methods, we could instrument the code by inserting suitable instructions at appropriate points to record information about the hook method calls; for example, just prior to each call

to a hook method, we could record the identity of the method being called, the values of the arguments, etc. But framework vendors, because of proprietary considerations, often will not provide the source code of their systems. Hence the challenge we face is to find a way to test the grey-box behavior of template methods without being able to make any changes to its code such as adding "monitoring code", indeed without even having the file containing source code of the system.

In this paper, we develop an approach that allows us to do this. Given a framework, the corresponding testing system we build using our approach itself turns out to be an application built on the framework. This 'testing application' can be generated automatically given information about the structure of the various classes that are part of the framework including the names and parameter types of the various methods and their specifications, and the compiled code of the framework. We have implemented a prototype *test system generator*; we will present some details about our prototype later in the paper.

How do we specify grey-box behavior? Standard specifications [12, 16] in terms of *pre-* and *post-* conditions for each method of each class in the system only specify the black-box behavior of the method in question. Consider a template (or polymorphic, we will use the terms interchangeably) method t(). There is no information in the standard specification of t() about the hook method calls that t() makes during execution. We can add such information by introducing a *trace* variable [4, 20], call it τ, as an *auxiliary* variable [17] on which we record information about the hook method calls t() makes. When the method starts execution, τ will be the empty sequence since at the start, t() has not made any such calls. As t() executes, information about each hook method call it makes will be recorded on τ. We will use a specification in which the post-condition of t() will give us not only information on the state of the object in question when t() terminates, but also about the value of τ, i.e., about the hook method calls t() made during its execution; we will see examples of this later in the paper. Given such a grey-box specification, the key question we address is, how do we test t(), without accessing or modifying its code, to see if its actual grey-box behavior satisfies the specification?

1.1 Related work

A number of authors have addressed problems related to testing of polymorphic interactions [15, 1, 19] in OO systems. In all of this work, the approach is to try to test the behavior of a polymorphic method t() by using objects of all or many different derived classes to check whether t() behaves appropriately in each case, given the different hook method definitions to which the calls in t() will be dispatched, depending on the particular derived class that the given object is an instance of. Such an approach is not suitable for testing frameworks. We are interested in testing the framework independently of any application that may be built on it, i.e., independently of particular derived classes and particular definitions of the hook methods. The only suitable way to do this is to test it directly to see that the actual sequences of hook method calls it makes during the tests are consistent with its grey-box specification. The other key difference is our focus on testing polymorphic methods without having to their source code.

Let us now consider the question of test coverage. Typical coverage criteria that have been proposed [1, 19] for testing polymorpohic code have been concerned with measuring the extent to which, for example, every hook method call that appears in the polymorphic method is dispatched, in some test run, to each definition of the hook method (in the various derived classes). Clearly a criterion of this kind would be inappropriate for our purposes since our goal is to test the polymorphic methods of the framework independently of any derived classes. What we should aim for instead is to select test cases in such a way as to ensure that as many as possible of the sequences of hook method calls allowed by the grey-box specifications actually

appear in the test runs. One problem here, as in any specification-based testing approach, is that the specification only specifies what behavior is *allowed*; but it is not necessarily the case that the system is actually capable of exhibiting each behavior allowed by the specification; hence measuring our coverage by checking the extent to which the different sequences of hook method calls allowed by the specification show up in the test runs may be too conservative if the framework is not actually capable of exhibiting some of those sequences. Another approach, often used with specification-based testing, is based on partitioning of the input space, i.e., the set of values allowed by the pre-condition of the method. But partition-based testing suffers from some important problems [6, 10] that raise concerns about its usefulness. We will return to this question briefly in the final section but we should note that our focus in this paper is developing an approach that, without needing us to access or modifying the source code of a template method, allows us to check whether the method meets its grey-box specification during a test run, rather than coverage criteria.

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

- It identifies the importance of testing grey-box behavior of OO systems.
- It develops an approach to testing a system to see if it meets its grey-box specification without accessing or modifying the code of the system under test.
- It illustrates the approach by applying it to a simple case study.

In Section 2 we consider how to specify grey-box behavior. In Section 3, we develop our approach to testing against such specifications without accessing the code. We use a simple case study as a running example in Sections 2 and 3. In Section 4 we present some details of our prototype system. In Section 5, we summarize our approach and consider future work.

2 Grey-box Specifications

Consider the Eater class, a simple class whose instances represent entitites that lead sedentary lives consisting of eating donuts and burgers, depicted in Fig. 1. The methods Eat_Donuts() and

```
class Eater {
  protected int cals_Eaten = 0;
  public void Eat_Donuts(int n) {
    cals_Eaten = cals_Eaten + 200 * n;}
  public void Eat_Burgers(int n) {
    cals_Eaten = cals_Eaten + 400 * n;}
  public final void Pig_Out() {
    Eat_Donuts(2); Eat_Burgers(2); }
}
```

Figure 1: Base class Eater

Eat_Burgers() simply update the single member variable cals_Eaten which keeps track of how many calories have been consumed; the parameter n indicates how many donuts or burgers is to be consumed. Pig_Out() is a template method and invokes the hook methods Eat_Donuts() and Eat_Burgers().

Let us now consider the specification of Eater's methods. These can be specified as usual in terms of pre- and post-conditions describing the effect of each method on the member variables of the class. The @pre notation [21] in the

$$\begin{aligned}
&\mathsf{pre.Eater.Eat_Donuts(n)} \equiv (\mathsf{n} > 0) && (1.1)\\
&\mathsf{post.Eater.Eat_Donuts(n)} \equiv \\
&\quad (\mathsf{cals_Eaten} = \mathsf{cals_Eaten@pre} + 200 * \mathsf{n})\\
&\mathsf{pre.Eater.Eat_Burgers(n)} \equiv (\mathsf{n} > 0) && (1.2)\\
&\mathsf{post.Eater.Eat_Burgers(n)} \equiv \\
&\quad (\mathsf{cals_Eaten} = \mathsf{cals_Eaten@pre} + 400 * \mathsf{n})\\
&\mathsf{pre.Eater.Pig_Out()} \equiv \mathit{true} && (1.3)\\
&\mathsf{post.Eater.Pig_Out()} \equiv \\
&\quad (\mathsf{cals_Eaten} = \mathsf{cals_Eaten@pre} + 1200) && (1)
\end{aligned}$$

post-conditions in (1) allows us to refer to the value of the variable in question at the time the method was invoked. Thus the specifications of Eat_Donuts() and Eat_Burgers() state that each of them increments the value of cals_Eaten appropriately. Given the behaviors of these methods, it is easy to see that the template method

Pig_Out() will meet its specification that it increments cals_Eaten by 1200.

The Eater_Jogger class, depicted in Figure 2, is a derived class of Eater. It keeps track not

```
class Eater_Jogger extends Eater {
  protected int cals_Burned = 0;
  public void Jog() {
    cals_Burned = cals_Burned + 500; }
  public void Eat_Donuts(int n) {
    cals_Eaten = cals_Eaten + 200 * n;
    cals_Burned = cals_Burned + 5 * n;}
  public void Eat_Burgers(int n) {
    cals_Eaten = cals_Eaten + 400 * n;
    cals_Burned = cals_Burned + 15 * n;}
}
```

Figure 2: Derived class Eater_Jogger

only of cals_Eaten but also cals_Burned. The new method Jog() increments cals_Burned. More important, Eat_Donuts() and Eat_Burgers() have been redefined to update cals_Burned.

What can we say about the behavior of Pig_Out() in this derived class? More precisely the question is, if ej is an object of type Eater_Jogger, what effect will the call ej.Pig_Out() have on ej.cals_Eaten and ej.cals_Burned? The calls in Pig_Out() to the hook methods will be dispatched to the methods redefined in Eatern_Jogger. If we had access to the body of Pig_Out() (defined in the base class), we can see that it invokes Eat_Donuts(2) and then Eat_Burgers(2), and hence conclude, given the behaviors of these methods as redefined in Eater_Jogger, that in this class, Pig_Out() would increment cals_Eaten by 1200 and cals_Burned by 40. What if we did not have access to the body of Pig_Out() and had only the information provided by (1.3), its black-box specification?

Behavioral subtyping [13] provides part of the answer to this question. In essence, a derived class D is a behavioral subtype of its base class B if every method redefined in D satisfies its B-specification. If this requirement is met then we can be sure that in the derived class, a template method t() will meet its original specification ((1.3) in the case of Pig_Out()). This is because when reasoning about the behavior of t() in the base class, we would have appealed to the base class specifications of the hook methods when considering the calls in t() to these methods. If these methods, as redefined in D, satisfy those specifications, then clearly that reasoning still applies when the calls that t() makes to these methods are dispatched to the redefined versions in D. Our redefined Eat_Donuts() and Eat_Burgers() do clearly satisfy their base class specifications (1.1) and (1.2), hence Pig_Out() in the derived class will meet its base specification.

But this is only part of the answer. The redefined hook methods not only satisfy their base class specifications but exhibit *richer* behavior in terms of their effect on the new variable cals_Burned. Indeed, the whole point of redefining them was to achieve this richer behavior; after all, if all we cared about was the base class behavior, there would have been no need to redefine them at all. And the richer behavior of the hook methods in the derived class in turn leads to richer behavior of the template method. How do we reason about this richer behavior?

The richer behaviors of the redefined hook methods are easily specified and are given in

$$\begin{aligned} &\mathsf{pre.Eater.Eat\_Donuts(n)} \equiv (\mathsf{n} > 0) \qquad (2.1)\\ &\mathsf{post.Eater.Eat\_Donuts(n)} \equiv \\ &\quad (\mathsf{cals\_Eaten} = \mathsf{cals\_Eaten@pre} + 200 * \mathsf{n}) \\ &\wedge (\mathsf{cals\_Burned} = \mathsf{cals\_Burned@pre} + 5 * \mathsf{n}) \end{aligned}$$

$$\begin{aligned} &\mathsf{pre.Eater.Eat\_Burgers(n)} \equiv (\mathsf{n} > 0) \qquad (2.2)\\ &\mathsf{post.Eater.Eat\_Burgers(n)} \equiv \\ &\quad (\mathsf{cals\_Eaten} = \mathsf{cals\_Eaten@pre} + 400 * \mathsf{n}) \\ &\wedge (\mathsf{cals\_Burned} = \mathsf{cals\_Burned@pre} + 15 * \mathsf{n}) \end{aligned}$$

(2.1) and (2.2). Can we use these richer specifications of the hook methods and the black-box specification (1.3) of Pig_Out() to arrive at the richer behavior of Pig_Out() in Eater_Jogger, in particular that it will increment cals_Burned by 40? The answer is clearly no, since there is nothing in (1.3) that tells us which, if any, hook methods Pig_Out() calls and how many times and with what argument values. Given (1.3), it is possible that it called Eat_Donuts() once with 6 as the argument and never called Eat_Burgers(); or Eat_Burgers() once with 3 as

the argument, and Eat_Donuts() zero times; it is even possible that Pig_Out() didn't call either hook method even once and instead directly incremented cals_Eaten by 1200. All of these and more –for e.g., it could have called Eat_Donuts() ten times with 2 as the argument each time and then decremented cals_Eaten by 2800– are possible; and depending on which of these Pig_Out() actually does, its effect on cals_Burned will be different. Note that for all of these cases, the original behavior (1.3) is still satisfied. *That* is ensured by behavioral subtyping. But if we are to arrive at the richer behavior of Pig_Out(), we need not just the black-box behavior of the template method in the base class as specified in (1.3), but also its grey-box behavior.

Consider the *grey-box* specification (1.3′). The label gb on the pre- and post-conditions

$$\begin{aligned}&\mathsf{gb.pre.Eater.Pig\_Out()} \equiv (\tau = \varepsilon) \quad (1.3')\\ &\mathsf{gb.post.Eater.Pig\_Out()} \equiv\\ &\quad [\,(\mathsf{cals\_Eaten} = \mathsf{cals\_Eaten@pre} + 1200)\\ &\quad \wedge (|\tau| = 2) \wedge (\tau[1].m = \text{“}\mathsf{Eat\_Donuts}\text{”})\\ &\quad \wedge (\tau[1].a = 2) \wedge (\tau[2].m = \text{“}\mathsf{Eat\_Burgers}\text{”})\\ &\quad \wedge (\tau[2].a = 2)\,]\end{aligned}$$

indicate that this is a grey-box specification. τ is the *trace* of this template method. τ is the empty sequence, ε, when t() begins execution. Each time Pig_Out() invokes a hook method, we add an element to record this hook method invocation. This element contains the name of the hook method called, the values of the member variables of the Eater class at the time of the call, their values at the time of the return from this call, the values of any additional arguments at the time of the call, their values at the time of the return, and the value of any additional result returned by the call. The grey-box postcondition gives us information about the value of τ when the method finishes, hence about the hook method calls it made during its execution. Thus (1.3′) states that $|\tau|$, the length of, i.e. the number of elements in, τ is 2; that the hook method called in the first call, recorded in the first element $\tau[1]$ of the trace, is Eat_Donuts; that the argument value passed in this call is 2; the hook method called in the second call is Eat_Burgers; and the argument passed in this call is 2. But note that (1.3′) does not give us information about the value the cals_Eaten had at the time of either call or return. While this simplifies the specification, it also means that redefinitions of the hook methods that depend on the value of cals_Eaten cannot be reasoned about given (1.3′). This is a tradeoff that we have to make when writing grey-box specifications; include full information, resulting in a fairly complex specification; or leave out some of the information, foreclosing the possibility of some enrichments (or at least of reasoning about such enrichments, which amounts to the same thing in the absence of access to the source code of the template method).

Given this grey-box specification, what can we conclude about the behavior of Pig_Out() in the derived class? Note first that from (2.1) and (2.2), we can see that Eater_Jogger.Eat_Donuts() and Eater_Jogger.Eat_Burgers() satisfy (1.1) and (1.2), i.e., they satisfy the requirement of behavioral subtyping; hence in Eater_Jogger, Pig_Out() will satisfy (1.3′). But we can also conclude given (2.1) and (2.2) and, as specified by (1.3′), that Pig_Out() will make two hook method calls during its execution, first to Eat_Donuts() with argument value 2, and then to Eat_Burgers() with argument value 2, that in Eater_Jogger, Pig_Out() will increment cals_Burned by 40. In [20], we have proposed a

$$\begin{aligned}&\mathsf{gb.pre.Eater\_Jogger.Pig\_Out()} \equiv (\tau = \varepsilon) \quad (2.3')\\ &\mathsf{gb.post.Eater\_Jogger.Pig\_Out()} \equiv\\ &\quad [\,(\mathsf{cals\_Eaten} = \mathsf{cals\_Eaten@pre} + 1200)\\ &\quad \wedge (\mathsf{cals\_Burned} = \mathsf{cals\_Burned@pre} + 40)\\ &\quad \wedge (|\tau| = 2) \wedge (\tau[1].m = \text{“}\mathsf{Eat\_Donuts}\text{”})\\ &\quad \wedge (\tau[1].a = 2) \wedge (\tau[2].m = \text{“}\mathsf{Eat\_Burgers}\text{”})\\ &\quad \wedge (\tau[2].a = 2)\,]\end{aligned}$$

set of rules that can be used in the usual fashion of axiomatic semantics to show: first, that the body of Pig_Out() defined in Fig. 1 satisfies the grey-box specification (1.3′); and second, by using the *enrichment rule* to "plug-in" the richer behavior specified in (2.1) and (2.2) for the redefined hook methods into (1.3′), that in the derived class, the template method will satisfy the richer specification (2.3′). But our goal here

is to test the template method to see whether it satisfies its specification, so we turn to that.

If we wished to test Pig_Out() against its black-box specification, the task can be carried out in a standard, straightforward fashion [16]. All we would need to do is create an object ee of type Eater, check that it satisfies the pre-condition given in (1.3) (which in this case is vacuous since it is simply true), apply Pig_Out() on ee, and check, when control returns, whether the post-condition specified in (1.3) is satisfied. But testing the grey-box behavior (1.3′) is more complex. First, (1.3′) refers to τ, and τ is not an actual variable of the class, but an *auxiliary* variable introduced for the purpose of specification. We can take care of this by introducing a trace variable, call it tau, as part of our testing setup and initialize it to the empty sequence immediately before invoking Pig_Out(). More seriously, tau needs to be updated whenever Pig_Out(), calls one of the hook methods; else, the value of tau will remain as ε and will not satisfy the conditions specified by (1.3′) even if in fact Pig_Out()'s grey-box behavior is in accordance with (1.3′). The obvious way to update tau would be to examine the code (in Fig. 1) of Pig_Out(), identify all the calls that appear in this code body to hook methods, and insert appropriate instructions into the body of Pig_Out() at these points to update tau appropriately. Thus we would replace the call Eat_Donuts(2) by:

Eat_Donuts(2); tau = tau ^ (Eat_Donuts, 2); (3)

where "^" denotes appending the specified element to tau; calls to Eat_Burgers() would be handled similarly. Once we insert these instructions, we go through our testing procedure. When Pig_Out() finishes, tau would indeed have been updated appropriately, and we can check whether the post-condition in (1.3′) is satisfied.

As we saw earlier, each element of the trace should record not just the name of the hook method called and the argument value passed, but also the state of the object at the time of the call as well as when the call returns. Thus what we have is incomplete. This does not matter in this example since the grey-box specification (1.3)′ does not refer to any of this additional information. In general though, it is necessary to include all of this information in each element of tau; and it is straightforward (if a bit tedious) to do this by modifying the instructions in (3) appropriately. But this approach does not meet our requirements. As we have noted before, we may not have access to the source code of the template method we want to test. Therefore, we certainly cannot make changes of the kind specified in (3) to that code. The fundamental problem we have to address is, how do we ensure that the trace tau is appropriately updated to record the hook-method calls that Pig_out() makes during its execution, without modifying its code, given that these calls are embedded in that code? In other words, how do we do *black-box* testing of Pig_Out()'s grey-box behavior?

3 Black-box Testing

The key problem we face in black-box testing of the grey-box behavior of Pig_Out() is that we cannot wait until it finishes execution to try to record information about its hook-method calls since, in general, by that point we no longer have that information. What we need to do instead is to *intercept* these calls as Pig_Out() makes them. But how can we do that if we are not allowed to modify Pig_Out() at the points of these calls? The answer is provided by the same mechanism that template methods are designed to exploit, i.e., *polymorphism*. That is, rather than intercepting the calls by modifying the code of the template method, we will redefine the hook methods so that *they* update the trace appropriately whenever they are invoked.

In Figure 3 we define our test class, Test_Eater. Since in the post-conditions of methods we are allowed to use, by means of the @pre-notation, the values that variables had when the method started execution, when testing against such specifications we need to save these initial values when a method begins executions. Thus in the test_Pig_out() method of Test_Eater, we use old_cals_Eaten to save the starting value of cals_Eaten.

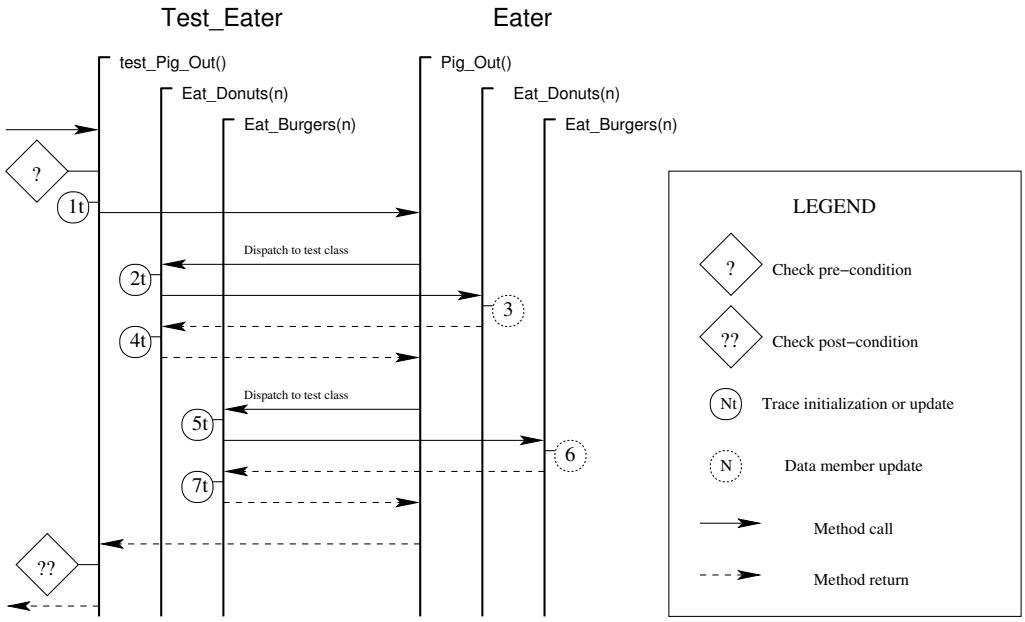

Figure 4: Sequence Call Diagram for Test_Eater.Pig_Out()

```
class Test_Eater extends Eater {
 protected trace tau;
 public void Eat_Donuts(int n) { // redefining hook
  traceRec tauel;
  tauel = ... info such as name of method called
   (Eat_Donuts), parameter value (n) etc.
  super.Eat_Donuts(n); // call original hook
  tauel = ... add info about current state etc.
  tau.append(tauel); }
 // Eat_Burgers() is similarly redefined.
 public void test_Pig_Out() {
  if (true) {
   int old_cals_Eaten = this.cals_Eaten;
    // allowed, since Test_Eater extends Eater
   tau = ε;
   this.Pig_Out();
   assert(grey-box post-condition of Pig_Out()
    with appropriate substitutions); };
 } }
```

Figure 3: Class Test_Eater

Test_Eater is a derived class of Eater, and we have redefined both the hook methods to update the trace. tau is the trace variable as before and tauel will record information about one hook method call which will be appended to tau once the call has finished and returned. Let us see how Test_Eater.test_Pig_Out() works using the *sequence call diagram*[3] in Fig. 4. The six vertical lines, each labeled at the top with the name of a method (the three on the left being from Test_Eater, the three on the right from Eater), represent time-lines for the respective methods. To test that Eater.Pig_Out() satisfies its grey-box specification, we create an appropriate instance of the Test_Eater class and apply Pig_Out() to it. This call 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 $\langle\rangle$ and saves the initial state in the old variable; this point is labeled (1t) in the figure. Next, it calls Pig_Out() (on the this object). Since Pig_Out() is not overridden in Test_Eater –it cannot be, since it is a final method– this is a call to Eater.Pig_Out(). This call is represented by the solid arrow from Test_Eater.test_Pig_Out() to Eater.Pig_Out().

Consider what happens when this method executes. First, it invokes Eat_Donuts() which we *have* overridden in Test_Eater, so this call is dispatched to Test_Eater.Eat_Donuts() since the object that Pig_Out() is being applied to is of type Test_Eater. This dispatch is represented by the solid arrow from the time-line for Pig_Out() 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 on the trace-record variable tauel; 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 appropriate additional information on tauel and appends this record to tau (represented by the point labeled (4t)), and finishes, so control returns to Eater.Pig_Out(); the return is indicated by the dotted arrow from Test_Eater.Eat_Donuts() to Eater.Pig_Out(). That method next calls Eat_Burgers() and this call is again dispatched to Test_Eater.Eat_Burgers(), represented by the solid arrow from Eater.Pig_Out() to Test_Eater.Eat_Burgers().

The process of recording initial information, delegating the call to the corresponding method in Eater, saving the result and appending the record to tau, 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 (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.Pig_Out(). That method is also done so it returns to Test_Eater.test_Pig_Out(). The final action, the one that we have been building up towards, is to check if the post-condition specified in the grey-box specification (1.3′) (with tau substituting for τ and old_cals_Eaten for cals_Eaten@pre) is satisfied, labeled by the diamond with the double question mark.

Thus by defining Test_Eater as a derived class of Eater, and by overriding the hook methods of Eater, we are able to exploit polymorphism to intercept the calls that the template method makes to the hook methods. This allows us to record information about these calls (and returns) without having to make any changes to the template method being tested, indeed without having any access to the source code of that method. This allows us to achieve our goal of black-box testing of the grey-box behavior of template methods[1].

If there were more than one template method, we could introduce more than one trace variable; but since only one template test method will be executing at a time, and it starts by initializing tau to $\langle\rangle$, this is not necessary. Consider now the derived class Eater_Jogger. How do we construct Test_Eater_Jogger? It should be a derived class of Eater_Jogger, not of Test_Eater, else the redefinitions of the hook methods in Eater_Jogger would not be used by the test methods in Test_Eater_Jogger. In general, test classes should be *final*. Test_C is only intended to test the methods of C. Another class D, even if D is a derived class of C, would have its own test class which would be a derived class of D.

4 Prototype Implementation

We have implemented a prototype testing system[2]. The system inputs the grey-box specifications for template methods of the class C

[1]Note that Test_Eater.Eat_Donuts() is not the test method for testing Eater.Eat_Donuts(). If we wished to test that method, we could include a test_Eat_Donuts() method in Test_Eater; this method would simply save the starting value of cals_Eaten, call the Eat_Donuts() method of the Eater class, then assert that the post-condition of (1.1) is satisfied when control returns from that call.

[2]Available at:
`http://www.cis.ohio-state.edu/~tyler`

under test, and the black-box specifications for the non-template methods. The system then creates the source code for the test class, along with other adjunct classes needed for the testing process, in particular those used in constructing traces when testing the template methods of `C`. The methods to be treated as hooks must be explicitly identified so that they are redefined in the test class. An alternate approach would have been to treat *all* non-final methods as hooks; but our approach allows greater flexibility. Each redefined hook method that the tool produces also checks its pre- and post-condition before and after the dispatched call is made. This helps pinpoint problems if a template method fails to satisfy its post-condition.

Currently, our system does not generate test cases, but creates skeleton calls to the test methods, where the user is required to construct test values by hand. To do the actual testing, the generated classes are compiled, and the test class executed. An example of the system's output is in Fig. 5. The last output shows a case

```
Test number 1: testing Eat_Donuts.
Test number 1 succeeded!

Test number 2: testing Eat_Burgers.
Test number 2 succeeded!

Test number 3: testing Pig_Out.
  Method Eat_Burgers called.
  Method Eat_Donuts called.
Postcondition of Pig_Out not met!
tau = (("Eat_Burgers", 1, 1),
    ("Eat_Donuts", 4, 4))
Test number 3 failed!

* * * RESULTS * * *
Number of tests run: 3
Number of tests successful: 2
```

Figure 5: Output from sample run

where the grey-box specification was not met. The problem was that the compiled `Eater` class had a bug in the code of `Pig_Out()`: it passed 1 as the parameter to `Eat_Donuts()` and 4 as the parameter to `Eat_Burgers()`; hence, although the black-box specification of `Pig_Out()` was satisfied, its grey-box specification was not.

5 Discussion

Our work was motivated by two observations: First, given that perhaps the most important aspect of template methods is the the hook method call patterns they implement, testing such methods requires us to test against their grey-box specifications. Second, application developers often build their systems using COTS components, including frameworks. If this developer wishes to test such a component, she will have to do so without having access to the source code of the component; Weyuker [22] also notes the importance of testing COTS components without having access to their source code. The approach we have developed addresses both of these considerations.

We conclude with some pointers for future work. We have ignored abstraction so far, instead working directly with the data members of the class under test. Cheon and Leavens [5] describe a testing system that can work with specifications that are given in terms of a conceptual model of the class under test. They do not consider grey-box specifications but we believe their approach can be extended to deal with grey-box behavior and we intend to explore that.

A more serious question is that of generating appropriate test cases to achieve reasonable coverage. As we noted earlier, our prototype system requires the human tester to provide the test cases. One interesting approach for generating test cases is used by TestEra [14] which is a system for specification-based testing Java programs. This system allows us to define, using a first-order relational language, complex properties that the objects must meet. Given a specification written in this notation, the system automatically generates instances that satisfies the pre-condition, so that we can then apply the method under test on the object in question. If the specification can be violated, TestEra generates a test case that shows that. The specifications that TestEra works with are black-box specifications; we plan to investigate whether a similar approach can be used to deal with grey-box specifications.

References

[1] R. Alexander and J. Offutt. Criteria for testing polymorphic relationships. In *Int. Symp. on Softw. Reliability Eng.*, pages 15–23, 2000.

[2] M. Barnett, W. Grieskamp, C. Kerer, W. Schulte, C. Szyperski, N. Tilmann, and A. Watson. Serious specifications for composing components. In *6th ICSE Workshop on Component-based Software Engineering*, pages 1–6, 2003.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

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

[5] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *Proc. of ECOOP 2002*, pages 231–255. Springer-Verlag LNCS, 2002.

[6] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. on Software Eng.*, 10:438–444, 1984.

[7] M.E. Fayad and D.C. Schmidt. Special issue on object oriented application frameworks. *Comm. of the ACM*, 40, 1997.

[8] G. Froehlich, H. Hoover, L. Liu, and P. Sorenson. Hooking into object-oriented application frameworks. In *Proc. of 1997 Int. Conf. on Software Engineering*, pages 141–151. ACM, 1997.

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

[10] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Software Eng.*, 16(12):1402–1411, 1990.

[11] R. Johnson and B. Foote. Designing reusable classes. *Journal of OOP*, 1:26–49, 1988.

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

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

[14] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *Proc. of 16th ASE*. IEEE, 2001.

[15] T. McCabe, L. Dreyer, A. Dunn, and A. Watson. Testing an object-oriented application. *J. of the Quality Assurance Institute*, 8(4):21–27, 1994.

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

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

[18] W. Pree. Meta patterns: a means for capturing the essentials of reusable OO design. In *Proceedings of the Eighth ECOOP*, pages 150–162, 1994.

[19] A. Rountev, A. Milanova, and B. Ryder. Fragment class analysis for testing of polymorphism in java software. In *Int. Conf. on Softw. Eng.*, pages 210–220, 2003.

[20] N. Soundarajan, S. Fridella. Framework-based applications: From incremental development to incremental reasoning. *Proc. of 6th Int. Conf. on Softw. Reuse*, pages 100–116. 2000.

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

[22] E. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, 1998.