Automatic Monitoring of Control-flow Through Inheritance Hierarchies

Benjamin Tyler
Computer Science & Eng.
Ohio State University
Columbus, OH 43210, USA
tyler@cse.ohio-state.edu

Neelam Soundarajan
Computer Science & Eng.
Ohio State University
Columbus, OH 43210, USA
neelam@cse.ohio-state.edu

ABSTRACT

Polymorphism, based on inheritance and dynamic binding in standard object-oriented languages, is one of the most powerful mechanisms available to the OO designer. It allows the system designer to customize the behavior of functions defined in particular base classes by suitably redefining, in derived classes, other functions that they invoke. At the same time, polymorphism, especially when used in conjunction with the `super` mechanism that most OO languages provide, can result in extremely complex control-flow among the various methods defined in the various classes. In this paper, we develop an approach that can be used by the designer to automatically trace this control-flow. We also present results from a prototype implementation based on our approach.

1. INTRODUCTION

A key aspect of the the Object Oriented (OO) approach is *polymorphism*[1]. Polymorphism enables a derived class designer, assuming that the base class has been suitably designed, to construct interesting and varied derived classes by just redefining an appropriate set of functions of the base class. Given such redefinitions, not only will the redefined functions exhibit new behavior, but since polymorphism ensures that calls in other functions of the base classes to these functions are dispatched to their redefined versions (assuming that the objects in question are instances of the derived classes), these other functions will also exhibit suitably enriched behavior. But polymorphism also poses some serious difficulties for the system designer. A fundamental problem [17, 2] has to do with the way that control flows between methods defined in different classes of an OO program. Our goal in this paper is to present an approach that exploits polymorphism in helping analyze this control flow.

There are two distinct aspects that contribute to the complexity of control-flow among the methods defined in different classes of a program. The first involves the related mechanisms of inheritance and dynamic binding. Thus in the example presented in [2], a fleshed-out version of which we will use as a small case-study in this paper, the problem shows up as follows: There are five classes, `C1` through `C5`, with `C5` being a derived class of `C4`, which is a derived class of `C3`, which in turn is a derived class of `C2`, etc. A method `c()` is defined in the class `C2`, inherited by `C3`, and redefined in `C4`, and inherited by `C5`. Another method `a()`, defined in `C1` contains, in its body, a call to `c()`. When this `a()` is applied to an object that is an instance of `C5`, dynamic binding will ensure that the call to `c()` that is made from within the body of `a()` will be dispatched to the `c()` defined in `C4`. On the other hand, if this `a()` were to be applied to an object that is an instance of `C3`, the same call will go to the `c()` defined in `C2`. This makes it rather difficult to follow the control-flow that results from applying a method such as `a()`; although that method is inherited by the various derived classes[2], what it actually does, in particular which method bodies it invokes as it executes, depends critically on the particular class of which the object in question is an instance.

The second aspect that contributes to the complexity of control-flow is, somewhat paradoxically, one designed to avoid the dynamic binding. Thus one of the methods, say, `c()` that is redefined in a derived class such as `C4` may contain, in the body of that redefinition, a call such as `super.c()`. The reason that standard OO languages provide the "`super`" mechanism is that often the redefinition of a method such as `c()` in a derived class has to perform all of the tasks carried out by the base class definition of the method, plus some additional activities typically related to the additional state (in the form of new member variables) of the derived class. While the former task could be achieved by duplicating the code of `c()`'s definition from the base class, the call `super.c()` serves the same purpose by invoking the base class `c()`. But this means that control transfers to the base class definition of the method –which was supposedly superseded by the derived class definition– which might invoke other methods that will be dynamically dispatched, unless those invocations also use the `super` mechanism, etc.

In our discussion, we will use the term *up-call* to refer to calls using the `super` mechanism since such a call will result in control going from the current method to a method defined in an ancestor class. Similarly, we will use the term *down-call* to refer to calls that are dispatched based on the class that the object on which the method is applied is an instance of. We should note that "down-call" is not always an accurate description; thus if a method `m()` defined in `C4` and not redefined in `C5` invokes `n()` in its body and `n()` is defined in `C3` and not redefined in `C4` or `C5`, and `m()` is applied to an instance of `C5`, then the call that `m()` makes to `n()` will be handled by `C3.n()`; thus control flows from `C4.m()` up to `C3.n()` since it is that definition of `n()` that applies to objects that are instances of `C5`. By contrast, the call `super.n()` necessarily results in control flowing up.

Taenzer, Ganti, and Podar [17] coined the term "yo-yo problem" to convey the effect of dynamically dispatched calls that typically transfer control to methods defined in derived classes, alternating with calls using the `super` mechanism that transfer control to methods in ancestor classes. To quote, "[t]he combination of polymor-

Submitted to SEKE 2006.

---

[1] Throughout, by 'polymorphism' we mean *inclusion* or *subtype* polymorphism [3] achieved in standard OO languages such as *Java* via *inheritance* and *dynamic binding*.

[2] The actual details of the example are slightly different, as we will see later in the paper.

phism and method refinement [i.e., methods that use inherited behavior by invoking the method defined in the parent class using the `super` mechanism] make it very difficult to understand the behavior of the lower level classes and how they work" [17]. Binder [2] argues that the "[l]oss of intellectual control that results from spaghetti polymorphism (the yo-yo problem) . . . " is one of the unique bug hazards of the OO approach.

Given this complexity, a graphical representation –which we call a *yo-yo graph*– of control flow through the inheritance hierachy would clearly be useful. Even more important is that the control flow in a system be monitored automatically by a suitable tool as the system executes; the information collected by the tool can then be used to generate the yo-yo graph. Similar graphs, generated by hand have been used by various authors. Given the complexity of the control flow and the resulting potential for mistakes in generating the graph by hand, the advantages of such a tool are clear.

In this paper, we present an approach to runtime monitoring of OO systems that allows us to automatically track the control flow through inheritance hierarchies. Interestingly, and this is a testament to the power of polymorphism, our approach, as we will see, exploits polymorphism for this purpose. Indeed, polymorphism allows us to capture the needed information without making *any* changes to the classes whose methods are to be monitored as far as tracking *down*-calls are concerned, and making only minimal changes for the purpose of tracking *up*-calls. We have implemented our monitoring approach in a prototype tool which, given a system and the list of names of the classes and methods to be tracked, automatically makes the needed changes to the system, and monitors the system at runtime, logging information about the control flow. Once the execution of the system completes, the tool generates the system's yo-yo graph based on the logged information.

The rest of the paper is organized as follows. In Section 2, we we present a fleshed-out version of an example from [17, 2] which we will use as our case-study. In Section 3, we develop our approach to tracing both down-calls and up-calls; provide some details of our prototype implementation based on our approach; and present results of using it on the case-study. In Section 4, we discuss related work. In Section 5, we summarize our approach and consider directions for future work.

2. CONTROL FLOW

Consider the program[3] shown in Fig. 1 consisting of classes `C1` through `C5`, with each class (except `C1`) being a derived class of the one immediately above it. This program is based on the one in [2]; the only changes we have made are to flesh out the individual method bodies to perform specific actions. But these actions are not really intended to be particularly interesting; our focus rather is on how control flows among the various methods as a result of the use of polymorphism and calls to `super` methods.

The `main()` function defined in `C5` creates an instance of `C5` and invokes `a()` on it. Since the closest ancestor of `C5` that has a definition of `a()` is `C4`, it is that definition that will be invoked. That method invokes `super.a()` which calls `C3.a()`, which in turn also invokes `super.a()` which calls `C1.a()`. That method invokes `b()` and then `c()` and these calls will be dispatched to their respective definitions applicable to instances of `C5`, i.e., `C3.b()` and `C4.c()` respectively, etc.

```
abstract class C1 {
   protected int x = 0;
   public void a() { x++; x = b(); c(x-1); }
   abstract public int b();
   abstract public void c(int k);  }
class C2 extends C1 {
   protected int y = 0;
   public int b() { y = 2*x; int j = d(y); return y+j; }
   public void c(int k) { x = x - k; }
   public int d(int k) { c(k+1); return x; }  }
class C3 extends C2 {
   protected boolean p;
   public void a() { p = !p; super.a(); }
   public int b() { return super.b(); }   }
   public void c(int k) { x = x + k; }
   public int d(int k) { return x - 1; }  }
class C4 extends C3 {
   protected int z;
   public void a() { super.a(); z++; }
   public void c(int k) { super.c(k); z=z+x; }
class C5 extends C4 {
   protected boolean q = true;
   public int d(int k) { q = !q; return super.d(k); }
   public static void main(String[] args)
                  { C5 c5 = new C5(); c5.a(); }   }
```

Figure 1: Program to be traced (original source code)

Fig. 2, based on the one in [2], represents the static inheritance structure of our program as well as the control flow. The left side of the figure depicts, for each class, the methods inherited from the parent class (these methods are labeled `inh`), defined or redefined in the class (these methods have no label next to their names), or as being redefined but the redefinition including a `super` call (labeled `ref` for "refinement"). The main part of the figure is yo-yo graph representing the control flow when the call `c5.a()` is executed. This graph seems to contain some errors. First, the method `d()` is redefined in `C3`, not inherited by it. Therefore the arrow going from `C5.d` to `C2.d` representing the `super.d()` call in the body of `C5.d()` should instead go to `C3.d`. To further add to the confusion, the discussion in [2] states that `C5.d()` invokes `super.a()`. If that were the case, the arrow from `C5.d` should go to `C3.a`, not `C2.d`. Similar problems can be seen with the method `c()`. `c()` is redefined in `C3`, not inherited. Therefore, the `super.c()` call in `C4`

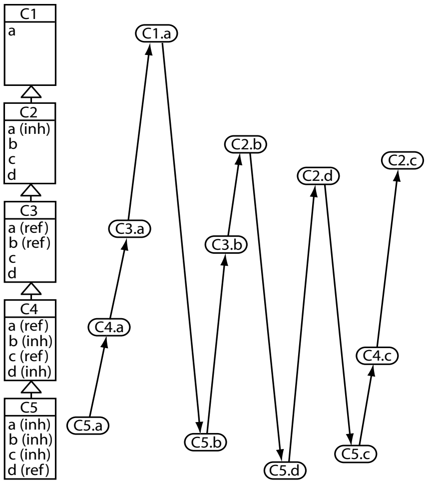

Figure 2: Yo-yo graph (generated by hand)

---

[3] For concreteness, we use *Java* in our discussions; but the approach does not depend on any unique facilities of *Java* such as *reflection*, and is usable for other common OO languages.

should go to C3.c, not C2.c; and, as in the case of C5.d(), an accompanying table states that C4.c() invokes super.a() (which would again be inconsistent with the arrow from C4.c to C2.c). It is possible that the problem lies with the left side of Fig. 2, i.e., the inheritance diagram rather than in the yo-yo graph. Thus according to the table accompanying the figure, methods c() and d() are *inherited* by C3, not redefined in this class. In that case, the arrow from C5.d to C2.d would indeed be correct as would the one from C4.c to C2.c. But the confusion regarding calls to super.a() that, according to the table, appear in C5.d() and C4.c() still remains.

In any case, what the example demonstrates is that even in relatively simple systems, with just a handful of base and derived classes, with methods redefined or refined in the derived classes, can lead to complex control flow. Therefore we need ways to automatically track the flow at runtime and tools that can produce suitable yo-yo graphs based on the information collected.

3. AUTOMATIC MONITORING

Given a system such as the one in Fig. 1, how can we automatically monitor the system to obtain information about how the control flows? In our approach, corresponding to each class Ci in the OO program whose control-flow we are interested in tracking, we will introduce a derived class TCi and define certain methods in TCi. If we are interested in seeing what control-flow would result when a method m() is applied to an object that is an instance of, say, C4, we create an object of type TC4 and apply m() to it. As we will see below, the TCi classes will be defined in such a way that the resulting control-flow will be essentially the same as if we had applied m() to an instance of C4; the only difference is that the "down" calls will, because of polymorphism, be "intercepted" by the methods we define in TC4 which will record suitable information about the call and then forward the call to the actual method that would have received the call if the object had been an instance of C4.

The up-calls present a more difficult challenge. The problem is that when a call such as super.n() is made from within the body of, say, C4.m(), control flows up to the n() defined in the closest ancestor of C4, *independent of the class that the current object is an instance of*. Therefore, there is no way to use polymorphism to intercept such a call. Given this, we will use a slightly more involved approach to handle these calls: In effect, in addition to certain methods that we will define in the TCi classes, we will also have to make certain minor modifications in the classes Ci. With this, we will be able to record the needed information about both the down-calls and the up-calls.

3.1 Tracking Down-calls

Consider a call such as m() that appears in the body of some method n() in some class Cj. Suppose the this object on which m() is being invoked is an instance of Ci. When this call is executed, it will be dispatched to the definition of m() that is in Ci or, if m() is not (re-)defined in Ci, the one in the closest ancestor of Ci that has such a definition.

In order to intercept such calls, in TCi, we will redefine every method[4] that is applicable to objects of type Ci. Consider, for example, the class TC4 corresponding to the class C4 that appears in Fig. 3. The methods applicable to C4 objects are a(), b(), c(), and d(). We have redefined each of these in TC4. All that the redefinitions do is to save information about the method call, invoke the method defined in the parent class (C4), and when that call returns, save information about the results, etc., and then return to the caller.

```
class TC4 extends C4 {
  public void a() {
    //. . . save information about this call . . .
    super.a();
    //. . . save information about return from call . . .  }
  public int b() {
    //. . . save information about this call . . .
    int x = super.b();
    //. . . save information about return from call . . .
    return x;  }
  public void c(int k) {. . . similar to a(); super.c(k);. . . }
  public int d(int k) { . . . similar to b() . . . }
```

Figure 3: TC4 class (first version)

Suppose now we have a variable xx (elsewhere in a portion of the program not shown in Fig.1) declared of type C2 that at run-time contained a reference to an object o1 that is an instance of C4; consider the call xx.b(); this call will be dispatched to C3.b() since that is the definition that applies to instances of C4. Thus this call will execute normally without being affected by TC4. Now suppose xx contained a reference to an instance of TC4 instead. In this case, the call xx.b() will be dispatched to TC4.b(). That method will save information about this call and then invoke super.b(); that invocation will then be forwarded to C3.b() since C4 is the base class of TC4 but C4 does not override b() but instead inherits it from its base class C3. Thus the method that is called at this point is the same as the one that was called when xx contained a reference to the C4 object. This method executes, and control then returns to TC4.b(). That method records information about the fact of the return, and finally returns the result returned by the call to super.b(). Thus the original call, xx.b() will receive the same value as it did when we were dealing with the original object of type C4 but now information about the call to and the return from b() has been saved.

We have not indicated how the information about the calls and the returns is saved, but those are primarily matters of detail that we will briefly address in Section 3.3. Let us now turn to the up-calls.

3.2 Tracking Up-calls

Consider the call super.a() that appears in the definition of C3.a(). When this call is executed, control will immediately transfer to C1.a() (since C2 does not redefine a()), independent of the run-time type of the object at hand. Thus, no matter what we do in the derived classes of C3, C4, etc., we cannot intercept this call. What we need to do instead is to rewrite these calls in such a manner that they can be intercepted.

```
class C4 extends C3 {
  public int z;
  public void a() { C4_super_a(); z++; }
  public void c(int k) { C4_super_c(k); z=z+x; }
  public void C4_super_a() { super.a(); }
  public void C4_super_c(int k) { super.c(k); }
```

Figure 4: Modified class C4

Consider the modified class C4 in Fig.4. This class differs from the original C4 in Fig.1 in two respects. First, we have introduced two new methods, C4_super_a() and C4_super_c() each of which simply calls the corresponding super method. Second, the super calls that appeared in the methods of the original C4 have been

---

[4] More precisely, we should say every *non-final* method will be overridden since final methods cannot, of course, be overridden.

replaced by calls to the corresponding new methods we have introduced; thus, the call super.a() in the original C4.a() has been replaced by a call to C4_super_a(), and similarly for the call to super.c() that appears in the original C4.c(). Note that we have not introduced methods C4_super_b() or C4_super_d() but that is because there are no calls of the form super.b() or super.d() in any of the methods of the original C4. If such calls had existed, we would have defined these methods. Alternately, we could introduce all such methods independent of whether the corresponding super calls exist since in those cases where such calls do not exist, these new methods will not be invoked.

With these changes, the modified C4 will still behave in exactly the same way as the original C4 as far as instances of C4 are concerned. To track the up-calls, we need to modify our TC4. In the TC4 defined in Fig. 5, we have redefined the methods C4_super_a() and C4_super_c() in exactly the same way as we redefined the original methods a(), b(), etc., in Fig. 3. Therefore, if we use an object that is an instance of TC4, rather than an instance of C4, and apply the method a() to it, the call to C4_super_a() in the modified C4 –which has replaced the call to super.a() that appeared in the original C4– will be dispatched to the redefinition of C4_super_a() in TC4. This method, as usual, saves information about this call, then forwards the call to C4.C4_super_a(), which in turn forwards the call to the a() defined in C3, which was the method we called from the original C4.a().

```
class TC4 extends C4 {
  // a(), b(), c(), d() as in Fig. 3
  public void C4_super_a() {
    //. . . save info about this *super* call . . .
    super.C4_super_a();
    //. . . save info about return from call . . .  }
  public void C4_super_c(int k) { . . .  similar . . . }
```

Figure 5: TC4 class (second version)

This seems to work but there is a subtle problem. Consider what happens when that method, C3.a(), executes. We would, of course, have modified C3 in the same manner as we have modified C4. So the super.a() call that appears in the original C3.a() would now be the call C3_super_a(). This method would be defined, analogously to C4_super_a(), to simply consist of a call to super.a(). Moreover, C3_super_a() is *not* redefined in TC4; it *will* be redefined in TC3 (in the same manner as C4_super_a() in TC4) but that definition does not apply here since the object at hand is of type TC4, not TC3, and there is no inheritance relation between these two classes. So the call to C3_super_a() will be handled by C3.C3_super_a() which will simply call super.a(). And that call will go to the a() defined in C1. That is what we want since that is the effect of the call in the original C3.a() but we *did not intercept* this super call, so no information about it has been recorded. Hence a yo-yo graph constructed from the information saved by classes such as the TC4 in Fig. 5 will miss some of the up-calls.

The solution is to redefine, in TC4, not just the methods a(), b(), etc., and the methods C4_super_a() C4_super_c() etc., but also methods such as C3_super_a(), and by extension, C2_super_a(), etc. The TC4 defined in Fig. 6 does that. With this addition to TC4, if the method a() is applied to an instance of TC4, not only is the call to C4_super_a() intercepted, but also the call to C3_super_a(). Therefore, this final version of TC4 will allow us to intercept all down-calls and up-calls invoked upon objects of type TC4 and save the necessary information about these calls.

```
class TC4 extends C4 {
  // a(), b(), c(), d() as in Fig. 3
  public void C4_super_a() {// as in Fig. 5 }
  public void C4_super_c(k) { . . .  similar . . . }
  public void C3_super_a() {
    //. . . save info about *this* super call (to C2.a()) and
    // the current state of object. . .
    super.C3_super_a();
    //. . . save info about return from call and
    // current state of object, results . . .   }
  public void C3_super_b(k) { . . . similar . . . }
  public void C3_super_c(k) { . . . similar . . . }
  public void C2_super_c(k) {
    // this one is not needed since there are no super.c()
    // calls in the methods of C2, but it would be more
    // uniform to have all of these . . . }
  // redefinitions of other _super_ methods  }
```

Figure 6: TC4 class (final version)

3.3 Implementation Details and Results

We have implemented our approach to automatically tracking control flow in a prototype tool, *PolyTracker*[5]. The tool operates in two phases. The first phase takes as input, the original program, consisting of all the original classes, which we have referred to as C1, C2 etc., in our discussion. It modifies all of these classes, replacing the super calls that appear in any of the methods in any of the classes, by the call to the corresponding Ci_super_ call. It also introduces the trivial definitions for the Ci_super_ methods in these classes, similar to the ones in Fig. 4. Next it produces the tracing classes, the ones we have referred to as TC1, TC2, etc. As we saw, these classes do not in any way depend upon the details of the methods defined in the original classes. All that is needed to define the tracing classes are the names and the parameter and result-type information about the various methods defined in the original classes. Thus the tracing classes are produced easily. For simplicity, we define all possible methods of the kind Ci_super_ in the modified Ci classes, and the corresponding redefinitions in the TCi classes, even if there are no calls to the corresponding super methods.

The main() method that appeared in one of the original classes would typically create an instance of one of the Ci classes, and then invoke some method m() to it. In an actual program, there may, of course, be additional methods invoked upon this object. Indeed, instances of many of the Ci classes may be constructed and various methods applied to them in various orders. Our approach can handle such situations and we will briefly consider this in the final section. Here we assume that only one instance of one of the Ci classes is constructed and only one method is invoked on it by the main() method of the program. In the modified program, *PolyTracker* replaces this instance by an instance of the corresponding trace class TCi. To log the information about the various calls and returns that are intercepted, all of the TCi classes use an instance of Tracker, a *singleton* class that *PolyTracker* uses for this purpose. A Tracker object is essentially a *sequence* of elements, each of which contains information about a single call or a return. We will not go into further details about the structure of the Tracker object, referring the interested reader to the documentation at the *PolyTracker* site.

Once the modified Ci, including the modified main() method,

---

[5] *PolyTracker*, its documentation, and examples are available at: www.cse.ohio-state.edu/~tyler/pTracker/index.html

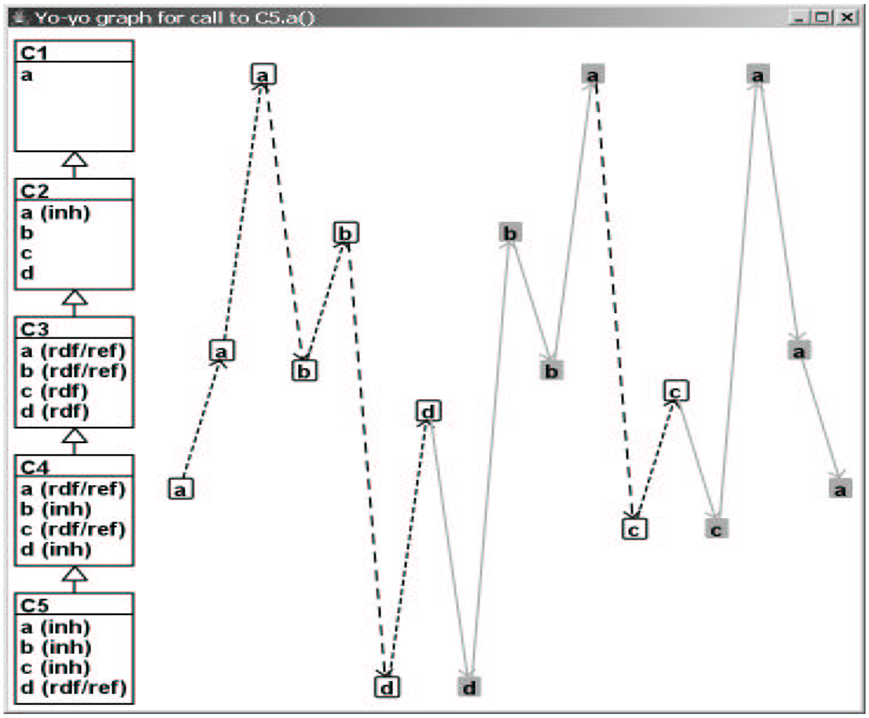

Figure 7: Yo-yo graph (generated by *PolyTracker*)

and the tracing classes have been produced, the resulting set of classes (as well as `Tracker` and related classes) are compiled using the standard *Java* compiler. The program is then executed. During execution, the various `TCi` classes collect and save information about the control flow in the `Tracker` object.

When the execution completes, we are ready to enter phase two, and render the yo-yo graph. Again we will omit the details of this rendering which are available at the *PolyTrakcer* site. Fig. 7 shows the graph generated by *PolyTracker* for our simple case-study in-Fig. 1. As in the case of the hand-generated graph in Fig. 2, on the left side of the figure, we have the classes along with the inheritance relations among them. Inherited methods are labeled `inh`. Methods that are redefined and, in their redefinition, include a `super`-call are labeled `rdf/ref`. Methods that are redefined but do not include such a call are labeled `rdf`. Methods that are defined for the first time do not have any label. We should also note that we use the label `ref` only in those cases where the redefinition of a method in the derived class invokes the `super`-version of the *same* method. If the method invokes the `super`-version of a different method, that is labeled `sup` (but there are no examples of this in Fig. 7). This seems appropriate since this is not a *refinement* in any real sense.

Some other points worth noting. First, when generating the yo-yo graph, *PolyTracker* uses different styles of arrows (short versus long dashes) to distinguish between up-calls and down-calls. Second, no *call nodes* –the small boxes representing the method definition executed for each invocation– are created for those class methods that are inherited. The rationale for this is that if a class `Cj` inherits a method `m()`, then there is no code for `m()` in `Cj` to be executed; such invocations "pass through" the class `Cj` to its first ancestor class that does implement `m()`. Some authors do use such call nodes for inherited methods in the bottom-most class in the hierarchy but more recent literature, such as [12], do not use them and we have adopted the same approach.

A more important point is that in addition to the flow of control at the time of the call, the yo-yo graph produced by *PolyTracker* also depicts the flow of control *as it returns* to the calling method. Control returning to a calling method is illustrated by the *grey* nodes and arrows in our graph, as against the white nodes outlined that represent control resulting from the initial method invocation. Including the *return nodes* results in a more informative graph since it shows the control-flow through the complete execution of a method that makes *multiple* calls, and accurately depicts the dependencies between the calling and the called methods.

This can be seen when we look closer at the traditional yo-yo graph in Fig. 2, alongside the associated program code in Fig. 1. In the code, we see that `C1.a()` makes two calls: one to `b()`, and one to `c()`. The first call is shown in the graph by the arrow starting from the node `C1.a`, found at the top of the diagram, and going down to the node `C5.b`. The second call, however, and the subsequent calls resulting from it are *not* shown. The call to `c()` that *is* shown is the one initiated from `C2.d`, not `C1.a`; this call was made as a consequence of `C1.a()`'s first call to `b()`, and is not related to the second call that it should make. The graph does not depict what happens after `C1.a()`'s first call to `b()` ends, and so does not go through what happens when it invokes `c()`. If we were to represent this call to `c()` by `C1.a` by simply linking the last call node visited (in this case, `C2.c`) to the node representing this call, it may appear that `C2.c()` is itself making a call to `c()`, which it is not. Chaining the call nodes together by chronology in this way is of limited use from the standpoint of program understanding and, especially, debugging since the notion of dependencies among method bodies would be lost. By contrast, our yo-yo graphs explicitly show how control returns to `C1.a()` from its call to `b()`, and then show what happens when it invokes `c()`.

Before concluding, we note that the problems we noted earlier in the graph in Fig. 2 do not appear in this graph. As we noted earlier, the `Tracker` object collects much more information about the individual calls and returns. The tool provides suitable facilities to enable a user to extract this information, and this can be valuable when debugging or tracing through complex systems. These details are available at the *PolyTracker* site.

4. RELATED WORK

The complexity of control-flow among methods defined in various classes in standard OO programs has been discussed by a number of authors. We have already mentioned the work of Taenzer *et al.* [17] and Binder [2]. Lange and Nakamura [7, 8] present a technique for tracing the execution of an OO program. Their technique is based on accessing, at the machine level, specific information contained in the run-time structures as the program executes. Hence this is specific to not just the language but also the particular implementation; on the plus side, they can extract considerably more information about the program's execution. They also discuss various graphical ways to display the information, such as interaction charts that indicate, on each object's lifeline, the invocations the particular object makes. De Pauw *et al.* [13] present an approach to visualizing the execution of OO programs, including object construction, destruction, method calls, etc. Their technique requires insertion of substantial amounts of code in the individual classes (which will then have to be removed once we are satisfied with the system), and depends on the *RTTI* mechanism of *C++* to access, at run-time, information about the actual types of given objects. Jerding [6] discusses ways in which the execution of OO programs can be visualized and displayed graphically. His main concern is to filter and extract the most relevant pieces from the large amount of information that may be obtained about the program's execution so that what is displayed is easy to comprehend and at the same time useful. He does not discuss the question of how to obtain information which is the main focus of our work.

Several authors have addressed problems related to testing of polymorphic interactions [9, 1, 14] in OO systems and related questions concerning coverage. The approach usually is, given the entire system, test the behavior of each polymorphic method `t()` by considering various possible bindings, i.e. by using instances of each of the derived classes, and check whether the resulting (functional) behavior of `t()` is appropriate in each case. The appropriateness of the behavior is determined on the basis of the output results produced by `t()` when it finishes. The question of automatically tracing the flow of control among the various methods which is the focus of our work, does not seem to have been addressed.

Some authors [11, 15, 16] have argued that given the complexity that results from their use, inheritance and polymorphism based on dynamic binding should be avoided or minimized as much as possible. On the other hand, several authors, for example [10, 4, 5], provide convincing arguments for, and compelling examples that demonstrate the power of, these mechanisms in building complex systems. In any case, given the many systems that do exploit these mechanisms, techniques and tools such as *PolyTracker* can be of great value in understanding and building these systems.

5. DISCUSSION

The use of polymorphism/dynamic binding and the `super` mechanism can lead to relatively complex control flow in OO systems. Representing this graphically in the form of yo-yo graphs can be of considerable help to designers and implementers but the very complexity of the flow means that generating the graphs by hand can result in subtle mistakes in the graphs. Our work shows how the power of these same mechanisms can be exploited to build a tool that can track the control flow automatically and use the data collected during the monitoring to generate the yo-yo graphs.

We conclude with some pointers to future work. So far in our work, we have only dealt with a single object and the control-flow that results from applying a particular method on it. In practical systems we will have to deal with multiple objects. Our approach should be directly applicable to such situations. Indeed, we can *selectively* trace methods invoked on *some* objects and ignore those invoked on others. To do this, we simply have to ensure that the objects for which calls should be traced, should be of the appropriate `TC` type, while the others should be of their original `C` type.

A more complex issue has to do with the fact that the program under study may have a class `C1` that has a member variable `x` of type, say, `C2` rather than the simple types such as `int`s. Suppose the program has an object `o1` of type `C1`. What if we wish to trace not just method invocations on `o1` but those invoked on `o1.x`? That is an object of type `C2` and normally we would simply create and use an instance of type `TC2` for this purpose. But that will not work since this is a part of the `o1` object, rather than being an independent object. We could create an `o1` object that is of type `TC1` but the `x` component of this object will still be of type `C2` rather than `TC2`. Finding a suitable solution to this problem is important since practical systems are likely to have such objects that have other objects as components and designers and analysts are likely to be interested in the behaviors of those components. Less challenging but still of practical importance is the question of finding suitable ways to display information about the control-flow when the system has several objects that we are interested in; here, we should be able to build on previous work in visualization [6, 8].

6. REFERENCES

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

[2] R. Binder. *Testing OO systems*. Addison-Wesley, 1999.

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

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

[5] M.E. Fayad and D.C. Schmidt. Sp. issue on OO application frameworks. *CACM*, 40, 1997.

[6] D. Jerding. Visualizing patterns in the execution of OO programs. In *Proceedings of CHI '96*. ACM, 1996.

[7] D. Lange and Y. Nakamura. Interactive visualization of patterns. In *Proc. OOPSLA*, pages 342–357. ACM, 1995.

[8] D. Lange and Y. Nakamura. Object-oriented program tracing and visualization. *Computer*, pages 63–70, Oct. 1997.

[9] 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.

[10] B. Meyer. *OO Software Construction*. Pren. Hall, 1997.

[11] S. Muralidharan and B. Weide. Should data abstraction be violated to enhance software reuse? In *Proc. 8th Ann. Natl. Conf. on Ada Tech.*, pages 515–524. ANCOST, 1990.

[12] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson. A fault model for subtype inheritance and polymorphism. In *Int. Symp. on Softw. Reliability Eng.*, pages 84–95, 2001.

[13] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. of OOPSLA '93*, pages 326–337. ACM, 1993.

[14] 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.

[15] A. Snyder. Inheritance and development of softw. components. In *Res. Dir. in OO Prog.* MITP, 1987.

[16] C. Szyperski. *Comp. softw.:Beyond OOP*. Addison, 1998.

[17] D. Taenzer, M. Ganti, and S. Podar. Problems in OO software reuse. In *Proc. of ECOOP*, pages 25–38, 1989.