Integration Testing of Procedural Object Oriented Programs with Polymorphism *

Alessandro Orso and Mauro Pezzè
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Phone: +39 2 2399 3638
[orso|pezze]@elet.polimi.it

Abstract

Object oriented features like information hiding, inheritance, polymorphism, and dynamic binding, present new problems, that cannot be adequately solved with traditional testing approaches. In this paper we address the problem of integration testing of procedural object oriented systems in the presence of polymorphism. The kind of polymorphism addressed is *inclusion polymorphism*, as provided by languages like C++, Eiffel, CLOS, Ada95, and Java.

This paper proposes a technique for testing combinations of polymorphic calls. The technique applies a data-flow approach to newly defined sets of *polymorphic definitions and uses*, i.e., definitions and uses that depend on polymorphic calls. The proposed testing criteria allow for selecting execution paths that can reveal failures due to incorrect combinations of polymorphic calls. The paper describes the technique and illustrates its efficacy through an example.

Keywords: testing object oriented programs with inclusion polymorphism, data-flow testing, integration testing, test data selection criteria.

1 Introduction

Object oriented systems present new challenges for testing, due to new features and different structure of object oriented programs [13]. Traditional approaches break down in the face of features like information hiding, inheritance, polymorphism, and dynamic binding. The problem of testing object oriented software has been investigated only recently. Some papers present methods and techniques to address specific problems, but many issues are still open. In particular, the problem of testing the effects of dynamic bindings of polymorphic calls has been investigated only in a few papers [7, 10, 15], that mainly concentrate on selecting test cases for testing isolated calls. These techniques, although extremely useful for testing polymorphic calls in isolation, do not address an important class of failures, i.e., failures related to interactions between different polymorphic invocations.

This paper concentrates on this latter problem: the selection of adequate test cases for testing combinations of polymorphic calls during integration testing. The kind of polymorphism addressed is *inclusion polymorphism*, as provided by languages like C++, Eiffel, CLOS, Ada95 and Java. The paper assumes a traditional bottom-up integration testing strategy, where a class A is integrated with all classes containing methods that can be bound to calls occurring in class A itself. This paper identifies an important class of failures that can derive from the use of polymorphic calls in object oriented programs. The considered failures are due to the combined effects of different polymorphic invocations along specific execution paths. Such failures occur during integration testing and can remain uncaught using currently available testing techniques. The technique proposed in this paper allows for selecting more accurate test cases aiming at revealing the identified class of failures. The paper shows that traditional data-flow test selection criteria [2] can be straightforwardly extended by suitably defining new *def* and *use* sets that take into account the presence of dynamic binding. The possibility of easily extending traditional criteria allows for applying a well known body of knowledge to the new problem

*This work has been partially supported by ESPRIT Project TWO (Test Warning Office - EP n.28940) and by Italian Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project.

and the easy combination of new and traditional coverage criteria. The next section briefly discusses the problems arising in the presence of polymorphic calls that can be dynamically bound to different methods. The following section surveys the main techniques proposed so far for testing object oriented programs in the presence of polymorphism. Section 4 introduces the new approach: it first describes Inter Class Control Flow Graphs (ICCFGs), the program representation used to define the technique; then, it illustrates the proposed technique referring to a simple example; finally, it introduces a few definitions, and discusses the application of different path selection criteria. The last two sections discuss the technique with the help of an example and conclude recalling the main results achieved so far.

2 Polymorphism and Testing

In object oriented programming, the term *polymorphism* indicates the possibility for the same entity to dynamically refer to instances of different classes. With typed languages, like the ones considered in this paper, polymorphism is constrained by inheritance [11], i.e., types can be substituted only within a type or a class hierarchy. For example, in Java a reference to an object of type `A` can be bound to an object of any type `B`, as long as `B` is a sub-type of `A` (including `A` itself). The type `A` provided in the declaration is called *static type*; the type of the object `B` bound during an execution is called *dynamic type*. This kind of polymorphism is referred to as inclusion polymorphism [1] or, more specifically, as subclass polymorphism. The invocation of a method on a polymorphic entity can result in different bindings depending on the dynamic type of the entity. The possibility of binding different methods at run time is called *dynamic binding*. With dynamic binding, the actual object that processes a message, and thus the method actually invoked, is not statically known, but it is an instance of one of a finite number of classes. In the following, we use the terms *virtual method* (borrowed from C++) or *polymorphic method* to denote a method defined in a class `A` and redefined in some of the subclasses of `A`. Analogously, we use the terms *virtual invocation* or *polymorphic invocation* to denote the invocation of a virtual (polymorphic) method.

Testing programs in the presence of inclusion polymorphism and dynamic binding presents new problems due to the infeasibility of statically identifying actual bindings. Exhaustive testing of all possible combinations of bindings may be impractical, and thus a technique is needed, which allows for selecting adequate test cases. The work surveyed in the next section provides interesting techniques that address the problem of testing polymorphic calls in isolation. As for traditional programs, most failures are not caused by a single invocation, but by the combined effects of different invocations along an execution path. Such failures can remain uncaught while focusing on isolated calls. In the case of polymorphic invocations, it is important to be able to select paths according to the combinations of contained invocations and corresponding bindings to adequately test their combined effects. For example, let class `Person`, with a public method `height`, be specialized into classes `Woman` and `Man`, both redefining the method `height`. Let us assume that, for some error in the internationalization of the code, method `height` in class `Man` returns the value in inches, while method `height` in class `Women` returns the value in centimeters. Testing the two polymorphic invocations in the fragment of code shown in Figure 1 independently, would not reveal the trivial problem derived from comparing inches and centimeters. An adequate test must consider combinations of invocations and corresponding bindings along execution paths.

```
1. Person p1, p2;
   ...
2. int h1=p1.height();
   ...
3. int h2=p2.height();
   ...
4. if(h1 < h2) ...
```

Figure 1: Faulty polymorphic invocations in Java

3 Related Work

The general problems of testing object oriented systems are addressed in several papers. However, the specific problem of testing programs with polymorphic references is discussed in some details only in a few papers. Here we survey the main proposals for inclusion polymorphism. Proposals for different kinds of polymorphism are not reviewed since they are not directly related with the work described in this paper.

Kirani and Tsai [7] propose a technique for generating test cases from functional specification for module and integration testing of object oriented systems. The method generates test cases that exercise specific combinations of method invocations. The method addresses object oriented testing in general, but is not specifically designed for coping with polymorphism and dynamic binding. In particular, it does not address the problem of selecting bindings for the polymorphic calls in the exercised combinations. A full solution of such problem would require analysis of the code, while Kirani and Tsai focus on functional testing.

McDaniel and McGregor [10] propose a technique for reducing the combinatorial explosion of the number of test cases for covering all combinations of polymorphic caller, callee, parameters, and related states. The technique is based on latin squares [9]: a set of specific orthogonal arrays are used to identify the subset of combinations of the state of each object and its dynamic type to be tested. The method ensures coverage of all pairwise combinations. It applies to single calls but does not consider the combined effects of different calls, as required in integration testing.

Paradkar [15] proposes a pairwise integration strategy based on the relationships among classes, and a heuristic method for selecting test cases based on states of objects. The heuristic method allows for identifying some infeasible combinations and thus limits the number of test cases for integration testing, focusing on the integration order, but not on the combination of different bindings.

4 A Data-Flow Technique for Testing Polymorphism

Programs that contain polymorphic invocations may fail due to specific combinations of dynamic bindings that occur along an execution path, and behave correctly for different combinations of dynamic bindings for the same path. To adequately test such programs, we need selection criteria that identify paths differing only for the specific combinations of dynamic bindings. Traditional data-flow test selection criteria distinguish paths that differ for occurrences of definitions and uses of the same variables, but do not take into account the possibility that such definitions or uses may depend on invocations of different methods dynamically bound. In this section we propose a new data-flow test selection technique that distinguish different combinations of dynamic bindings for the same paths.

Before describing the new technique, we briefly illustrate the mechanism referring to the example of Figure 1. Let us assume that the fragment of code shown in the figure is part of a large program, comprising different complex paths. A test selection criterion able to reveal the failure due to the different units of measure must generate test data that exercise a path containing the nodes representing statements 2, 3, and 4, but this is not enough. The fault is revealed only if at least one of these test cases corresponds to different bindings of the polymorphic invocations that occur at nodes 2 and 3, e.g., a test case that causes the polymorphic invocation at line 2 to be dynamically bound to method `height` of class `Man` and the invocation at line 3 to be bound to method `height` of class `Woman`. Traditional data-flow testing criteria do not distinguish among different bindings, and thus cannot generate the required test data. They could success in revealing the failure only by chance. The technique proposed in this paper overcomes this problem by defining new sets def^p and use^p, that contain variables together with the polymorphic methods that can be "directly" or "indirectly" responsible for their definition or use.

In the example of Figure 1, the node representing the statement at line 2 would be associated with a def^p set containing the two pairs $\langle h1, Man.height\rangle$ and $\langle h1, Woman.height\rangle$, that reflect the different methods that can be dynamically responsible for the definition of variable $h1$. Analogously, the node corresponding to line 3 would be associated with a def^p set containing the two pairs $\langle h2, Man.height\rangle$ and $\langle h2, Woman.height\rangle$. The node representing the statement at line 4 would be associated with a use^p set containing the four pairs $\langle h1, Man.height\rangle$, $\langle h1, Woman.height\rangle$, $\langle h2, Man.height\rangle$, and $\langle h2, Woman.height\rangle$. This latter case is less intuitive than the former ones. As explained in detail in Section 4.2, in this case the set use^p capture how the result of the computation could depend on the invocation of different polymorphic methods, either directly or through intermediate variables.

The new sets allow for easily adapting traditional data-flow test selection criteria to cover paths differing only for the specific combinations of dynamic bindings. The new testing criteria require the coverage of different combinations of elements of the defined sets, and thus different combinations of bindings of polymorphic methods. In particular, any of the criteria described later in this paper would select test data required to reveal the failure of the program in Figure 1. Notice that the simple unfolding of all polymorphic calls would not lead to the same result, since it would not distinguish between polymorphic and non-polymorphic definitions and uses. The simple unfolding would be equivalent to the integrated criteria mentioned in Section 4.3. Although integrated criteria are extremely appealing, they are more expensive, while the criteria derived from the new sets represent interesting intermediate cases, that allow for identifying smaller set of notable test cases. Considering only polymorphically related definitions and uses allows for focusing on the class of failures identified in this paper.

This section starts recalling the definition of Inter Class Control Flow Graph (ICCFG), used as a reference model for the software to be tested; it then introduces the new concepts of *polymorphic definition* and *polymorphic use*; it finally describes

```
 class A {
 public void m1() {...};
 public void m2() {...};
};
 class B extends A {
 public void m1() {...};
};
 class C {
 private A refToA;
 private A a;
 public C() {refToA=null;
             a=new A;}
 public void setA(A a) {refToA=a;}
 public void m() {
   if(refToA != null)
     refToA.m1();
   ...
   a.m2();
   ...
   return;
  }
};
```

Figure 2: A fragment of Java code

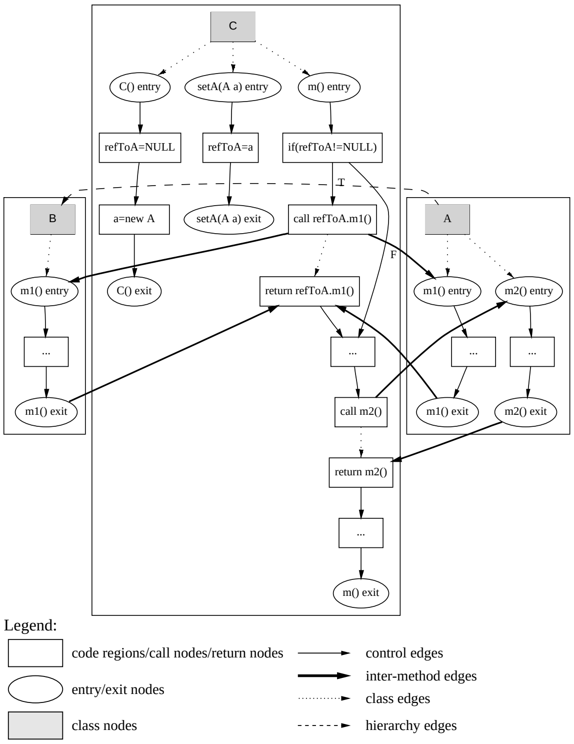

Figure 3: The ICCFG for the program of Figure 2

how traditional data-flow selection criteria can be adapted to the new sets, and discusses problems of infeasible paths, scalability, and complementarities with traditional data-flow test selection criteria.

4.1 ICCFGs

Traditional data-flow test selection criteria are defined starting from a control flow graph representation of the program to be tested. In this paper, we refer to a simplified version of the Inter Class Control Flow Graph (ICCFG) [3], that extends interprocedural control flow graphs to the case of object oriented programs. To avoid language dependent assumptions, for presenting the technique we consider a Java-like programming language, that provides a subset of Java constructs, namely, basic control constructs, inheritance, polymorphism and dynamic binding. As an example, Figure 2 shows a simple Java-like program, and Figure 3 shows the corresponding ICCFG. In ICCFGs, each method is represented by a Control Flow Graph (CFG). Nodes represent single-entry, single-exit regions of executable code. Edges represent possible execution branches between code regions. Each CFG corresponding to a method has an *entry node* and an *exit node*, both labeled with the name of the method. Classes are represented with *class nodes*, labeled with the name of the class. *Class nodes* are connected to the *entry nodes* of all the methods of the class with *class edges*. The hierarchy relation among classes is represented with *hierarchy edges* between *class nodes*.

Each Method invocation is represented with a *call node* and a *return node*, suitably connected to the *entry node* and the *exit node* of the invoked method with *inter-method edges*. Each non-polymorphic call corresponds to a single pair of inter-method edges. Each polymorphic call corresponds to a set of pairs of inter-method edges, one for each method in the *binding set*, i.e., the set composed by all the possible dynamic bindings of the invoked method. In both cases, inter-method edges must be properly paired in a path, i.e., when exiting a method, we need to follow the edge corresponding the considered invocation. Paths comprising only properly paired inter-method edges are traditionally called *valid paths*. Hereafter, we refer only to valid paths.

In the general case, a superset of the binding set can be statically constructed as follows: if `A` is the static type of the object the method `m1` is invoked upon, then we add to the approximated binding set `A.m1` and all the methods overriding `A.m1` in `A`'s subclasses. In the example shown in Figure 2, the approximated binding set constructed in this way for the call "`refToA.m1()`" would contain `A.m1` and `B.m1`. This simple algorithm can include many infeasible bindings. More accurate approximations of the binding set can be constructed by applying several methods, most of which corresponding to polynomial algorithms [8, 14, 5, 16, 6]. However, the general problem of identifying the exact binding set is undecidable. The algorithms proposed so far work for special cases and in general can determine an approximation, not the exact set. As discussed later in this section, the determination of a good approximation of the binding set can greatly alleviate the problem of infeasible paths, but does not solve the problem addressed in this paper, namely the identification of a reasonable set of test cases for exercising relevant combinations of bindings occurring along execution paths.

A *path* in the ICCFG is a finite sequence of nodes (n_1, n_2, ..., n_k), with $k \geq 2$, such that there is an edge from n_i to n_{i+1} for i= 1, 2, ..., k-1. A path is *loop-free* if all occurring nodes are distinct. A path is *complete* if its first node is the *entry* node of a method, and its last node is the *exit* node of the same method.

We model statements containing more than one polymorphic invocation with several nodes, to have at most one polymorphic invocation per node. For example, the statement "`if(p.m() < q.m())`", where both invocations are polymorphic, is modeled with two nodes corresponding to statements "`tmpvar = p.m()`" and "`if(tmpvar < q.m())`", respectively. Without loss of genericity, we assume that code regions contain at most one polymorphic call and only mutually related definitions and uses, i.e., if a variable v_1 belongs to the set *use(n)*, and a variable v_2 belongs to the set *def(n)*, then v_1 contributes to the definition of v_2.

4.2 Polymorphic Definitions and Uses

The data-flow testing technique proposed in this paper aims at identifying paths containing polymorphic invocations whose combination may lead to incorrect results. As stated above, incorrect behaviors may depend on the specific bindings of the invocations along the execution paths. Traditional *def(n)* and *use(n)* sets do not distinguish among different bindings, and thus they do not provide enough information for our goal. To meet the goal, we annotate nodes of the ICCFG graph with the new sets $def^p(n)$ and $use^p(n)$, that provide the required information. Sets $def^p(n)$ and $use^p(n)$ contain only variables defined or used as a consequence of a polymorphic invocation. Variables in the sets $def^p(n)$ and $use^p(n)$ are paired with the polymorphically invoked method responsible for their definition or use, respectively. The same variable often occurs in several pairs of the same $def^p(n)$ or $use^p(n)$ set, since it can be defined or used as a consequence of the polymorphic invocations of different methods. These definitions of sets $def^p(n)$ and $use^p(n)$ allow for easily adapting traditional data-flow test selection criteria to the case of programs containing polymorphic invocations. The obtained criteria distinguish among paths that differ for the polymorphically invoked methods responsible for the definitions and uses of the same variable. Thus, they can identify paths containing different polymorphic invocations whose combination may lead to incorrect results. In this section, we introduce sets $def^p(n)$ and $use^p(n)$. Test selection criteria are discussed in the next section.

Let us assume that each node n of the ICCFG graph is annotated with the traditional sets *def(n)* and *use(n)* [17]. Sets *def(n)* contain all the variables which are bound to a new value as a consequence of the execution of the code region modeled with node n. Sets *use(n)* contain all the variables whose values are used by the code region modeled with node n. A *def-clear* path with respect to a variable v is a path (n_1, n_2, ..., n_k) such that $v \notin def(n)$ for $n = n_2, n_3, \ldots, n_{k-1}$.

Each node n of the ICCFG is associated with two additional sets $def^p(n)$ and $use^p(n)$. Sets $def^p(n)$ contain pairs composed of a variable name and a method name. In this paper we assume that names uniquely identify the corresponding elements, i.e., are disambiguated by prefixing the name of the class they occur in, when needed. A pair $\langle v, m\rangle$ belongs to set $def^p(n)$ if variable v is either *directly* or *indirectly* defined by virtual method m at node n. A variable v is *directly* defined by a virtual method m at node n if the statement that defines variable v contains an invocation that can be dynamically bound to method m. In this case, the polymorphic invocation is directly responsible for the computation of the new value of variable v. A variable v is *indirectly* defined by a virtual method m at node n if a variable w that contributes to define variable v is *directly* or *indirectly* defined by virtual method m at a node p, and there exists a def-clear path from node p to node n with respect to w. In this case there exists a chain of related definitions and uses from a polymorphic definition of a variable w_1 to the definition of variable v. More specifically, the polymorphic invocation of method m is directly responsible for the computation of the new value of a variable w_1; such value may be used to define the value of a variable w_2, and so on; a path of uses and definitions leads to the definition of variable w, whose value is used to compute the new value of variable v. Such a path, that can be arbitrarily long, cannot contain additional definitions of one of the involved variables, i.e., each sub-path from the definition of w_i to its use to define w_{i+1} is a def-clear path with respect to w_i.

Similarly, sets $use^p(n)$ contain pairs composed of a variable name and a method name. A pair $\langle v, m\rangle$ belongs to the set $use^p(n)$ if variable v is used in either *direct* or *indirect* relation with a virtual method m. A variable v is used in *direct* relation with the virtual method m at node n, if it is used in a statement that contains an invocation that can be dynamically bound to method m. In this case, the result of the computation depends on the combination of the value of variable v and the results of the polymorphic invocation. A variable v is used in *indirect* relation with the virtual method m at node n if it is used in a statement that uses a variable w, variable w is *directly* or *indirectly* defined by virtual method m at a node p, and there exists a def-clear path from node p to node n with respect to w. In this case the result of the computation depends on the combination of the value of variable v and the value of variable w, whose definition depends on a polymorphic invocation. Intuitively, the result of the computation depends on the combination of the value of variable v and the results of the polymorphic invocation through the chain of definitions and uses that determine the indirect polymorphic definition of variable w. In general, the concepts of indirect definitions and uses avoid loss of information caused by the use of intermediate variables between different polymorphic invocations.

Examples of variables *directly* and *indirectly* polymorphically defined or used by virtual methods are given in Figure 4.

```
1. k=9;
2. y=0;
3. x=polRef.m()+k;
4. do {
5.     z=y;
6.     y=x*2;
7. } while(z < w);
```

Figure 4: Examples of direct and indirect polymorphic definitions and uses

direct polymorphic definition: Variable x is polymorphically directly defined by all methods m_1, ... m_n, that can be dynamically bound to the invocation *polRef.m()* at statement 3. Thus, pairs $\langle x, m_1\rangle \ldots \langle x, m_n\rangle$ belong to the set *def^p(3)* associated to statement 3.

indirect polymorphic definition: Variable y is polymorphically indirectly defined at statement 6 by all methods m_1, ... m_n that can be dynamically bound to the invocation *polRef.m()* at statement 3, since y is defined using x at statement 6; x is polymorphically defined at statement 3; and there exists a def-clear path with respect to x from statement 3 to statement 6 ((3, 4, 5, 6)). Thus, pairs $\langle y, m_1\rangle \ldots \langle y, m_n\rangle$ belong to the set *def^p(6)* associated to statement 6.
Variable z is polymorphically indirectly defined at statement 5 by all methods m_1, ... m_n that can be dynamically bound to the invocation *polRef.m()* at statement 3, since z is defined using y at node 5; y is polymorphically defined at node 6; and there exists a def-clear path with respect to y from node 6 to node 5 ((6, 7, 4, 5)). Thus, pairs $\langle z, m_1\rangle \ldots \langle z, m_n\rangle$ belong to the set *def^p(5)* associated to statement 5.

direct polymorphic use: Variable k is polymorphically directly used by all methods m_1, ... m_n that can be dynamically bound to the invocation *polRef.m()* at statement 3, since k is used in an expression comprising such polymorphic call. Thus, pairs $\langle k, m_1\rangle \ldots \langle k, m_n\rangle$ belong to the set *def^p(3)* associated to statement 3.

indirect polymorphic use: Variable w is polymorphically indirectly used at statement 7 by all methods m_1, ... m_n that can be dynamically bound to the invocation *polRef.m()* at statement 3, since w is used in an expression that also uses z; z is polymorphically defined at statement 5; and there exists a def-clear path with respect to z from statement 5 to statement 7 ((5, 6, 7)). Thus, pairs $\langle w, m_1\rangle \ldots \langle w, m_n\rangle$ belong to the set *use^p(7)* associated to statement 7.

An algorithm for computing sets *def^p(n)* and *use^p(n)* is given in Figure 5. Since this paper is mostly focused on demonstrating the essentials of the proposed technique, we only consider alias-free programs and we focus on intramethod definition-use chains. We are currently working on extending the technique to the intermethod case by following an approach similar to the one presented by Harrold and Soffa [4]. The algorithm is applied to a subgraph of the ICCFG corresponding to the control flow graph of a single method. The algorithm assumes that the code regions associated to the nodes of the ICCFG contain only definitions and uses mutually related, as stated above.

The algorithm of Figure 5 is polynomial in the number of nodes. Here we indicate a rough approximation of its complexity; a formal and precise computation can be found in [12]. Loops (1), (9), (26) and (30) are executed once for each node. Loop (14) is executed as many times as the maximum number of nodes, since each iteration adds the information about indirectness relative to an additional level of ancestors (assuming a preorder traversal), and the number of ancestors of each node cannot exceed the total number of nodes. Each execution of loop (14) comprises a loop on all nodes (statement 17) which comprises a loop on the preset of each node (statement 19), whose cardinality does not exceed the total number of nodes in the worst case. Thus, we have a cubic factor in the worst case.

The traditional concepts of *du-path* and *du-set*[17] can be easily extended as follows. A *du-path^p* with respect to a variable v is a path (n_1, n_2, ..., n_k) such that $\langle v, m_1\rangle \in$ *def^p(n_1)*, $\langle v, m_2\rangle \in$ *use^p(n_k)* (for any virtual methods m_1 and m_2), and (n_1, n_2, ..., n_k) is a def-clear path with respect to

Input:

$G = (N, E, n_0)$: a subgraph of the ICCFG, corresponding to the control flow graph of a method annotated with code regions corresponding to each node.

Output:

$DEF^p(G)$: set of $def^p(n)$, one for each node $n \in N$.
$USE^p(G)$: set of $use^p(n)$, one for each node $n \in N$.

for all nodes $n \in N$ **do**
  /* build the sets $def(n)$ and $use(n)$ */
  $def(n) = \{v | v$ is a variable defined by a statement occurring in the code region of node $n\}$
  $use(n) = \{v | v$ is a variable used by a statement occurring in the code region of node $n\}$
  /* build the initial sets $def^p(n)$ and $use^p(n)$ considering only direct definitions and uses */
  $def^p(n) = \{\langle v, m\rangle | v$ is a variable defined by a statement occurring in the code region of node n and m is a virtual method which can be dynamically bound to an invocation occurring in the same node$\}$
  $use^p(n) = \{\langle v, m\rangle | v$ is a variable used by a statement occurring in the code region of node n and m is a virtual method which can be dynamically bound to an invocation occurring in the same node$\}$
end for
for all nodes $n \in N$ **do**
  /* build the initial sets $avail^p(n)$ from the initial sets $def^p(n)$, thus considering only direct definitions */
  $avail^p(n) = \bigcup_{k \in preset(n)} \{def^p(k)\}$
end for
$changed = true$
while $changed$ **do**
  /* build the sets $avail^p(n)$ by incrementally adding pairs $\langle v, m\rangle$ such that either one of the following conditions holds:
  1) $\langle v, m\rangle$ belongs to the set $avail^p(k)$, being k an immediate predecessor of node n, and variable v is not defined by any statement occurring in the code region of node k
  2) variable v is defined in a non-polymorphic way by a statement occurring in the code region of node n which uses variable v_1, and the pair $\langle v_1, m\rangle$ belongs to $avail^p(k)$, of an immediate predecessor of node n.
  Terminate when the last iteration does not modify any of the sets.
  $preset(n)$ indicates the immediate predecessors of node n. */
  $changed = false$
  for all nodes $n \in N$ **do**
    $old = avail^p(n)$
    $avail^p(n) = avail^p(n) \cup \{\langle v, m\rangle | \exists k \in preset(n)(\langle v, m\rangle \in avail^p(k) \wedge v \notin def(k))\}$
      $\cup \{\langle v, m\rangle | v \in def(n) \wedge \nexists m_1(\langle v, m_1\rangle \in def^p(n)) \wedge \exists v_1 \in use(n) \wedge \exists k \in preset(n)(\langle u, m\rangle \in avail^p(k))\}$
    if $avail^p(n) <> old$ **then**
      $changed = true$
    end if
  end for
end while
for all nodes $n \in N$ **do**
  /* build the complete sets $def^p(n)$ starting from sets $avail^p(n)$, thus considering also indirect polymorphic definitions; a pair $\langle v, m\rangle$ is added to the set $def^p(n)$ if variable v is defined in a non-polymorphic way by a statement occurring in the code region of node n, and the pair $\langle v, m\rangle$ belongs to $avail^p(n)$ */
  $def^p(n) = def^p(n) \cup \{\langle v, m\rangle | v \in def(n) \wedge \nexists m_1(\langle v, m_1\rangle \in def^p(n)) \wedge \langle v, m\rangle \in avail^p(n)\}$
end for
for all nodes $n \in N$ **do**
  /* build the complete sets $use^p(n)$ starting from sets $avail^p(n)$, thus considering also indirect polymorphic uses; a pair $\langle v, m\rangle$ is added to the set $def^p(n)$ if variable v is used in a non-polymorphic way by a statement occurring in the code region of node n in conjunction with the use of a variable v_1, and the pair $\langle v_1, m\rangle$ belongs to $avail^p(n)$ */
  $use^p(n) = use^p(n) \cup \{\langle v, m\rangle | v \in use(n) \wedge \nexists m_1(\langle v, m_1\rangle \in use^p(n)) \wedge \exists v_1(\langle v_1, m\rangle \in avail^p(n) \wedge v_1 \in use(n))\}$
end for

Figure 5: An algorithm for computing sets $def^p(n)$ and $use^p(n)$.

v. A polymorphic du-set, *du^p(v,n)*, for a variable v and a node n is the set of nodes i such that there exists a *du-path* from node n to node i.

4.3 Path Selection Criteria

Starting from the data-flow information associated with the ICCFG, it is possible to define a family of test adequacy criteria for exercising polymorphic interactions among classes by extending traditional data-flow selection criteria [17]. The extensions consider the differences between the traditional sets *def(n)* and *use(n)* and the newly defined sets *def^p(n)* and *use^p(n)*. Traditional data-flow selection criteria only require given nodes to be traversed according to given sequences of definitions and uses. New criteria take into account also the dynamic type of the polymorphic references occurring in the paths, i.e., they indicate which dynamic bindings must be exercised. To formalize this principle, we introduce the concepts of *polymorphic coverage* (*p-coverage*) and *coverage of polymorphic uses* (*u-coverage*). Given a node n, a pair $\langle v,m\rangle$ in *def^p(n)* (resp. in *use^p(n)*), and a path q that includes node n, an execution of path q p-covers the pair $\langle v,m\rangle$ for node n if the definition (resp. the use) of variable v at node n depends (either directly or indirectly) on the polymorphic invocation of method m in the considered execution of path q. Informally, the execution of path q *p-covers* the pair $\langle v,m\rangle$ at node n if the virtual invocation that defines (resp. uses) variable v is dynamically bound to method m while executing path q.

Given a node n, a pair $\langle v,m\rangle$ in *use^p(n)*, and a set P of paths that include node n, the set P *u-covers* v for n if for each pair $\langle v,m_1\rangle \in$ *use^p(n)*, there exists a path $q \in P$ whose execution p-covers $\langle v,m_1\rangle$ for n. Informally, a set of paths P *u-covers* variable v for node n if the executions of the paths in P *p-cover* all pairs containing v in *use^p(n)*. In the following we also use the expression "a path *p-covers* a pair $\langle v,m\rangle$" to indicate that an execution of the path p-covers such pair. Extended criteria require not only the traversal of specific paths, but also that the executions of such paths p-cover specific pairs.

Most traditional data-flow test selection criteria can be extended to the polymorphic case. To illustrate the technique, we present the *all-defs*, *all-uses*, and *all-du-paths* criteria extended for polymorphism. A complete set of criteria can be found in [12].

Given an ICCFG and a method m, let T be a set of test cases corresponding to executions of the set of complete paths P for the control flow graph G of a method:

T satisfies the *all-defs^p* criterion if for every node n belonging to G and every pair $\langle v,m\rangle \in$ *def^p(n)*, at least one path in P p-covers $\langle v,m\rangle$ for n and the set P u-covers v for at least one node $n_1 \in$ *du^p(v,n)*. Intuitively, for each polymorphic definition of each variable, the *all-defs^p* criterion exercises all possible bindings for at least one polymorphic use. It naturally extends the traditional *all-defs* criterion by requiring the execution of all bindings for the chosen use.

T satisfies the *all-uses^p* criterion if for every node n belonging to G and every pair $\langle v,m\rangle \in$ *def^p(n)*, at least one path in P p-covers $\langle v,m\rangle$ for n and the set P u-covers v for all nodes $n_1 \in$ *du^p(v,n)*.
Intuitively, the *all-uses^p* criterion subsumes the *all-defs^p* criterion by extending the coverage to all polymorphic uses of each polymorphic definition, exactly like the traditional *all-uses* criterion subsumes the *all-defs* criterion.

T satisfies the *all-du-paths^p* criterion if for every node n belonging to G, for every pair $\langle v,m\rangle \in$ *def^p(n)*, and for every node $n_1 \in du^p(v,n)$, at least one path in P p-covers $\langle v,m\rangle$ for n and P u-covers v for n_1 along all possible def-clear paths with respect to v.
Intuitively, the *all-du-paths^p* criterion subsumes the *all-uses^p* criterion, by requiring the selection of all def-clear paths from each polymorphic definition to each corresponding polymorphic uses for all possible bindings.

Any of the defined criteria applied to the simple example of the fragment of code shown in Figure 1, would require the two possible binding of the virtual method `height` to be exercised in combination. Thus, for this simple example, any of the proposed methods would reveal the trivial failure of the program, that may remain uncaught with other approaches that focus on single polymorphic calls.

The criteria proposed in this paper add a new dimension, namely the dynamic bindings, to the traditional dimensions of definitions and uses. Ignoring the different bindings, the new criteria do not differ from the corresponding traditional criteria projected on the variables involved in polymorphic definitions and uses. Integrated approaches can be straightforwardly defined by applying a traditional criterion to all variables, and extending the coverage of variables involved in polymorphic definitions and uses referring to the corresponding new criterion. "Hybrid" criteria can be straightforwardly defined by introducing new criteria that refer to a mixture of traditional and polymorphic def and use sets. For example, by requiring the coverage of all polymorphic uses of each traditional definition. A detailed discussion of integrated and hybrid criteria can be found in [12].

4.4 Feasibility of the Approach

The impossibility of determining the feasibility of execution paths and dynamic bindings causes problems similar to the ones experienced in traditional approaches, namely, the impossibility of determining the exact coverage. Infeasible execution paths affect the new criteria as the traditional criteria, since polymorphism does not modify the set of feasible paths. Infeasible dynamic bindings create new problems, that depend on the approximation of the computed binding sets. The simple algorithm sketched in Section 4.1 can identify many infeasible bindings that can greatly reduce the effectiveness of the approach. However, a careful choice of an appropriate methods for computing the binding set, e.g., one of the methods cited in Section 4.1, can greatly reduce the problem. As in the traditional case, the problem of infeasible paths depends on the chosen criterion: we did not notice any notable change with respect to the traditional case when using simple criteria, such as the *all-def*^{p} criterion; the infeasibility problem can become heavier with more sophisticated criteria, like the *all-du-paths*^{p}. The experiments conducted so far on well designed programs did not reveal a notable increase of infeasible paths due to bad approximations of the binding sets, computed with an appropriate method.

5 Example

In this section we present the application of the proposed approach to an example: a set of four Java classes: `Polygon`, `Circle`, `Square`, and `Figure`. Figure 6 shows the skeleton of the Java code of the example. The statements composing the method `addPolygon` have been numbered and edited to have at most one polymorphic call per line of code, thus simplifying the correspondence between the code itself and the partial ICCFG shown in Figure 7. Class `Figure` is a container of objects of type `Polygon`, hierarchically specialized as `Circle` and `Square`. In the example, class `Figure` can contain up to two polygons. We are interested in computing the area of objects of type `Figure` starting from the areas of the contained objects. Contained objects of type `Polygon` have a sign determining whether their area has to be computed as positive or negative.

Figures can be built by adding `Polygons` with the following rules: a polygon cannot intersect previously inserted polygons; if the inserted polygon is completely contained in a previously inserted polygon, then its sign becomes negative; if the inserted polygon completely contains a previously inserted polygon, then the sign of the contained polygon becomes negative. Classes `Square` and `Circle` are implemented as subclasses of `Polygon`. Class `Polygon` is an abstract class which defines the concrete methods `getX`, `getY`, `setX`, `setY`, `setSign`, `getSign`, and `area`, which are inherited unchanged by both classes `Square` and `Circle`. Class `Polygon` declares

```
 class Figure {
  ...
  public boolean addPolygon(Polygon p) {
1   if(poly1==null) {
2     poly1=p;
3     return true; }
4   else if(poly2 != null) return false;
5   else {
6     int pminX=p.minX();
7     int pmaxX=p.maxX();
8     int pminY=p.minY();
9     int pmaxY=p.maxY();
10    if(pminX < poly1.minX()) {
11      if(pmaxX > poly1.maxX()) {
12        if(pminY < poly1.minY()) {
13          if(pmaxY > poly1.maxY()) {
14            poly1.setSign(1);
15            poly2=p;
16            return true; }}}}
17    if(pminX > poly1.minX()) {
18      if(pmaxX < poly1.maxX()) {
19        if(pminY > poly1.minY()) {
20          if(pmaxY < poly1.maxY()) {
21            p.setSign(-1);
22            poly2=p;
23            return true; }}}}
24    if(!(pminX>poly1.maxX())) {
25      if(!(pmaxX<poly1.minX())) {
26        if(!(pminY>poly1.maxY())) {
27          if(!(pmaxY<poly1.minY())) {
28            return false; }}}}
29    poly2=p;
30    return true; }
  }
  public double area() {
    double a=0;
    if(poly2 != null) a+= poly2.area();
    if(poly1 != null) a+= poly1.area();
    return a;
  }
}
abstract  class Polygon {
  ...
  protected abstract double unsignedArea();
  public abstract int minX();
  public abstract int minY();
  public abstract int maxX();
  public abstract int maxY();
  ...
  final public int getX() {return x;}
  final public int getY() {return y;}
  final public void setX(int xx) {x=xx;}
  final public void setY(int yy) {y=yy;}
  final public void setSign(int s) {...}
  final public int getSign() {...}
  final public double area() {...}
}
 class Circle extends Polygon {
  ...
  protected double unsignedArea()
         {return (3.14*radius*radius);}
  public int minX() {return (getX()-radius/2);}
  public int maxX() return (getX()+radius/2);
  public int minY() return (getY()-radius/2);
  public int maxY() return (getY()+radius/2);
  ...
}
 class Square extends Polygon {
  ...
  protected double unsignedArea()
         {return (edge*edge);}
  public int minX() {return (getX()-edge/2);}
  public int minY() return (getY()-edge/2);
  public int maxX() return (getX()+edge/2);
  public int maxY() return (getY()+edge/2);
  ...
}
```

Figure 6: The Java skeleton of the sample program

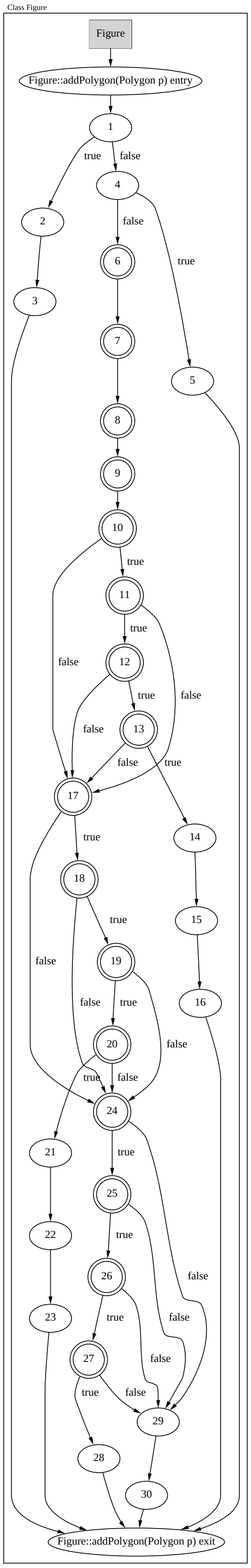

Figure 7: A partial ICCFG for the example

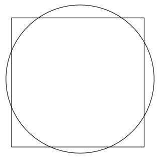

Figure 8: An example in which the *contain* relation would be computed incorrectly

Node	**def^p(n)**
6	<pminX,Circle.minX>, <pminX,Square.minX>
7	<pmaxX,Circle.maxX>, <pmaxX,Square.maxX>
8	<pminY,Circle.minY>, <pminY,Square.minY>
9	<pmaxY,Circle.maxY>, <pmaxY,Square.maxY>
Node	**use^p(n)**
10	<pminX,Circle.minX>, <pminX,Square.minX>
11	<pmaxX,Circle.maxX>, <pmaxX,Square.maxX>
12	<pminY,Circle.minY>, <pminY,Square.minY>
13	<pmaxY,Circle.maxY>, <pmaxY,Square.maxY>
17	<pminX,Circle.minX>, <pminX,Square.minX>
18	<pmaxX,Circle.maxX>, <pmaxX,Square.maxX>
19	<pminY,Circle.minY>, <pminY,Square.minY>
20	<pmaxY,Circle.maxY>, <pmaxY,Square.maxY>
24	<pminX,Circle.maxX>, <pminX,Square.maxX>
25	<pmaxX,Circle.minX>, <pmaxX,Square.minX>
26	<pminY,Circle.maxY>, <pminY,Square.maxY>
27	<pmaxY,Circle.minY>, <pmaxY,Square.minY>

Table 1: Sets *def^p(n)* and *use^p(n)* for the example

also the abstract methods `minX`, `minY`, `maxX`, `maxY`, and `unsignedArea`, which are defined in the two subclasses.

In the example, method `addPolygon` checks if a newly inserted polygon is enclosed in an existing one by comparing their cartesian coordinates, that are computed by the dynamically bound methods `minX`, `minY`, `maxX`, and `maxY`. Due to the way coordinates are computed, method `addPolygon` would erroneously consider cases like the one shown in Figure 8. This fault can be revealed only by suitable combining dynamic bindings of polymorphic invocations of methods `minX`, `minY`, `maxX`, and `maxY` in method `addPolygon`. Testing techniques dealing with single calls would not reveal such faults, since the fault is not due to a single invocation, but to the combined use of different dynamic bindings. In this section we show how the *all-du-paths^p* criterion used for integration testing of the four classes would select a sequence of dynamic bindings that could reveal the fault. In the following, we apply the technique to the method `addPolygon`, that contains the fault.

The subset of the ICCFG for method `addPolygon` of class `Figure` is shown in Figure 7. Nodes containing relevant polymorphic invocations are highlighted with double circles; nodes *call*, *return*, and relative inter-method edges are omitted. Table 1 shows the sets *def^p(n)* and *use^p(n)*, respectively, for all relevant nodes of the ICCFG.

	def	use
1	$<pminX, C.minX>$ (6)	$<pminX, C.minX>$ (10)
2	$<pminX, C.minX>$ (6)	$<pminX, S.minX>$ (10)
3	$<pminX, C.minX>$ (6)	$<pminX, C.minX>$ (17)
4	$<pminX, C.minX>$ (6)	$<pminX, S.minX>$ (17)
5	$<pminX, C.minX>$ (6)	$<pminX, C.maxX>$ (24)
6	$<pminX, C.minX>$ (6)	$<pminX, S.maxX>$ (24)
7	$<pminX, S.minX>$ (6)	$<pminX, C.minX>$ (10)
8	$<pminX, S.minX>$ (6)	$<pminX, S.minX>$ (10)
9	$<pminX, S.minX>$ (6)	$<pminX, C.minX>$ (17)
10	$<pminX, S.minX>$ (6)	$<pminX, S.minX>$ (17)
11	$<pminX, S.minX>$ (6)	$<pminX, C.maxX>$ (24)
12	$<pminX, S.minX>$ (6)	$<pminX, S.maxX>$ (24)
13	$<pmaxX, C.maxX>$ (7)	$<pminX, C.maxX>$ (11)
14	$<pmaxX, C.maxX>$ (7)	$<pminX, S.maxX>$ (11)
15	$<pmaxX, C.maxX>$ (7)	$<pminX, C.maxX>$ (18)
16	$<pmaxX, C.maxX>$ (7)	$<pminX, S.maxX>$ (18)
17	$<pmaxX, C.maxX>$ (7)	$<pminX, C.minX>$ (25)
18	$<pmaxX, C.maxX>$ (7)	$<pminX, S.minX>$ (25)
19	$<pmaxX, S.maxX>$ (7)	$<pminX, C.maxX>$ (11)
20	$<pmaxX, S.maxX>$ (7)	$<pminX, S.maxX>$ (11)
21	$<pmaxX, S.maxX>$ (7)	$<pminX, C.maxX>$ (18)
22	$<pmaxX, S.maxX>$ (7)	$<pminX, S.maxX>$ (18)
23	$<pmaxX, S.maxX>$ (7)	$<pminX, C.minX>$ (25)
24	$<pmaxX, S.maxX>$ (7)	$<pminX, S.minX>$ (25)
25	$<pminY, C.minY>$ (8)	$<pminY, C.minY>$ (12)
26	$<pminY, C.minY>$ (8)	$<pminY, S.minY>$ (12)
27	$<pminY, C.minY>$ (8)	$<pminY, C.minY>$ (19)
28	$<pminY, C.minY>$ (8)	$<pminY, S.minY>$ (19)
29	$<pminY, C.minY>$ (8)	$<pminY, C.maxY>$ (26)
30	$<pminY, C.minY>$ (8)	$<pminY, S.maxY>$ (26)
31	$<pminY, S.minY>$ (8)	$<pminY, C.minY>$ (12)
32	$<pminY, S.minY>$ (8)	$<pminY, S.minY>$ (12)
33	$<pminY, S.minY>$ (8)	$<pminY, C.minY>$ (19)
34	$<pminY, S.minY>$ (8)	$<pminY, S.minY>$ (19)
35	$<pminY, S.minY>$ (8)	$<pminY, C.maxY>$ (26)
36	$<pminY, S.minY>$ (8)	$<pminY, S.maxY>$ (26)
37	$<pmaxY, C.maxY>$ (9)	$<pminY, C.maxY>$ (13)
38	$<pmaxY, C.maxY>$ (9)	$<pminY, S.maxY>$ (13)
39	$<pmaxY, C.maxY>$ (9)	$<pminY, C.maxY>$ (20)
40	$<pmaxY, C.maxY>$ (9)	$<pminY, S.maxY>$ (20)
41	$<pmaxY, C.maxY>$ (9)	$<pminY, C.minY>$ (27)
42	$<pmaxY, C.maxY>$ (9)	$<pminY, S.minY>$ (27)
43	$<pmaxY, S.maxY>$ (9)	$<pminY, C.maxY>$ (13)
44	$<pmaxY, S.maxY>$ (9)	$<pminY, S.maxY>$ (13)
45	$<pmaxY, S.maxY>$ (9)	$<pminY, C.maxY>$ (20)
46	$<pmaxY, S.maxY>$ (9)	$<pminY, S.maxY>$ (20)
47	$<pmaxY, S.maxY>$ (9)	$<pminY, C.minY>$ (27)
48	$<pmaxY, S.maxY>$ (9)	$<pminY, S.minY>$ (27)

Table 2: Polymorphic *definition-use* pairs for the example (where *C.* stands for *Circle.*, and *S.* stands for *Square.*)

Table 2 pairs polymorphic definitions with related polymorphic uses on def-clear paths. Each line indicates a polymorphic definition as a pair $\langle variable, method \rangle$, the corresponding use, and the nodes of the ICCFG they are associated with.

The *all-du-paths*^{p} criterion requires at least a test case for each path covering all pairs shown in Table 2. Table 3 shows a possible set of (sub)paths covering all such pairs, and indicates the pairs covered by each (sub)path. Any set of test cases, that exercise a set of complete paths including the subpaths shown in Table 2 satisfy the *all-du-paths*^{p} criterion. In Table 3, when necessary, bindings for a node n belonging to a path are shown with the following syntax: "*ref*⇒`C`" (resp. `S`) states that the reference *ref* must be bound, in node n, to an object of type `Circle` (resp. `Square`), while *any* indicates that there are no constraints on the bindings for node n.

The paths selected by the *all-du-paths*^{p} criterion represent all combinations of possible dynamic bindings, including the ones leading to the described failure, i.e., paths covering the polymorphic definitions-use pairs of lines 9, 21, 33, 45 of Table 2. Tests selected according to the boundary values criteria for the given paths and bindings would reveal the fault.

6 Conclusions

Inclusion polymorphism and dynamic binding introduce new problems, as far as integration testing of procedural object oriented systems is concerned. In this paper we identify an important class of failures that can derive from the presence of polymorphic calls in object oriented programs. The considered failures are due to the combined effects of different polymorphic invocations along an execution path. Such failures occur during integration testing and can remain uncaught using the currently available techniques for testing in the presence of polymorphism, that focus on the testing of polymorphic calls in isolation.

In this paper we have presented a data-flow testing technique which allows for selecting more accurate test cases aiming at revealing the identified class of failures. The paper has shown that the data-flow test selection criteria for identifying such failures can be straightforwardly derived from traditional data-flow testing techniques by suitably defining a new kind of *def* and *use* sets. The possibility of easily extending traditional criteria allows for applying a well known body of knowledge to the new problem and the easy combination of new and traditional coverage criteria.

References

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

[2] P. Frankl and E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, SE-14(10):1483–1498, Oct. 1988.

[3] M. J. Harrold and G. Rothermel. A coherent family of analyzable graph representations for object-oriented software. Technical Report OSU-CISRC-11/96-TR60, The Ohio State University, November 1996.

Path	**du pairs**
6(p⇒C),7(p⇒C),8(p⇒C),9(p⇒C),10(poly1⇒C), 11(poly1⇒C),12(poly1⇒C),13(poly1⇒C),14,15,16	1, 13,25,37
6(p⇒C),7(p⇒C),8(p⇒C),9(p⇒C), 10(poly1⇒S),11(poly1⇒S),12(poly1⇒S), 13(poly1⇒S),14,15,16	2,14,26,38
6(p⇒S), 7(p⇒S),8(p⇒S),9(p⇒S),10(poly1⇒C),11(poly1⇒C), 12(poly1⇒C),13(poly1⇒C),14,15,16	7,19,31,43
6(p⇒S),7(p⇒S),8(p⇒S),9(p⇒S),10(poly1⇒S), 11(poly1⇒S),12(poly1⇒S),13(poly1⇒S),14,15,16	8, 20,32,44
6(p⇒C),7(p⇒C),8(p⇒C),9(p⇒C),10(any),17(poly1⇒C), 18(poly1⇒C),19(poly1⇒C),20(poly1⇒C),21,22,23	3, 15,27,39
6(p⇒C),7(p⇒C),8(p⇒C),9(p⇒C),10(any), 17(poly1⇒S),18(poly1⇒S),19(poly1⇒S), 20(poly1⇒S),21,22,23	4,16,28,40
6(p⇒S), 7(p⇒S),8(p⇒S),9(p⇒S),10(any),17(poly1⇒C), 18(poly1⇒C),19(poly1⇒C),20(poly1⇒C),21,22,23	9, 21,33,45
6(p⇒S),7(p⇒S),8(p⇒S),9(p⇒S),10(any), 17(poly1⇒S),18(poly1⇒S),19(poly1⇒S), 20(poly1⇒S),21,22,23	10,22,34,46
6(p⇒C),7(p⇒C),8(p⇒C),9(p⇒C),10(any),17(any), 24(poly1⇒C),25(poly1⇒C),26(poly1⇒C), 27(poly1⇒C),28	5,17,29,41
6(p⇒C),7(p⇒C), 8(p⇒C),9(p⇒C),10(any),17(any),24(poly1⇒S), 25(poly1⇒S),26(poly1⇒S),27(poly1⇒S),28	6,18,30, 42
6(p⇒S),7(p⇒S),8(p⇒S),9(p⇒S),10(any),17(any), 24(poly1⇒C),25(poly1⇒C),26(poly1⇒C), 27(poly1⇒C),28	11,23,35,47
6(p⇒S),7(p⇒S), 8(p⇒S),9(p⇒S),10(any),17(any),24(poly1⇒S), 25(poly1⇒S),26(poly1⇒S),27(poly1⇒S),28	12,24,36, 48

Table 3: A possible set of paths satisfying the *all-du-paths*^{p} criterion

[4] M. J. Harrold and M. L. Soffa. Computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[5] D. G. J. Dean and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95*, pages 77–101. Springer-Verlag, 1995. LNCS 952.

[6] R. N. H. J. Vitek and J. S. Uhl. Compile-time analysis of object-oriented programs. In Springer-Verlag, editor, *Proceedings of the 4th International Conference on Compiler Construction (CC'92)*, pages 236–250, 1992. LNCS 641.

[7] S. Kirani. *Specification and Verification of Object-Oriented Programs*. PhD thesis, University of Minnesota, Minneapolis, Minnesota, December 1994.

[8] J. Knoop and W. Golubski. Abstract interpretation: A uniform framework for type analysis and classical optimization of object–oriented programs. In *Proceedings of the 1st International Symposium on Object–Oriented Technology "The White OO Nights" (WOON'96) (St. Petersburg, Russia).*, pages 126 – 142, 1996. Proceedings are also available by anonymous ftp: ftp.informatik.uni–stuttgart.de/pub/eiffel/WOON_96.

[9] R. Mandl. Orthogonal latin squares: An application of experimental design to compiler testing. *Communications of the ACM*, 1985.

[10] J. McGregor and T. Korson. Testing of the polymorphic interactions of classes. Technical Report TR-94-103, Clemson University, 1994.

[11] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., second edition, 1997.

[12] A. Orso. Integration testing of procedural object oriented programs with polymorphism. Technical report, Politecnico di Milano, 1998.

[13] A. Orso and S. Silva. Open issues and research directions in Object Oriented testing. In *4th International Conference on "Achieving Quality in Software: Software Quality in the Communication Society" (AQUIS'98)*, Venice, April 1998.

[14] H. D. Pande and B. G. Ryder. Static type determination for c++. Technical Report LCSR-TR-197-A, Rutgers University, Lab. of Computer Science Research, October 1995.

[15] A. Paradkar. Inter-Class Testing of O-O Software in the Presence of Polymorphism. In *Proceedings of CASCON96*, Toronto, Canada, November 1996.

[16] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the 9th ACM SIGPLAN Annual Conference on Object-Orinted Programming, Systems, Languages, and Applications (OOPSLA'94)*, pages 324–340, 1994. ACM SIGPLAN Notices 29, 10.

[17] S. Rapps and E. J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, Apr. 1985.