Interclass Testing of Object Oriented Software

Vincenzo Martena
Politecnico di Milano
Dip. di Elettronica e Informazione
martena@elet.polimi.it

Alessandro Orso
Georgia Inst. of Technology
College of Computing
orso@cc.gatech.edu

Mauro Pezzè
Università degli Studi
di Milano Bicocca - DISCo
pezze@disco.unimib.it

Abstract

The characteristics of object-oriented software affect type and relevance of faults. In particular, the state of the objects may cause faults that cannot be easily revealed with traditional testing techniques. This paper proposes a new technique for interclass testing, that is, the problem of deriving test cases for suitably exercising interactions among clusters of classes. The proposed technique uses data-flow analysis for deriving a suitable set of test case specifications for interclass testing. The paper then shows how to automatically generate feasible test cases that satisfy the derived specifications using symbolic execution and automated deduction. Finally, the paper demonstrates the effectiveness of the proposed technique by deriving test cases for a microscope controller developed for the European Space Laboratory of the Columbus Orbital Facility.

1. Introduction

The object-oriented paradigm is successfully applied in many software projects, and the use of object-oriented languages is increasingly widespread. Object-oriented technologies can reduce or eliminate some problems typical of procedural software, but may introduce new problems that can result in classes of faults hardly addressable with traditional testing techniques [2, 30]. In particular, state-dependent faults tend to occur more frequently in object-oriented software than in procedural software; almost all objects have an associated state, and the behavior of methods invoked on an object depends in general on the object's state. Such faults can be very difficult to reveal because they cause failures only when the objects are exercised in particular states.

In this paper, we extend our previous results in the automatic generation of test cases for single classes [3] to address the problem of interclass testing, i.e., test of interactions among classes. We present a technique that automatically produces test case specifications from object oriented code, and automatically generates feasible test cases for a subset of object oriented programs that include most safety critical applications. The technique incrementally generates test cases starting from simple classes (i.e., classes with scalar instance variables only) and then moving to more complex classes (i.e., classes whose instance variables are objects or references to objects).

In the paper, we also present an empirical validation of the technique. We describe our prototype tool that implements the technique and illustrate how we used the tool to generate test cases for a microscope controller developed for the European Space Laboratory of the Columbus Orbital Facility.

2. Background

In previous work, we presented a technique for intraclass testing[1] that allows for generating test cases for single classes [3]. In this section, we present the essential information required to understand the technique proposed in this paper for interclass testing. The technique is based on the idea that the execution of a method is affected by the instance variables used by the method, and thus by the methods that determine the values of such variables. Therefore, we identify pairs of methods that define and use the same instance variable and then try to select a feasible sequence of method invocations that contains the identified definition and use in the correct order. The identified sequences represent the test cases for the target class.
The technique consists of three main steps:

Data flow analysis. This phase computes def-use association, that is, ordered pairs of statements in which the first statement defines and the second statement uses the same instance variable. The def-use association are identified with data-flow analysis of the whole class, focusing on instance variables only.

Symbolic execution. This phase computes conditions related to path executions and variable definitions. For every path within each method, we identify the conditions associated with the execution of that path, the relationship between input and output values of the method with respect

[1] Intraclass testing tests the interactions of the methods of a class when they are called in various sequences.

to that path, and the set of variables defined along the path. This information is obtained by applying well-tried symbolic execution techniques to the method's code [6]. Cycles are treated by either fixing an upper bound to the number of loop iterations or adding suitable invariants.

Automated deduction. This phase produces complete sequences of method invocations that exercise the def-use associations identified during the first phase. Such sequences are incrementally built by applying automated deduction techniques to method preconditions and postconditions that are output by the second phase.

This technique is useful for unit testing of *simple classes*, that is, classes whose member attributes are scalar variables only. Although the analysis of simple classes is fundamental for generating test cases for object-oriented software, it does not apply directly to *complex classes*, that is, classes that contain instances of or references to other classes. In the next section, we show how to extend this technique to interclass testing.

3. Interclass Testing

Interclass testing is the testing of a set of classes composing a system or subsystem, usually performed during integration. Typically, such classes are not stand-alone entities, but mutually cooperate in several ways. These relationships among classes are a fundamental characteristic of object-oriented systems and define the nature of the interactions among objects at runtime. Different classifications exist of interclass relationships [26]. In this paper, we are interested in relations of aggregation and use. An *aggregation* relation holds between two classes A and B when an object of type A can include one or more objects of type B.[2] In such a case, the state of an object of type A depends on the state of the object(s) of type B it contains. A *use* relation holds between two classes A and B when one or more methods of A have at least a local variable or a parameter of type B.

The technique presented in this paper generates sequences of calls for a set of objects composing a subsystem. The generated sequences cover pairs of methods that modify and use the state of the same object. By doing so, the sequences exercise the objects in the subsystem in different states, and can reveal errors that occur only when an object of a class is in particular states.

To be able to extend our intraclass test-generation technique to the interclass case, we must account for the two relationships between objects mentioned above: aggregation and use. In particular, we must (1) extend the concept of def-use association to the case of variables that are not scalar entities, but objects, and (2) extend the intraclass data-flow analysis to be able to incrementally analyze a set of classes by reusing summarized information for each class. The technique first automatically produces test case specifications and then generates feasible test cases for the produced specifications.

[2]This relation is also known as *has-a* or *part-of* relation.

3.1. Generating Test Case Specifications

The test case specifications for interclass testing produced with our technique are described as pairs of methods. The first method modifies the state of the object, whereas the second method accesses the modified state. Pairs of methods are generated by (1) identifying an order of classes that allows for incremental data flow analysis, and (2) performing incremental data-flow analysis that allows for dealing with classes whose state consists of instances of other classes.

Class Ordering. To perform incremental data-flow analysis at the interclass level, we must be able to analyze a class at a time and summarize the analysis results for each class. To this end, we must analyze classes that are used by or contained in other classes before the using or containing classes. Therefore, the first step that we perform in our interclass analysis is the identification of an analysis order based on the aggregation and use relationships among classes.

The two binary relationships over the set of classes C define a directed graph whose nodes are the classes in the set, and whose edges represent use and/or aggregation relations. In the following, we safely assume the graph to be connected. A non-connected graph would imply the presence of independent subsystems, which could be analyzed separately.

In well-designed object-oriented software, the structure and the dependencies should result in a DAG. This is confirmed by the cases that we analyzed so far. If the graph is a DAG, there exists a partial order on the elements of the graph, and it is possible to define a topological total order over such elements. The analysis of classes according to this total order lets us analyze a used class before the classes that use it and a contained class before the classes that contain it. If the graph contains cycles, they can be eliminated by deleting one or more edges and manually providing the information that is normally computed automatically from the edges, as described in the next subsections.

Because of the nature of the use and aggregation relation, the first classes in the total ordering, which correspond to leaves in the graph, are classes whose state does not depend on other classes (i.e., classes whose member variables are all scalar attributes). Therefore, we can apply our intraclass technique to such classes. After analyzing this first set of classes, we summarize the analysis results and use them to continue the analysis on the following classes.

Interclass Data-Flow Analysis. A *def-use association* is a triple (d, u, v) where d and u are statements and v is a variable, d defines v, u uses v, and there exists a path from d to u along which v is not redefined (a *def-clear paths*). Traditional data-flow techniques for procedural programs have been extended by Harrold and Rothermel to handle object-oriented code [15]. To perform intra-class testing, we apply this technique to compute a subset of the def-use association for a class [3]. The subset that we are interested in computing are all the def-use associations involving scalar instance variables of a class.

To be able to perform interclass data-flow analysis, we need to extend the data-flow technique presented by Harrold and Rothermel [15]. For a scalar variable, definitions and uses are syntactically identifiable in a straightforward way: any assignment to the variable is a definition, and any access to its value is a use. For an object, however, a definition is not necessarily a simple assignment. Consider the example in Figure 1.

```
class Foo {                 class Bar {
  public:                     private:
   int x;                      Foo foo;
1. void incX() {              public:
2.  x++;                       void m() {
   }                        5.  foo.x=0;
3. int getX() {             6.  foo.incX();
4.  return x;               7.  print(foo.getX());
   }                           }
};                          };
```

Figure 1. Example of direct defs and uses of objects.

We can easily identify a definition of object `foo` at statement 5, but we cannot assess the effect of statement 6. Statement 6 may contain a definition of `foo`, a use of `foo`, or both, depending on the semantics of method `Foo::incX`. Therefore, the analysis of statement 6 requires a prior analysis of class `Foo`. The same consideration holds for statement 7. On the other hand, after analyzing class `Foo`, we are able to classify its methods as methods that define, use, or both use and define `Foo`'s attributes (i.e., methods modifying, inspecting, or both inspecting and modifying `Foo`'s state). For this example, we classify `Foo::incX()` as a method that inspects and modifies `Foo`'s state, and `Foo::getX()` as a method that inspects `Foo`'s state. By using this summary information, we can correctly identify a use and a definition of `foo` at statement 6 and a use of `foo` at statement 7.

In general, we say that an object is defined (resp., used) at a statement when any of its member variables is defined (resp., used) at that statement. This definition is recursive because a member variable can in turn be an object. Therefore, before analyzing a class C, we need summary information for the set S of classes that C is directly or indirectly depending on. As an example, consider the slightly different version of classes `Foo` and `Bar` in Figure 2.

To be able to analyze class `Bar`, we need summary information for class `Foo` (direct dependence) and for class `Complex` (indirect dependence through `Foo`).

```
class Complex;              class Bar {
                              private:
                               Foo foo;
class Foo {                   public:
  public:                      void m() {
   Complex x;                    ...
1. void incX() {            3.   foo.incX();
2.  x.incRe();                   ...
   }                           }
};                          };
```

Figure 2. Example of indirect defs and uses of objects.

In this context, computing summary information for a class means classifying the methods of the class as modifier, inspector, or inspector-modifier methods,[3] according to the effect of the method executions on the state of the class. A method is a *modifier method* if its invocation causes a modification on the state of the class, that is, an assignment to a member variable of the class or the invocation of a *modifier method* on a member variable. A method is an *inspector method* if its invocation causes the use of the value of a member variable or the invocation of an *inspector method* on a member variable. A method is an *inspector-modifier method* if it is both an *inspector method* and a *modifier method*. Methods independent from the state of the class (i.e., methods that neither modify nor use instance variables) are ignored in the incremental analysis.

To analyze a class C, we need to compute summary information for the closure of both the aggregation and use relations starting from C. The total ordering of the classes in the system, computed in the previous step, lets us perform such computation in an efficient way by always analyzing a class before the classes that depend on it.

By using summary information, we are able to perform interclass data-flow analysis incrementally. At each analysis step, we consider a single class. Interactions with other classes are implicitly taken into account using summary information that identifies inspector and modifier methods.

So far, we considered methods comprising a single path. If a method contains more than one execution path, the method can be a modifier along some paths, an inspector along some other paths, an inspector-modifier along yet other paths, and cannot affect at all the state of the object along different paths. If we lose the distinction among the different paths when computing the summary information, we can generate incorrect results. For example, we can generate a sequence of calls that contains a modifier and an inspector such that neither the modifier actually traverses any definition nor the inspector traverses any use.

[3]According to the traditional object-oriented terminology [26], a method is classified as a *modifier* if it changes the value of one or more instance variables of the class it belongs to; and a method is classified as an *inspector* if it accesses the value of one or more instance variables of the class it belongs to.

To consider the different characteristics of the paths within a method, we can simply consider each path in a method as if it were a different method of the class, and thus apply data-flow analysis as described above to each path. This approach works well for small- and mid-sized applications, but may be impractical for large and complex applications, since at every step we must consider all combinations of paths identified at the former steps.

A similar problem occurs also for traditional data-flow testing techniques: in large programs the number of combinations of definitions and uses can grow very large. Traditional data-flow testing techniques overcome this problem by introducing different testing criteria that require to cover different subsets of combinations of definitions and uses. In this way, it is possible to contain the number of test cases to be generated at the price of performing a less effective test [13].

In our approach, we adopt an analogous solution. When the size of the system does not allow to consider each path independently, at each step we select only one path representative of each data-flow fact occurring in the method.

The reduction can be selectively applied during the analysis, and involves user intervention. When the number of combinations of paths considered grows bigger than a given threshold, the user can backtrack the analysis and decide what paths to discard, based on his or her knowledge and depending on their importance. The reduction implies that the definitions and uses occurring in an ignored path are tested only up to the point where the path is discarded from the analysis. This incremental abstraction of details is a common practice in testing of large and complex systems.

3.2. Generating Test Cases

The def-use associations produced by data-flow analysis can be used to identify a set of test cases for interclass testing. A test case corresponding to a def-use association is a sequence of method invocations that starts with a constructor and includes the invocation of both methods occurring in the def-use associations through a def-clear path. We attempt to generate test cases corresponding to def-use associations by first identifying pre- and post-conditions for the executions of paths within single methods using symbolic execution, and then matching the generated conditions using automated deduction techniques.

Interclass Symbolic Execution. As a result of the previous phase, we get for each class a set of pairs of methods that define and use the state of a given object. For each pair, the first method contains a definition of the state of the object it belongs to. Such definition can be either direct or indirect, through a call to a modifier method; in both cases, we simply refer to it as *the definition* in the following. Analogously, the second method contains a use of the state of the object, which can be either direct or indirect, through a call to an inspector method; in both cases, we simply refer to it as *the use* in the following.

We use symbolic execution to compute the execution conditions of single paths within methods. Symbolic execution has been proposed in the seventies as a method for generating test cases for procedural programs [5, 19]. In the last decade, symbolic execution has been applied to many application domains [7, 8, 9, 22]. In this paper, we use symbolic execution on paths within single methods as an aid to identify feasible sequences of method invocations. We limit our attention to paths that contain the definitions and the uses identified in the previous step. Symbolic execution is performed on each method of each class, one method at a time. Symbolic execution lets us compute, for the relevant paths within each method, (1) the conditions associated with the execution of the path, and (2) the relationship between input and output values of the method with respect to the path.

Classes are analyzed in the order identified in the first step, that is, from classes with simple state to classes with state that includes objects of other classes. Methods within a class are analyzed in the order defined by the intraclass call graph [16], starting from leaf methods. In this way, methods are symbolically executed before their callers. Thus, a call can be executed symbolically by checking the validity of the pre-condition of the called method in the current symbolic state. If the pre-condition is satisfied, we substitute the symbolic values occurring in the current symbolic state with the symbolic values of the corresponding variables after the execution of the called method, according to the method's post-conditions. The path condition is also updated with the pre-conditions of the called method. If the method contains several paths, they must be considered with the same strategy illustrated in the previous subsection, that is, each method call shall be considered a call to each different path individually. In the presence of direct or indirect recursion, the call graph contains cycles. In this case, the involved methods cannot be executed before their callers, and we perform symbolic execution by unfolding the calls up to a given threshold.

Symbolic execution may fail to compute execution conditions for a path because of the presence of unsolvable constraints or unbounded loops. However, we apply symbolic execution to single methods, which are usually procedures with a simple control structure [34]. We also bind the number of executions of loops, so as to avoid potentially unbounded computations. Moreover, the failure of symbolic execution on a method does not affect the possibility of analyzing other methods, and can be overcome by adding suitable information, such as loop invariants [7].

Symbolic execution can deal with simple usages of pointers (e.g., in parameter passing), but cannot handle unconstrained dynamic allocation of memory. For generating test cases in the presence of pointers, programs must be suitably annotated with assertions, as illustrated in Section 6.

Test Case Generation. The last step of our technique consists in the generation of sequences of calls to methods that exercise each def-use association.[4]

In the rest of this section, we use the following terminology: u is a statement that contains a use of variable v; d is a statement that contains a definition of variable v; m_u is the method that contains u; m_d is the method that contains d; PCU is the path-condition of m_u;[5] a *def-free* method with respect to a variable v is a method whose execution does not cause a redefinition of v; finally, a *def-free* path with respect to a variable v is a sequence of def-free methods with respect to v.

Given a def-use association (d, u, v) for a class C, a sequence of method invocations that exercises the association must satisfy the following properties:

- Begin with the invocation of a constructor of class C. (The constructor instantiates C and all classes that define directly or indirectly the state of C.)
- Contain an invocation of method m_d that causes the execution of statement d (Such statement can be a simple assignment or an invocation of either a modifier or an inspector-modifier method.)
- Contain an invocation of method m_u that causes the execution of statement u. (Like statement d, such statement can be a direct use or an invocation of either an inspector or an inspector-modifier method.)
- The sequence of invocations between m_d and m_u must be a def-clear path with respect to variable v (i.e., it cannot cause the execution of any statement that defines, either directly or indirectly, variable v).

We generate such a method sequence in reverse order, starting from m_u and applying a set of backward-chained deductions. In brief, our initial goal is condition *PCU*. If there is a method whose postconditions imply *PCU*, we add the method to the front of the sequence. If no such method is found, we look for a method whose postconditions do not contradict *PCU*, and add it to the front of the sequence—we refer to the added method as m_k hereafter. To compute the new goal, we (1) simplify the current condition by eliminating those clauses (if any) satisfied by m_k's postconditions, (2) perform the union of the simplified condition and m_k's preconditions, and (3) further simplify the resulting condition, if possible. Because, at each chaining step, there can be multiple suitable m_k methods, the deductive process for a given def-use association can be represented as a tree. Each tree node corresponds to a pair consisting of a method and a condition (a predicate on the instance variables of the class and on the parameters of the method). The root of the tree corresponds to method m_u and condition *PCU*. Notice that our method does not impose a given sequence of methods' invocations, but finds at least one sequence of methods' invocations for each pair.

The first objective of the deductive process is to add a node corresponding to m_d to the tree. The second objective is to add a class constructor to the subtree rooted at m_d. The deductive process ends when this second objective is satisfied. A constraint on the deductive process is that only def-free methods with respect to v can be added to the tree before the first objective is met (i.e., the path between m_d and m_u must be a def-clear path).

The deductive process may end for one of three reasons: (1) both the above objectives are met, that is, the search for a feasible method sequence is completed successfully; (2) the tree cannot be further expanded, which means that the def-use association is infeasible; or (3) the depth of the tree reaches a given threshold before a feasible sequence is found.

To improve the efficiency of the deductive process, we use several heuristics. First, we reduce the deduction-tree size by pruning subtrees whose roots have conditions that imply a predecessor's condition. This is achieved by avoiding further exploration of such roots, unless they correspond to either method m_d or a constructor, whose inclusion in the tree represents an objective of the deduction process. Second, we insert a node corresponding to the method responsible for the definition (i.e., m_d) as soon as possible and as a unique successor. Also, when exploring the successors of m_d, we insert a constructor in the tree as soon as possible.

Our technique for the construction of the tree is based on the use of automated deduction. Even if automated deduction techniques, such as constraint solving, may fail when coping with complex expressions, nowadays there are several efficient constraint solvers that can efficiently handle large sets of expressions (e.g., [31, 32]). In addition, failures of the constraint solver can be overcome by requiring user intervention.

4. Tool Support

In this section, we outline the architecture of a tool for the automatic generation of test cases based on the technique presented in the paper. We also describe a prototype that has been implemented to experimentally evaluate the technique.

Figure 3 shows the main components of the tool. Each component performs one of the main steps of the method. The *Class Analyzer* parses the source code, generates the interclass control-flow graph, and computes the method and class ordering to be used by the data-flow analyzer and the

[4]It is worth noting that, by construction, def-use associations identified in Step 2 consist of definitions and uses within the same class.

[5]For the sake of clarity, we present the test case generation phase assuming that each method contains exactly one path. The case of methods containing more paths can be addressed by treating each path as a different method and by possibly considering only subsets of paths, if the number of paths becomes unmanageable.

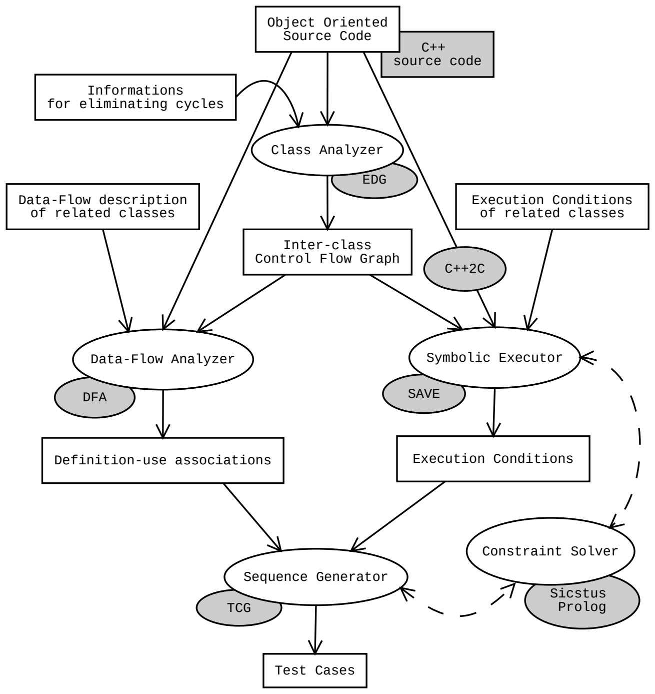
Object Oriented Source Code
C++ source code
Informations for eliminating cycles
Class Analyzer
EDG
Data-Flow description of related classes
Execution Conditions of related classes
Inter-class Control Flow Graph
C++2C
Data-Flow Analyzer
Symbolic Executor
DFA
SAVE
Definition-use associations
Execution Conditions
Constraint Solver
Sequence Generator
Sicstus Prolog
TCG
Test Cases

Figure 3. Software architecture for the test-case generator.

symbolic executor for their incremental analysis. At this stage, the analysis may require additional user-provided information for eliminating cycles from the graph.

The *Data-Flow Analyzer* computes def-use associations in the classes under test. The analysis is performed starting from simple to more complex classes, according to the order identified by the Class Analyzer.

Externally-provided data-flow information may be used for classes that have been already analyzed, such as external code belonging to previously-analyzed packages, or that we cannot analyze (e.g., binary libraries).

The *Symbolic Executor* computes the execution conditions of paths containing definitions or uses. Symbolic execution of the methods in the classes composing the system is performed following the order computed by the Class Analyzer. Like for the Data-Flow Analyzer, externally-provided information may be used for classes that have been already analyzed or for classes whose source code is not available.

The *Sequence Generator* is responsible for computing sequences of method invocations that exercise def-use associations within classes. To this end, the generator exploits the information provided by the Data-Flow Analyzer (i.e., the def-use associations) and by the Symbolic Executor (i.e., the execution conditions for paths within methods). Both the Symbolic Executor and the Sequence Generator rely on a constraint solver for the simplification of expressions.

Based on above architecture, we implemented a prototype test-case generator for C++ code. The grey boxes in Figure 3 provide the details of the implementation of the corresponding architectural elements (white boxes). The Class Analyzer implementation relies on the *EDG* front-end compiler [12], which handle the complete C++ language.

The Data-Flow Analyzer is based on a polynomial algorithm specialized for the computation of def-use association involving class members [29]. The current implementation of the analyzer handles only C++ code that does not use exception-handling constructs, polymorphic calls, and pointers.

The Symbolic Executor implementation is based on SAVE, a symbolic executor for C [6]. To be able to use SAVE, we translate single C++ methods to C code.

Both the Sequence Generator and the Symbolic Executor use the Constraint Solver based on the Sicstus prolog libraries for simplification [32].

5. Empirical Study

In this section, we illustrate a case study performed on an application provided by our industrial partners to assess the applicability of our approach. We used the prototype described in Section 4 to generate test cases for a microscope controller developed for the European Space Laboratory of the Columbus Orbital Facility (hereafter *MSController*).

MSController is part of the on-board system that controls (1) the motors responsible for the movement along the three axes of the microscope's slide support, (2) the automatic focus of the microscope, and (3) the selection of different slides and different lenses.

Class	Lines of code	Methods	Paths	Included Objects
Coord	67	6	6	0
Slide	37	3	3	0
Lens	47	4	4	0
Motor	85	5	14	0
Arm	81	5	14	0
SlideSelector	58	3	23	1
MScope	121	5	89	7
Total	496	31	139	

Table 1. Characteristics of the analyzed C++ code.

Table 1 summarizes the main characteristics of the analyzed code. The numbers of lines of code shown in the table do not include comments. Classes `Motor` and `Arm` contain cycles and thus an unbounded number of paths. The number of paths indicated in Table 1 refers to the bounded number of paths automatically analyzed by the tool, which considers up to two iterations of each loop. The complete code of *MSController* is given in Reference [24].

We performed the study by (1) seeding state-dependent faults in the program, (2) using our tool for generating test cases for the system, and (3) assessing the effectiveness of the generated test cases in revealing the seeded faults. We inserted faults by randomly picking a set of definitions and uses of instance variables and modifying them. The seeded faults affected both direct and indirect definitions and uses of instance variables. Seeded faults were distributed as follows: two faults in class `Motor`, one fault in class `Arm`, two faults in class `SlideSelector`, and four faults in class `MScope`.

The interclass control-flow graph produced by the Class Analyzer does not contain cycles. The Class Analyzer computed the following class order: `Coord`, `Slide`, `Lens`, `Motor`, `Arm`, `SlideSelector`, and `MScope`. The Data-Flow Analyzer automatically computed 471 def-use associations and the Symbolic Executor successfully produced the execution conditions for all paths without requiring manual inputs. The Sequence Generator automatically produced 128 test cases cases. The generated test cases revealed seven of the nine seeded faults. The two unrevealed faults were located in paths that were infeasible due to other seeded faults. After removing the seven revealed faults, the tool was able to reveal the two remaining faults.

Class	def use associations	covered def use ass.	execution time
Coord	18	18	< 1 sec
Slide	2	2	< 1 sec
Lens	4	4	< 1 sec
Motor	50	37	43 sec
Arm	40	29	82 sec
SlideSelector	240	109	1630 sec
MScope	627	257	4536 sec
Total	981	456	6294 sec

Table 2. Def-use associations for the analyzed C++ code.

After removing all the nine seeded faults, we generated test cases for the resulting system and we did not find any additional fault. Table 2 lists, for each class, the number of def-use associations generated by the Data Flow Analyzer, the number of associations covered by the generated test cases, and the execution times of the prototype. For the study, we used a system running Linux (kernel 2.4.18) on an 850 MHz Athlon with 512 megabytes of RAM.

The considerable number of def-use associations is due to the combinatorial effect of combining paths while traversing the aggregation hierarchy, as discussed in Section 3. The combinatorial effect is also responsible for the increasing percentage of infeasible def-use associations because the def-use associations of higher-level classes include also combinations of infeasible paths within lower-level classes. Obviously, the combinatorial effect could be significantly reduced by discarding, through user intervention, a subset of paths while traversing the aggregation hierarchy. The execution times of the data-flow analyzer and the symbolic executor are almost negligible. Most of the computation time is spent by the constraint solver used by the Sequence Generator. The interested reader can refer to Reference [24] for a detailed description of the technique that we used to seed faults, a complete list of the generated faults, and details about the set of test cases generated by the tool for the considered subject.

6. Limitations and Applicability of the Technique

In the former sections, we indicated some limitations for the applicability of our technique. In this section, we summarize such limitations. We first discuss a set of object-oriented features that are not handled by our technique, but that may be handled orthogonally using different approaches. We then discuss the main limitations of the technique proposed in this paper. Finally, we briefly recall the current limitations of the prototype that constraint the experimental evaluation of the technique.

Object-Oriented Features.

Object-oriented programs contain several features that require specific attention during testing: hidden state of objects, inheritance, polymorphism and dynamic binding, and exception handling. The technique proposed in this paper mostly accounts for problems related to the objects' state, and does not specifically address problems related to other object-oriented features. (However, the presence of such features does not prevent the applicability of our technique.) Therefore, the test cases generated with our approach must be complemented with test cases generated using approaches specifically designed for addressing different object-oriented features. The generation of different test suites for dealing with different characteristics of the application under test is common good practice, and allows for a useful and sometime needed separation of concerns.

Carefully considering inheritance allows for reducing the testing effort by avoiding re-testing methods already tested in the ancestor classes. Ignoring inheritance by flattening class hierarchies simply results in the generation of redundant test cases. Specific techniques, such as the testing history approach presented by Harrold, McGregor, and Fitzpatrick [14], can be used to reduce the number of test cases to be generated.

Testing object-oriented programs in the presence of polymorphism and dynamic binding requires considering different bindings for each polimorphic call. Such bindings can be computed with specialized techniques (e.g., the technique proposed by Orso and Pezz`e in Reference [28]). The presence of information about which bindings to test could be used to generate additional test-case specifications for our technique.

Exception-handling constructs complicate testing due to the presence of implicit transfers of control that occur when exceptions are raised. Again, there are testing techniques that are specifically designed to address exception handling (e.g., the technique proposed by Sinha and Harrold in Reference [33]) and that could be used in conjunction with our technique.

Limitations of the Technique.

Our technique comprises two main phases: generation of test-case specifications through data-flow analysis and generation of feasible test cases through symbolic execution and automated deduction.

Test-case specifications can be automatically generated for almost any object-oriented program. To apply the technique, we need the source code of the program, and we require that the aggregation and use relationships among classes do not form cycles. The availability of the source code is a common requirement of structural testing approaches, but excludes the possibility of handling pre-compiled libraries. The presence of such libraries can be overcome by indicating inspector and modifier methods in the libraries and by providing pre- and post-conditions for method execution. Absence of cycles in aggregation and use relationships is considered good practice in the design of object-oriented software. (Cycles are in fact not present in several object-oriented programs that we examined.) Moreover, even in the presence of cycles, it is always possible to break such cycles by eliminating one or more aggregation or use relations—although the breaking requires the involvement of the user.

The presence of pointers does not affect the applicability of our technique. The only requirement for our data-flow analysis in the presence of pointers is the availability of some kind of alias analysis, such as the one provided by SUIF [35]. The presence of methods with multiple or even virtually infinite paths is not a problem for data-flow analysis. However, as discussed in Section 3, incrementally reducing the number of paths can largely reduce the computations costs (and can be done with little overhead by expert users).

Test case generation from test case specifications is based on symbolic execution and automated deduction techniques. Despite the presence of powerful tools (e.g., [6, 32]), such techniques have some intrinsic limitations. Symbolic execution can handle all static data structures, including arrays [7], but cannot deal with unconstrained dynamic memory allocation. Therefore, although our technique can automatically generate test-case specifications for almost all object-oriented programs, it can automatically generate corresponding test cases only for programs that do not make use of unconstrained dynamic memory allocation. Safety critical applications belong to this class of programs. In fact, we successfully applied our tool for generating test cases for safety critical applications, such as the one presented in Section 5.

Moreover, the presence of dynamic memory allocation can be partially overcome by manually inserting pre- and post-conditions for methods that make use of such feature. For example, method `pop` of a class `stack` that uses a linked list can be annotated with a pre-condition that requires the stack not to be empty and a post-condition that indicates the modification in the number of elements in the stack. A good use of encapsulation can greatly simplify the job of annotating programs. However, the presence of unconstrained use of dynamic memory may greatly reduce the applicability of our technique for automatically generating test cases. In these cases, our technique will only generate test-case specifications and, possibly, a subset of test cases for those parts of the program that do not use dynamic memory.

Loops do not constitute a problem for the application of our technique because the technique does not need to consider all possible paths. It is always possible to limit the number of paths by constraining the number of loop iterations or by manually adding invariants [7].

Undecidability of automated deduction may affect the generation of test cases. However, the tools currently available on the market can handle more and more complex expression and give us confidence in the applicability of the technique for a meaningful set of systems.

Limitations of the Prototype.

The prototype described in Section 4 and used in our experiments on C++ programs is not fully implemented and has some additional limitations that constraint its applicability. The data-flow analyzer does not accept code that contains inheritance, polymorphism, exception-handling constructs and gotos (although, conceptually, none of these features constitute a problem for our technique). Similarly, we do not perform alias analysis yet and information about aliases must be pre-computed and provided to the tool. The symbolic executor has not been extended to treat arrays yet, and requires all methods using pointers to be suitably annotated. Annotations are not required for a limited use of pointers, such as in parameter passing. The Sicstus simplifier [32] used for automated deduction is programmed to handle integers and floating point numbers only.

7. Related Work

The problem of revealing state-dependent faults in object-oriented software is extremely relevant and has been tackled in several papers. Most of the papers that address this problem propose functional testing techniques to derive test cases. In particular, several techniques are based on the generation of test cases from either formal specifications [10, 11, 36], or from UML diagrams [1, 17, 25].

Structural testing for object-oriented software has not been widely investigated yet. Some work investigates mutation-based techniques [21]. Other work focuses on the combination of functional and structural testing, but does not tackle the problem of generating structural-based test cases [4, 20]. Yet other work focuses on algorithmic aspects of data-flow analysis, thus building the basis for test case generation [15, 34].

Our technique is code-based rather than specification-based. To our knowledge, the only other technique for generating test cases from code was defined by Kung and colleagues [23]. That technique generates message sequences for class testing from a state-based model extracted from the source code. The intermediate state-based model is a set of finite state machines—one machine for each instance variable. These state machines are built through a combination of symbolic execution and deductive techniques. Subsequently, message sequences are generated through an exhaustive reachability analysis performed by combining the different state machines.

Analogously to our method, the approach of Kung and colleagues performs well when instance variables are scalar and when symbolic execution can be completed successfully. However, their technique fails to produce any useful information when these assumptions do not hold. This is in contrast with our approach, which always produces information useful to testers. An additional disadvantage of their approach is that it does not consider possible dependencies among instance variables in the preconditions on method execution. For instance, if a precondition on the execution of a given path includes even a very simple predicate of the form $(x < y)$, where x and y are instance variables, their approach fails to take this predicate into account when constructing method sequences. Finally, their technique is defined only for single classes.

Some authors tackle the issue of class test automation from a different viewpoint, mostly concerned with the problems related to the generation of scaffolding code [27, 18]. Those approaches start from the tester's knowledge of the source code and seek to generate automatically drivers and stubs for the class under test. Because these techniques are concerned with the automation of the execution of tests, rather than their generation, they are complementary to the technique presented in this paper.

8. Conclusion

We described an approach for the automatic generation of test cases for interclass testing of object-oriented systems. Our approach seems to be quite powerful in that the test cases that we generate automatically can detect faults due to the combined effects of method invocations on the objects' state. An additional advantage of our technique is that it works in phases of increasing complexity; the analysis of a system considers the simple classes in the system first and then uses the results to further analyze more complex classes. Furthermore, by using an incremental approach, we can also account for the possible failure of some of the analysis phases due to their computational complexity; at every step, the technique can exploit user-provided information (in the form of summary information) to abstract details of the analyzed classes and to complete the test-case generation process. Finally, our approach is a generative technique. Consequently, it does not require code instrumentation to ensure the adequacy of the generated test cases because coverage is granted by construction.

The applicability of the approach has been evaluated by examining an application implementing a microscope controller developed for the European Space Laboratory of the Columbus Orbital Facility. We successfully generated a set of test cases for the system using our prototype, and the generated test cases were able to reveal a set of faults seeded in the program.

We are currently working on improving the implemented prototype to reduce some of its limitations and on identifying additional systems to be used as subjects for experimentation.

Acknowledgments

Giordano Sassaroli helped with the design and the implementation of the prototype. This work was supported in part by the ESPRIT Project TWO (EP n.28940), by the MIUR in the framework of the QUACK Project, by a grant from Boeing Aerospace Corporation to Georgia Tech, by NSF awards CCR-9988294, CCR-0096321, and EIA-0196145 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission.

References

[1] A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In A. Evans, S. Kent, and B. Selic, editors, *Proceeedings of the Third International Conference on The Unifi ed Modeling Language, Advancing the Standard, York, UK*, volume 1939 of *LNCS*, pages 383–395. Springer, October 2000.

[2] S. Barbey and A. Strohmeier. The problematics of testing object-oriented software. In M. Ross, C. A. Brebbia, G. Staples, and J. Stapleton, editors, *Proceedings of the Second Conference on Software Quality Management, Edinburgh (Scotland, UK)*, volume 2, pages 411–426, July 1994.

[3] U. Buy, A. Orso, and M. Pezzé. Automated testing of classes. In *Proceedings of the International Symposium in Software Testing and Analysis (ISSTA '00)*, pages 39–48, 2000.

[4] H. Y. Chen, T. H.Tse, F. T.Chan, and T. Y.Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250–295, July 1998.

[5] L. A. Clarke. A system to generate test data and symbolically execute programs. *Transactions on Software Engineering*, 2(3):215–222, September 1976.

[6] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezz`e. Using symbolic execution for verifying safety critical systems. In *Proceedings of the 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE01)*, ACM Software Engineering Notes, NY, September 2001.

[7] A. Coen-Porisini, F. D. Paoli, C. Ghezzi, and D. Mandrioli. Software specialization via symbolic execution. *IEEE Transactions on Software Engineering*, (SE-17(9)):884 – 899, September 1991.

[8] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test-data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, Sept. 1991.

[9] L. K. Dillon. Using symbolic execution for verifi cation of Ada tasking programs. *ACM Transactions on Programming Languages and Systems*, (12(4)):643 – 669, 1990.

[10] M. Donat. Automating Formal Specifi cation Based Testing. In M. Bidoit and M. Dauchet, editors, *Proceedings of the International Conference on Theory and Practice of SW Development (TAPSOFT 97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 833–847, Lille, France, 1997. Springer-Verlag, Berlin.

[11] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[12] Edison Design Group. C++ front end. Technical report, Edison Design Group Inc., 2000.

[13] P. G. Frankl and E. G. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, October 1993.

[14] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrich. Incremental Testing of Object-oriented Class Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 68–80, May 1992.

[15] M. J. Harrold and G. Rothermel. Performing data fbw testing on classes. In *2nd ACM-SIGSOFT Symposium on the foundations of software engineering*, pages 154–163. ACM-SIGSOFT, December 1994.

[16] 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.

[17] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *Proceedings of the International Symposium in Software Testing and Analysis (ISSTA '00)*, pages 60–70. IEEE Computer Society Press, 2000.

[18] D. Hoffman and P. Strooper. ClassBench: A framework for automated class testing. *Software Practice and Experience*, 27(5):573–597, May 1997.

[19] W. E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *Transactions on Software Engineering*, 3(4), July 1977.

[20] P. Jorgensen and C. Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38, September 1994.

[21] S. Kim, J. A. Clark, and J. A. McDermid. Class mutation: mutation testing for object-oriented programs. In *Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems*, 2000.

[22] B. Korel. Dynamic method for software test data generation. *JSTVR*, 2(4):203–213, 1992.

[23] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y.-S. Kim, and Y.-K. Song. Developing and oject-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–86, October 1995.

[24] V. Martena, A. Orso, and M. Pezzé. Experimental evalaution of interclass testing of object oriented software. Technical Report LTA-2002-01, Testing and Analysis Lab. - Universit`a degli Studi di Milano - Bicocca, 2002.

[25] J. D. McGregor. Testing models: the requirement model. *Journal of Object-Oriented Programming*, 1998.

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

[27] G. C. Murphy, P. Townsend, and P. S. Wong. Experiences with cluster and class testing. *Communications of the ACM*, 37(9):39–47, September 1994.

[28] A. Orso and M. Pezz`e. Integration testing of procedural object-oriented languages with polymorphism. In *Proceedings of the 16th International Conference on Testing Computer Software: Future Trends in Testing (TCS'99)*, Washington, D.C., June 1999.

[29] A. Orso, F. Saini, and N. Trevisan. Un algoritmo per il calcolo di coppie defi nizione-uso interprocedurali. Technical report, Politecnico di Milano, 1999. (in Italian).

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

[31] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifi - cations: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[32] SICStus prolog 3 - user manual. Technical report, SICStus, 2000.

[33] S. Sinha and M. J. Harrold. Criteria for testing exception-handling constructs in Java programs. In *Proceedings of the International Conference on Software Maintenance*, pages 265–274, Setember 1999.

[34] A. Souter, L. Pollock, and D. Hisley. Inter-class Def-Use analysis with partial class representations. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, volume 24.5 of *Software Engeneering Notes (SEN)*, pages 47–56, N. Y., September 1999. ACM Press.

[35] SUIF Compiler System. Technical report, The Stanford SUIF Compiler Group, September 2001.

[36] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Proceedings of the International Conference on Software Maintenance*, pages 302–310. IEEE Society Press, September 1993.