Automated Testing of Classes *

Ugo Buy
University of Illinois at Chicago
EECS Dept. (M/C 154)
851 South Morgan Street
Chicago, IL (USA) 60607
Phone: +1-312-413-2296
buy@eecs.uic.edu

Alessandro Orso
Politecnico di Milano
DEI
P.zza Leonardo Da Vinci, 32
I-20133 Milano, Italy
Phone: +39-02-2399-3638
orso@elet.polimi.it

Mauro Pezzè
Politecnico di Milano
DEI
P.zza Leonardo Da Vinci, 32
I-20133 Milano, Italy
Phone: +39-02-2399-3523
pezze@elet.polimi.it

ABSTRACT

Programs developed with object technologies have unique features that often make traditional testing methods inadequate. Consider, for instance, the dependence between the state of an object and the behavior of that object: The outcome of a method executed by an object often depends on the state of the object when the method is invoked. It is therefore crucial that techniques for testing of classes exercise class methods when the method's receiver is in different states. The state of an object at any given time depends on the sequence of messages received by the object up to that time. Thus, methods for testing object-oriented software should identify sequences of method invocations that are likely to uncover potential defects in the code under test. However, testing methods for traditional software do not provide this kind of information.

In this paper, we use data flow analysis, symbolic execution, and automated deduction to produce sequences of method invocations exercising a class under test. Since the static analysis techniques that we use are applied to different subproblems, the method proposed in this paper can automatically generate information relevant to testing even when symbolic execution and automated deduction cannot be completed successfully.

Keywords

Testing and Analysis, Testing Object-Oriented Software, Class Testing, Data Flow Analysis, Symbolic Execution.

*This work has been partially supported by the ESPRIT Project TWO (Test & Warning Office - EP n.28940) and by the 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '00, Portland, Oregon.
Copyright 2000 ACM 1-58113-266-2/00/0008...$5.00.

1. INTRODUCTION

The object-oriented paradigm plays a prominent role in the development of many modern software systems. The different structure and behavior of object-oriented software help in solving or mitigating several problems of procedural software, but raise new problems that often cannot be addressed with traditional techniques. In particular, object-oriented software presents new classes of faults that require new testing techniques [17]. Despite the widespread use of the object-oriented paradigm, the many new problems related to testing of object-oriented systems have not been adequately investigated. The most interesting studies published so far address several important issues, but do not cover all issues related to testing of object-oriented software [5, 8, 12, 22, 25].

Here we focus on the issue of class testing. Major problems in class testing derive from the presence of instance variables and their effects on the behavior of methods defined in the class. A given method can produce an erroneous result or a correct one, depending on the value of the receiver's variables when the method is invoked. It is therefore crucial that techniques for the testing of classes exercise each method when the method's receiver is in different states.

Existing approaches to class testing are based either on a specification of the class state or on data flow analysis. Most state-based approaches ([1, 5, 25]) tie test case generation to the existence of suitable class specifications. Unfortunately, we often need to test software whose specifications are incomplete or even nonexistent. The existing techniques for deriving state information from source code usually make strong assumptions on the code that can be analyzed. For example, the method defined by Kung et al. [14] can be applied only to scalar state variables without mutual dependencies. That method does not seem to scale up as the size of the code under test increases. Data flow-based approaches [9, 24] focus mostly on algorithmic rather than methodological aspects of testing; thus, these approaches represent an important basis for testing, but do not completely address the problem of test case generation. In this paper we show how the results of data flow analysis defined by Harrold and Rothermel [9] can be used as part of a method for generating test cases for classes.

This paper unifies existing approaches in a coherent framework that combines data flow analysis, symbolic execution

and automated deduction in an effort to generate method sequences for structural testing of classes. Our framework mitigates well-known limitations of the three techniques we use by restricting their scope of application. In particular, data flow analysis is applied only to instance variables of the class under test (CUT); symbolic execution is applied only to individual methods, which usually have a simple intra-procedural control structure; and automated deduction is used to generate sequences of method invocations. Symbolic execution and automated deduction techniques can sometimes fail, depending on the code of the CUT. When this happens, however, our framework can still generate some relevant sequences of method invocations and produce information (e.g., du-pairs, as we explain below) that can help test engineers identify additional test cases. The applicability of symbolic execution and automated deduction can be increased through user-supplied information, such as loop invariants. The main advantage of our technique is that it can discover errors caused by interactions among class methods. Traditional testing techniques can fail to discover errors of this kind.

It is quite possible that some of the method sequences we generate may never be invoked when the program containing the CUT is executed. This can happen, for instance, if a generated sequence violates a precondition restricting the use of the CUT by its clients. These test cases can still be executed during unit or integration testing through scaffolding. Executing these cases can provide valuable information about the robustness of the CUT.

Our approach uses exclusively structural (i.e., code-based) analysis techniques. Therefore, we aim at automatically generating method sequences for class testing, but we do not address the problem of generating test oracles. The automatic generation of oracles would require the existence of program specifications, which are seldom available in current industrial practice. Given that the generation of oracles is orthogonal to the generation of test data, the approach proposed in this paper can be complemented with any technique, either automated or manual, for the generation of oracles. Consequently, we assume that testers using our framework will define, for each generated methods sequence, expected values of instance variables and return parameters. Issues related to the actual execution of test cases and scaffolding are beyond the scope of this paper. However, we assume that our generated method sequences will be executed by a testing system that provides access to all instance variables of the objects under test.

An implementation of our framework is currently under way. We have completed prototypes that carry out two out of three key steps of our approach, namely data flow analysis and symbolic execution of C++ code. We are now developing a prototype for automated deduction and also integrating the various prototypes into a unified toolset. The toolset will allow us to evaluate empirically the effectiveness of our method on real-world applications.

The paper is organized as follows. Section 2 describes the proposed methodology. Section 3 introduces the example used throughout the paper to illustrate our methodology. Sections 4, 5, and 6 illustrate the main phases of the proposed technique. Section 7 sketches the overall structure of the toolset under development. Section 8 surveys related work. Section 9 outlines future research directions.

2. TEST GENERATION FOR CLASSES

The technique proposed in this paper seeks to reveal state-dependent failures, that is, failures that manifest themselves only when an instance is in a certain state before executing a method. To test for such failures, our technique generates sequences of method invocations that bring the object under test to states in which a given method is then exercised. The underlying idea is quite simple: 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. To test methods for state-dependent faults, we identify pairs of methods that define and use the same instance variable. Once we identify one such pair, we try to select a complete sequence of method invocations that contains the two methods in the correct order. The identified sequences represent the test cases for the target class.

Note that we consider the testing of classes in isolation. Therefore, if the class under test (CUT) belongs to an inheritance hierarchy, we perform a flattening of the classes in the hierarchy prior to the application of the technique. Our technique is based on three main phases:

Data flow analysis. This phase defines so-called du-pairs. In brief, these are ordered pairs of mutually related statements in which the first statement defines and the second statement uses the same instance variable. The du-pairs are defined by applying data flow analysis to the whole class while focusing on instance variables only. Thus, the statements comprising a du-pair may belong to different methods.

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

Automated deduction. This phase identifies complete sequences of method invocations that exercise the du-pairs identified during the first phase. Thus, each method sequence leads to a use of an instance variable through a definition of the same variable starting from an initial state. Such sequences are incrementally built by applying automated deduction techniques to method preconditions and postconditions that are output by the second phase.

The main disadvantage of our testing approach is the computational complexity of the techniques that we employ, especially symbolic execution and automated deduction. However, we significantly reduce the effects of this disadvantage on the overall effectiveness of our approach by applying the

three techniques in phases of increasing complexity and by producing useful results at the end of each phase.

Evidently, our approach is most effective when all three techniques can be applied successfully. In this case, we produce method sequences that exercise all feasible du-pairs in the CUT. In addition, our analyses will statically identify and discard all infeasible du-pairs.

If, however, symbolic execution does not succeed for a given method, our technique will produce at least a set of du-pairs for the CUT. In this case, testers can either build method sequences involving this method by hand or manually define preconditions and postconditions for the method. In the first case, our framework helps developers in establishing data-flow coverage for a test set. In the second case, user-supplied conditions will permit the application of automated deduction in an effort to identify method sequences covering the du-pairs identified earlier.

When symbolic execution is successful but automated deduction fails for a given du-pair, a developer can define method sequences by using methods' preconditions and postconditions. In particular, the preconditions of the two paths that contain the statements in the du-pair can help identify class states that cause these statements to be executed. In addition, the postconditions of each method can help the developer in defining method sequences that will bring an instance into the states of interest.

Data flow analysis works on simple variables. When some instance variables are complex (e.g., arrays, structures, and recursive types), testers can decide whether to skip the analysis of such variables, or to identify manually statements defining and using the variables. For instance, if v is a non-scalar instance variable, e.g. an object, the invocation of a *const* method on v can be considered a use of v, while the invocation of a non-const method on v can be considered a definition.[1] An analysis of this kind could be easily automated. A detailed discussion of this issue is beyond the scope of this paper, which is focused on unit testing of classes. We plan to address the issues arising from the use of class instances as instance variables as part of integration testing.

In summary, the shortcomings of the techniques that we use do not prevent the application of our framework, but simply reduce the number of automatically generated message sequences. This means that the presence of unbounded loops (possibly affecting symbolic execution) or complex expressions (possibly affecting automated deduction) will only result in the generation of a partial set of message sequences. In addition, missing sequences can be easily identified by analyzing the du-pairs not covered by any sequence, thus providing the user with information about program components that are not adequately tested. Users can decide on a case-by-case basis whether to provide additional information (e.g., information on definitions and uses for non-scalar instance variables or methods' preconditions and postconditions). Alternatively, a user could complement our technique with other testing approaches.

[1]In Standard C++ a *const* method cannot modify any of the receiver's instance variables (data members).

3. RUNNING EXAMPLE

We illustrate our technique with the example of class *CoinBox*, which contains a known defect [14]; we automatically generate test cases that uncover the defect. The fault in the class results in a failure only when method *Vend* is invoked in a particular state. Although this example is quite simple, its defect may not be revealed with traditional test-case generation techniques. We show that our technique can automatically generate a test set that exposes the defect. In addition, our test set is smaller than the corresponding test set generated by Kung et al. [14]. Finally, as we show below, our technique works when the value of an instance variable depends on the value of other instance variables, while the technique of Kung et al. does not cope with such cases.

The code for class *CoinBox* appears in Figure 1. Class *CoinBox* has three instance variables, *totalQtrs*, *curQtrs*, and *allowVend*. Variable *totalQtrs* keeps track of all coins collected for items sold by the machine. Variable *curQtrs* keeps track of coins entered since the last sale. The vending machine requires at least two coins before serving a drink. Variable *allowVend* indicates whether enough coins have been entered in order for a sale to take place.

```
class CoinBox {
  unsigned totalQtrs;
  unsigned curQtrs;
  unsigned allowVend;
public:
CoinBox() {
      totalQtrs = 0;
      allowVend = 0;
      curQtrs = 0;
   }
   void returnQtrs() {
      curQtrs = 0;
   }
   void addQtr() {
      curQtrs = curQtrs + 1;
      if (curQtrs > 1)
         allowVend = 1;
   }
   void vend() {
      if (allowedVend) {
         totalQtrs = totalQtrs + curQtrs;
         curQtrs = 0;
         allowVend = 0;
      }
   }
};
```

Figure 1: Class CoinBox.

Method *CoinBox()* is a constructor that initializes all three instance variables to zero. Method *addQtr()* captures the action by which a user enters a coin. It increments variable *curQtrs* and sets variable *allowVend* to one if *curQtrs* is greater than one. Method *vend()* models the purchase of an item by the user: The current value of *curQtrs* is added to variable *totalQtrs*; *curQtrs* and *allowVend* are reset to zero. Method *returnQtrs()* models the actions by which coins entered by a user are returned to the user before a sale. It resets *curQtrs* to zero. Class *CoinBox()* contains a fault in that method *returnQtrs()* does not reset variable *allowVend*. As a result, any sequence containing at least two consecutive invocations of method *addQtr()* followed by an invocation of method *returnQtrs()* brings the object into an inconsistent

state. When this happens, method *vend()* results in a successful sale although all entered coins have been returned.

4. DATA FLOW ANALYSIS

In our framework, data flow analysis is applied to the class control flow graph (CCFG) to identify pairs of methods that define and use the same instance variable. The CCFG of class *CoinBox* is shown in Figure 4.

Graph nodes represent single-entry, single-exit regions of executable code. Edges represent possible execution branches between code regions. The CCFG of a class consists of a set of Control Flow Graphs (CFGs), one for each method of the class, connected through an additional class node. This node captures the fact that the methods can be invoked in an arbitrary order by other client classes. Each CFG has an *entry node* and an *exit node*, both labeled with the name of the corresponding method. The additional class node is labeled with the name of the class, and has an input edge from the exit node of every method and an output edge to the entry node of every method. For instance, nodes 13 and 17 in Figure 4 are the entry and exit nodes for method *AddQtr()*; the node labeled *CoinBox* is the class node.

Since single CFGs are chained through the additional class node in the CCFG, standard interprocedural data flow analysis can be applied to the methods of the class. In this way, we are able to identify both intraprocedural and interprocedural definition-use dependencies. Infeasible invocation sequences are detected later on by symbolic execution and automated deduction, as we explain below. Interprocedural analysis is required because methods can invoke other methods in the CUT. The CCFG representation of a class and the application of interprocedural data flow analysis to instance variables are derived from existing work [9].

Data flow analysis identifies du-pairs for instance variables. A du-pair for an instance variable v consists of two nodes, s_d and s_u, such that s_d and s_u are both contained in a class C, s_d modifies (writes) the value of v, s_u uses (reads) v's value, and there is a def-clear path from s_d to s_u. We denote the methods containing s_d and s_u by m_d and m_u, and we write $\langle s_d, s_u, v\rangle$ to denote the du-pair. A def-clear path for a du-pair $\langle s_d, s_u, v\rangle$ is a path from s_d to s_u that does not contain additional definitions of v. Note that we implicitly assume that member variables are written (resp., read) only through accessor methods, since the simple case of a direct definition (resp., use) of an attribute can be trivially handled by any driver for the class, and does not the require the application of our technique.

Table 4 shows the du-pairs of class *CoinBox*. These pairs were defined by means of an approximate conservative algorithm that identifies and eliminates those infeasible pairs whose infeasibility can be statically identified [19]. This happens, for example, when the statements in a du-pair are not connected with a def-clear path.

A potential weakness of data flow testing is that sometimes possible faults may not be revealed by the mere execution of du-pairs. This can happen, for instance, when the variable definition contained in a du-pair has no effect on the observable output of the program being tested. This problem

#	variable	m_d (node#)	m_u (node#)	notes
01	curQtrs	CoinBox (4)	addQtr (14a)	
02	curQtrs	CoinBox (4)	addQtr (15)	inf.
03	allowVend	CoinBox (3)	vend (19)	
04	totalQtrs	CoinBox (2)	vend (20a)	
05	curQtrs	returnQtrs (11)	addQtr (14a)	
06	curQtrs	returnQtrs (11)	addQtr (15)	inf.
07	curQtrs	returnQtrs (11)	vend (20a)	
08	curQtrs	addQtr (14b)	addQtr (14a)	
09	curQtrs	addQtr (14b)	addQtr (15)	
10	curQtrs	addQtr (14b)	vend (20a)	
11	allowVend	addQtr (16)	vend (19)	
12	totalQtrs	vend (20b)	vend (20a)	
13	curQtrs	vend (21)	addQtr (14a)	
14	curQtrs	vend (21)	addQtr (15)	inf.
15	curQtrs	vend (21)	vend (20a)	inf.
16	allowVend	vend (22)	vend (19)	
17	curQtrs	CoinBox (4)	vend (20a)	

Table 1: Du-pairs for class *CoinBox*. Node numbers refer to the CCFG of Figure 4. Entry *inf.* indicates infeasible pairs statically identified by data-flow analysis.

is related to the presence, in the code, of chains of definitions and uses that propagate the erroneous value from the faulty definition to the output. To reveal this kind of faults, not only the faulty definition has to be exercised, but the whole chain has to be traversed by a given test case. Düsterwald, Gupta, and Soffa propose a technique for addressing this problem, which is based on the use of software slicing techniques [6]. In our case, the added complexity of such a technique is probably unnecessary because (1) we apply data flow analysis only to the instance variables of the CUT, thus decreasing the chances that chains will occur, and (2) we assume that all instance variables are compared with their expected values at the end of each test case, thus decreasing the chances for an erroneous value to be overlooked.

Data flow analysis does not handle easily dynamic or non-scalar data types; this technology is traditionally considered more suitable for intraprocedural than interprocedural analysis. With our technique, data flow analysis is applied only to the instance variables of a class, which are typically fairly simple. Moreover, data flow analysis is applied to methods within a single class. Thus, we mitigate the most relevant problems of interprocedural analysis, namely aliasing and parameter passing. Finally, data flow analysis can provide useful information even if applied only to a subset of class instance variables, say, by ignoring non-scalar attributes (i.e., instance variables) or by manually identifying their definitions and uses.

5. SYMBOLIC EXECUTION

The du-pairs computed with data flow analysis indicate which methods determine the values of instance variables later used by other methods. To determine a set of test cases, we need to identify for each du-pair a feasible sequence of paths through methods that contain the definition and use of the variable contained in the du-pair, if such an invocation sequence exists. The sequence cannot contain additional definitions of the variable between the definition and the use indicated by the du-pair. To compute such se-

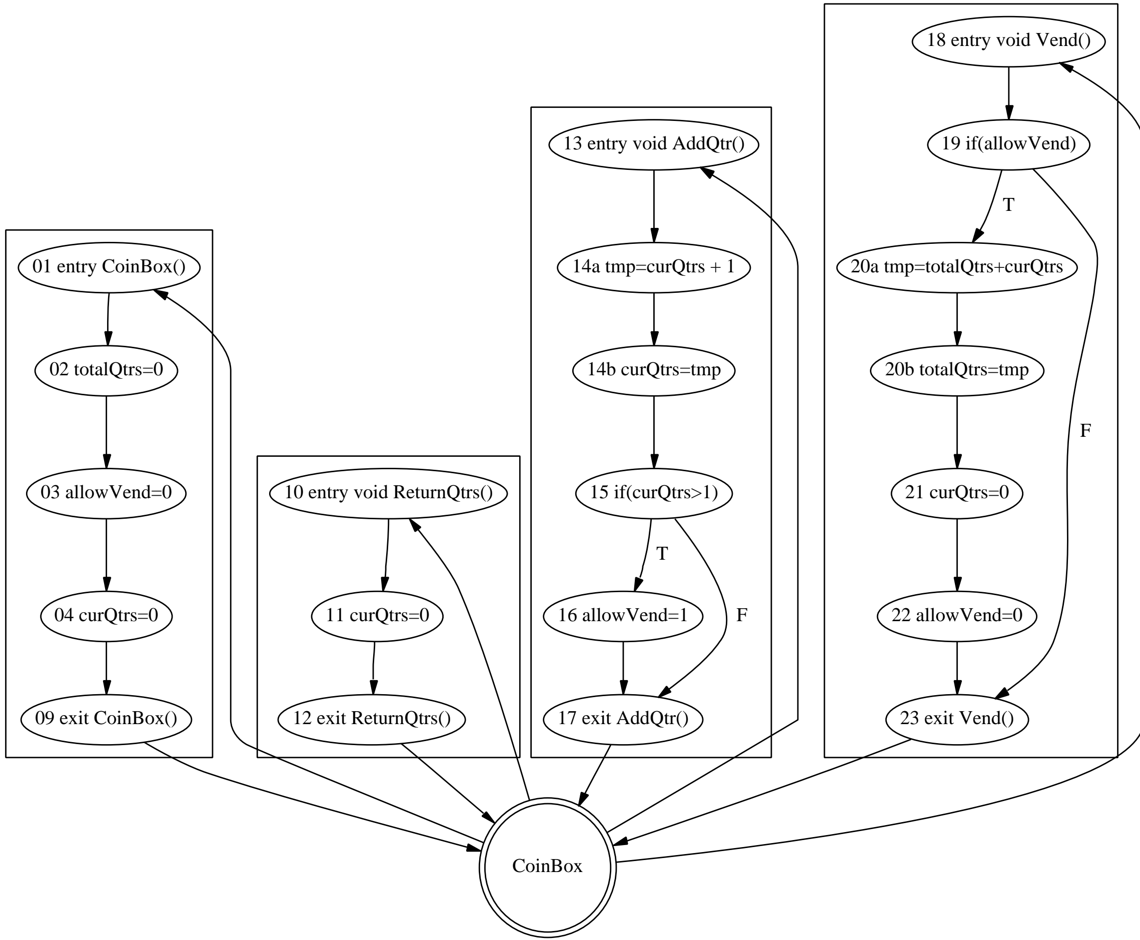

Figure 2: The CCFG for class CoinBox.

quences of execution paths, we need the following data for each path within each method: (1) the conditions associated with the execution of the path, (2) the relationship between input and output values of the method with respect to the path, and (3) the set of variables defined along the path. In addition, for each method we need to evaluate the conditions associated with the execution of paths leading to definitions and uses within the method (i.e., path conditions), as identified by data flow analysis. All the above information is computed through symbolic execution of each method of the CUT.

It is well-known that symbolic execution may fail to compute execution conditions for a path, because of the presence of unsolvable constraints or unbounded loops. We apply symbolic execution to single methods, which are usually procedures with a simple control structure [24]. Moreover, the failure of symbolic execution on a method does not affect the possibility of analyzing other methods, and can be overcome by additional information such as invariants added by the user [3, 4].

In general, path conditions are expressed as a set of propositional formulas involving the values of method parameters and class attributes before method execution, after method execution, or both. The condition associated with the execution of a path, the set of variables defined along the path, and the relationship between input and output values of the method for the given path are expressed as follows:

$$< precondition > \quad \Rightarrow \quad (< attrib >' = < symbExpr >)^* \\ \text{def}=\{< setOfDefAttr >\}$$

Here *precondition* is a predicate on attributes and parameters of the method; this predicate indicates the conditions on the execution of the considered path through the method. In addition, *symbExpr* is a symbolic expression that indicates the value of an attribute computed by executing the path through the method. This expression involves parameters of the method, values associated with the attributes when the method is invoked, and the return value (if there is one). In *symbExpr*, the return value is indicated with the special identifier $return'$. Finally, *setOfDefAttr* indicates the set of attributes defined along the path. The conditions on the execution of definitions and uses within methods are path conditions denoted by PCD and PCU.

Table 5 shows the conditions for executing the paths of all methods of class *CoinBox*, the set of variables defined along each path, and the relationship between input and output values of each method along each path. For sake of read-

ability, we do not explicitly write any postcondition of the kind $v'_i = v_i$ if v_i is not included in the *def* set for the method path. For the considered example all conditions can be computed automatically without requiring users to provide additional information. The PCD and PCU used to compute test cases relevant for the example are shown in Section 6.

CoinBox		
$(true)$ def={totalQtrs, curQtrs, allowVend}	$\Rightarrow$	$totalQtrs' = 0$ $curQtrs' = 0$ $allowVend' = 0$
addQtr		
$(curQtrs > 0)$ def={curQtrs, allowVend}	$\Rightarrow$	$curQtrs' = curQtrs + 1$ $allowVend' = 1$
$(curQtrs == 0)$ def={curQtrs}	$\Rightarrow$	$curQtrs' = 1$
vend		
$(allowVend \neq 0)$ def={totalQtrs, curQtrs, allowVend}	$\Rightarrow$	$totalQtrs' = totalQtrs + curQtrs$ $curQtrs' = 0$ $allowVend' = 0$
$(allowVend == 0)$ def={}	$\Rightarrow$	$allowVend' = 0$
returnQtrs		
$(true)$ def={curQtrs}	$\Rightarrow$	$curQtrs' = 0$

Table 2: execution conditions computed with symbolic execution for class *CoinBox*.

Class *CoinBox* is a server that can be analyzed independently from its clients. In general the computation of class conditions can require some knowledge about the behavior of classes whose methods are used by the CUT. We can provide such information by means of suitable stubs that will be needed for module testing or simply as assertions that specify the behavior of the server classes.

6. GENERATION OF CALL SEQUENCES

Sequence generation uses information produced with symbolic execution to construct sequences of method invocations that exercise the du-pairs identified during data flow analysis. In brief, a method sequence for a du-pair must begin with a class constructor and must end with the method, m_u, that executes the use statement (i.e., s_u) in the du-pair. In addition, the sequence must contain the method, m_d, that executes the definition statement (i.e., s_d) in the du-pair and a def-clear path from s_d to s_u.

In formal terms, given a du-pair $d = \langle s_d, s_u, v \rangle$, a feasible method sequence exercising d is a sequence of methods $(m_1, m_2, \ldots, m_n)$ subject to the following constraints. First, m_1 must be a constructor for the CUT. Second, the execution of m_n must result in the execution of statement s_u, which uses v. Third, when the sequence is executed, statement s_d, which defines variable v, must be executed at least once. Let m_i, with $i \in [1, n]$, be the last method whose execution results in the execution of s_d. It must be the case that $m_i = m_d$. Fourth, for each $j \in [i+1, n]$ the execution of m_j must not contain any additional definitions of v.[2]

[2] In the case of m_n, this constraint can be relaxed by allowing definitions of v, provided that such definitions occur after s_u is executed.

Given a du-pair $d = \langle s_d, s_u, v \rangle$, we generate a method sequence for d in reverse order by starting from the method m_u that contains statement s_u and by applying a set of backward-chained deductions. In brief, our initial goal is PCU, the preconditions upon which method m_u executes s_u. If there is a method m_k, whose postconditions imply PCU, then the method is prepended to the front of the sequence. If no such method is found, we look for a method whose postconditions do not contradict PCU, and we prepend such a method, m_k, to the sequence. The condition to be satisfied by the resulting sequence is defined as follows: (1) The current condition is simplified by eliminating those clauses (if there are any) that are satisfied by m_k's postconditions; (2) the union of the simplified condition and m_k's preconditions is taken; (3) the resulting condition is further simplified, if this is possible.

In general, given a condition P, there can be multiple methods whose postconditions either imply P or do not contradict P. Consequently, we represent the deductive process for a given du-pair $d = \langle s_d, s_u, v \rangle$ as a tree. Each tree node corresponds to a pair consisting of a method and a condition, that is, 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 in conjunctive normal form. The first objective of tree construction is to include a node corresponding to m_d in the tree. The methods corresponding to the nodes on the path from the root to m_d must not execute any definitions of variable v. An additional objective is to include a node corresponding to a class constructor in the subtree rooted at m_d. The deductive process ends when this goal is satisfied.

We explore the tree for a du-pair d in depth-first fashion, starting from the root. As stated above, given a node n_i corresponding to method m_i and condition P_i, the children of n_i correspond to methods whose postconditions do not contradict any of the conjuncts in P_i. If there are no such methods, the node cannot be expanded and tree exploration is continued by considering another node yet to be expanded. Tree exploration may end for one of several reasons:

1. Nodes corresponding to m_d and a class constructor are found in the tree. In this case, the node, n_c, that corresponds to the constructor is a leaf and the path from the root to n_c defines a feasible method sequence for du-pair d in reverse order. The portion of the method sequence from m_d to m_u defines a def-clear execution path because of the way in which we build the tree. Also, the sequence begins with a class constructor and ends with method m_u. Thus, the search for a feasible method sequence is completed successfully.

2. The tree does not contain nodes to be expanded or a feasible method sequence. In this case, the du-pair is deemed infeasible.

3. The depth of the tree reaches a given threshold before a feasible sequence is found. In this case, our analysis is inconclusive and we report this fact to the user.

Our technique for the construction of the tree is based on the use of automated deduction. Even if automated de-

duction techniques, such as constraint solving, may fail in coping with complex expressions, nowadays there are several efficient constraint solvers applicable to large sets of expressions (see, for instance, [20, 23]). In addition, failures of the constraint solver can be overcome by requiring a manual generation of message sequences. To date, we have defined an algorithm for performing the backward chaining and logical inference. We are currently investigating two possible alternatives for the implementation of the algorithm. The first one is based on the use of the PVS reasoning system [20]. The second one is based on the use of the Omega C++ library, which is a complete system for simplifying and verifying Presburger formulas (i.e., linear constraints over integer variables) [21].

We use several heuristics in order to improve the efficiency of our tree construction. First, we reduce 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 a goal of tree construction. 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.

When constructing a method invocation sequence, the case of s_d and s_u belonging to the same method is handled in three possible ways.

1. Statements s_d and s_u are executed in this order along every possible path traversing the method. In this case, the sequence consists of a constructor followed by $m_u = m_d$.
2. Statements s_d and s_u are executed in this order along some path(s) traversing the method (i.e., only under a given precondition). In this case, PCU becomes that precondition, and we follow the general method for tree construction (by taking into account that m_d has already been inserted in the sequence).
3. Statements s_d and s_u are never executed in this order along any paths traversing the method. In this case, we follow the general approach and a feasible sequence (if there are any) will necessarily contain two invocations of $m_u = m_d$.

To illustrate our technique for tree construction, we show the process for automatically building the sequence of invocations for du-pair #7 from Table 4.

variable:	$curQtrs$
definition:	$returnQtrs$, node 11
use:	$Vend$, node 20a
PCD:	$true$
PCU:	$allowVend \neq 0$

The method sequence automatically produced for this du-pair corresponds to the test case that reveals the defect in class *CoinBox*. The set of method sequences automatically generated for all du-pairs of this example can be found in [18].

The tree generation process for du-pair #7 starts with the invocation of method *Vend* that uses the value of variable $curQtrs$, and the corresponding PCU: "$allowVend \neq 0$". Method *Vend* is the root of the tree and its PCU is the associated condition. Examining the postconditions of method invocations reported in Table 5, we can see that it is already possible to add m_d (i.e., method *returnQtrs*) to the tree. In fact, the postcondition of method *returnQtrs* does not contradict condition PCU.

According to the heuristics discussed above, method *returnQtrs* is added to the tree as the only successor of the root (node 1 in Figure 6). The condition of the new node is obtained by removing the conjuncts implied by the postconditions of the new method invocation and by adding the corresponding preconditions. In this case, the condition of the new node is the same as that of its predecessor, because (1) the postconditions of *returnQtrs* do not satisfy any of the conjuncts (only one in this case) composing the root's condition, and (2) method *returnQtrs* does not have any preconditions. Even if node 1's condition implies its predecessor's condition we do not rule it out, since it corresponds to method m_d.

Next, we explore the subtree rooted in node 1. Because this node adds method m_d to the tree, we now need to complete the generated sequence with a subsequence ending in a constructor and satisfying node 1's condition. The constructor's postconditions ("$totalQtrs' = 0; curQtrs' = 0; allowVend' = 0$") contradict the condition associated with the node ("$allowVend \neq 0$"). Thus, we cannot add the constructor as a direct successor of node 1, and we must consider alternative methods. The postconditions of both paths in method *addQtr* and of method *ReturnQtrs* do not contradict the condition of node 1. Thus, we define three successors for node 1, namely nodes 2, 3, and 4. The conditions of the three new nodes are shown in Figure 6. In each case, node conditions are defined as follows.

Node 2: The invocation of method *addQtr*, corresponding to node 2, has postconditions "$totalQtrs' = totalQtrs$; $curQtrs' = curQtrs + 1$; $allowVend' = 1$" and precondition "$curQtrs > 0$". Therefore, we eliminate from the condition of node 2 the conjunct "$allowVend \neq 0$", which is implied by the postconditions of *addQtr*. Although node 1 has no additional conjuncts, we must still add the corresponding precondition of *addQtr* to the conditions of node 2. The resulting condition for node 2 consists of one conjunct, "$curQtrs > 0$".

Node 3: The invocation of method *addQtr*, corresponding to node 3, has postconditions "$totalQtrs' = totalQtrs$; $curQtrs' = 1$" and precondition "$curQtrs = 0$". The postconditions do not satisfy any of the conjuncts (only one in this case) composing the condition associated with node 1. Therefore, this condition is carried over to node 3, in addition to the preconditions of the *addQtr* invocation that we are considering. The resulting condition for node 3 is "$allowVend \neq 0 \wedge curQtrs = 0$".

Node 4: The invocation of method *returnQtrs* corresponding to node 4 has postconditions "$totalQtrs' = totalQtrs$; $curQtrs' = 1$" and no preconditions. As with node 3, node 4's postconditions do not imply any of the conjuncts of the condition associated with node 1. Since no preconditions are added, the resulting condition for node 4 is "$allowVend \neq 0$".

The conditions of nodes 3 and 4 contain (i.e., imply) their parent's condition. Intuitively, this means that the execution of the methods corresponding to nodes 3 and 4 will not help a *Coinbox* instance in reaching a state satisfying the condition that we are considering (i.e., the condition for executing the definition statement in du-pair #7). Thus, we do not explore nodes 3 and 4 any further. We indicate this fact by crossing out these nodes in Figure 6.

Next, we expand node 2, the only open node left. Similar to the case of node 1, we cannot add the constructor as a direct successor of node 2 because one of the constructor's postconditions ("$curQtrs' = 0$") contradicts the condition associated with node 2 ("$curQtrs > 0$"). However, the postconditions of both invocations of method *addQtr* and of one of two possible invocations of method *Vend* do not contradict node 2's condition. Thus, we define node 2 to have children nodes 5, 6, and 7 corresponding to these three invocations. These three nodes are explored next.

The conditions of nodes 5 and 7 imply node 2's condition. Similar to nodes 3 and 4, nodes 5 and 7 are not further explored. However, the condition associated with node 6 ("$curQtrs = 0$") is not contradicted by the constructor's postconditions. Therefore we can add to the tree a node (i.e., node 8) that corresponds to an invocation of the constructor. This ends the construction of the tree. Note that conditions associated with the node that represents the constructor (if there are any) express constraints on the constructor's arguments.

The resulting method sequence is:

CoinBox(), addQtr(), addQtr(), returnQtrs(), vend()

This sequence reveals the *CoinBox* error that we discussed earlier.

7. THE TOOLSET

We are currently developing a toolset that automates our approach to class testing. We intend to use the toolset for extensive experiments aimed at assessing the effectiveness of the approach. Preliminary versions of two key prototypes for data flow analysis and symbolic execution have been completed. The construction of the toolset is taking place within ESPRIT project TWO [3], of which Politecnico di Milano is an active participant. For our experiments we will use mostly software supplied by our industrial partners in project TWO. This software includes aeronautical applications, automatic machine control systems, and automotive control systems.

[3]ESPRIT Project TWO (Test & Warning Office - EP n.28940).

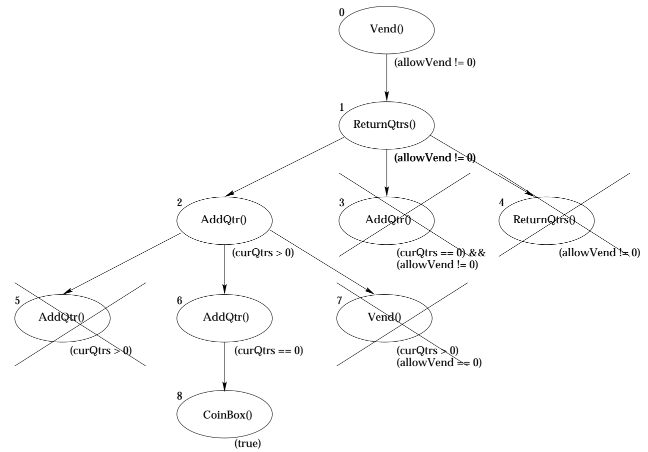

Figure 3: Tree for du-pair #7.

To date, we have conducted a preliminary examination of the code provided by our partners and discovered that most of the code satisfies the assumptions of our three methods for code analysis: about fifty percent of the classes we received use only discrete instance variables; statements involving non-scalar instance variables can be easily represented in terms of definitions and uses of such variables; most execution conditions can be solved with exisiting automated constraint solvers. For these reasons, we expect that test case generation will be completed successfully (i.e., with the generation of method sequences) for at least a meaningful subset of the classes supplied by our partners. In this section we briefly survey the architecture of the toolset, including the current state of development of each component of the toolset.

CCFG generator

The CCFG generator parses the source code of a class and generates the corresponding CCFG. A first prototype based on a commercial C++ parser produced by the Edison Design Group [7] is being tested. The current version of the prototype parses the full Standard C++ language. All constructs but exception handling are also represented in the corresponding CCFG. We are currently investigating CCFG extensions to model the behavior of exception handling constructs with respect to the flow of control.

Data flow analyzer

The data flow analyzer identifies du-pairs for instance variables of the CUT starting from the CCFG output by the CCFG Generator. The current prototype implements a polynomial conservative approximation algorithm [19].

Symbolic executor

The symbolic executor computes conditions for path execution and variable definitions. In particular, it computes the conditions associated with the execution of paths within a method, the relationship between inputs and outputs of a method, the set of variables defined along each path, and the conditions associated with the execution of paths leading to definitions and uses within a method. The current prototype uses a symbolic executor developed by the European consortium LAW for avionics applications. The symbolic executor works for Safer C, the subset of C used for safety critical

applications [10]. We have refined the symbolic executor in order to handle C++ extensions to C.

Sequence generator

This tool will generate method sequences using automated reasoning. The technique is based on the solution of the constraints generated with symbolic execution on paths defined by data flow analysis. We plan to take advantage of the deductive power of existing theorem provers, such as PVS [20, 23], in order to perform some of the deductions required for the generation of method sequences.

8. RELATED WORK

The problem of testing object-oriented software has been addressed by many other authors. However, only a few authors address the specific problem of test case generation for classes. All the existing techniques generate sets of message sequences starting from some kind of description of the CUT. Most techniques generate message sequences from formal class specifications [2, 5, 13, 15, 25]. For example, the ASTOOT system described by Doong and Frankl generates automatically both method sequences and oracles from algebraic specifications [5]. ASTOOT cleverly exploits rewrite rules induced by program specifications in order to manipulate message sequences.

Our techique is code-based rather than specification-based. To our knowledge, the only other technique for generating test cases from code was defined by Kung et al. [14]. That technique generates message sequences for class testing from a state-based model extracted from 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 search of the various state machines.

As with our method, the approach of Kung et al.'s works well when instance variables are scalar and when symbolic execution can be completed successfully [14]. 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 as we explain in Section 2. 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. Our approach takes into account all predicates appearing in methods' preconditions and postconditions when performing sequence generation.

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 [16, 11]. Those approaches start from the tester's knowledge of the source code and seek to generate automatically drivers and stubs for the CUT. 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.

9. CONCLUSIONS

In this paper we reported on an approach for the automatic generation of test cases for testing of classes. The results of our investigations are quite promising. Our approach seems to be quite powerful in that the test cases we generate automatically can detect faults due to the combined effects of method invocations on the object state. Here we explained in detail how we detect one such fault in the *CoinBox* example. An additional advantage when dealing with large classes is that our approach can work incrementally by first generating a subset of test cases; a developer can then provide additional specifications (e.g., preconditions and postconditions) in order 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 since coverage is granted by construction.

The applicability of the approach has been initially evaluated by examining a large set of case studies provided by our industrial partners. In particular, we checked manually the frequency of occurrences of constructs that cannot be analyzed automatically using our framework. Although difficult to quantify, the results of these informal investigations are quite promising: most of the code we examined does not contain constructs that would prevent the application of our techniques. Empirical studies with available toolset components confirm our preliminary observations. We intend to conduct much more extensive experiments as soon as we complete our tool for the automatic generation of method sequences. The goal of our experiments is to evaluate quantitatively the industrial applicability of our framework.

We are also investigating the possibility of extending our framework to the case of classes containing instance variables that are objects. In order to manage object instance variables, we must refine our notion of *def* and *use* sets for these variables. This extension will allow us to transition smoothly from unit to integration testing of classes.

10. REFERENCES

[1] S. Barbey, D. Buchs, and C. Péraire. A theory of specification-based testing for object-oriented software. In *Proceedings of EDCC2 (European Dependable Computing Conference), Taormina (Italy), October 1996*, Lecture Notes in Computer Science 1150, pages 303–320. Springer-Verlag, 1996.

[2] 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.

[3] M. Clerici and L. Mera. Esecuzione simbolica per l'analisi di sistemi critici. Laurea's thesis, Politecnico di Milano, 1999. (in Italian).

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

[5] 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.

[6] E. Düsterwald, R. Gupta, and M. L. Soffa. Rigorous data flow testing through output influences. In *Proceedings of the Second Irvine Software Symposium*, pages 131–145, Irvine (California), March 1992.

[7] Edison Design Group. http://www.edg.com, August 1999.

[8] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental Testing of Object-Oriented Class Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 68–80, Melbourne (Australia), May 1992.

[9] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *2nd ACM-SIGSOFT Symposium on the foundations of software engineering*, pages 154–163, New Orleans, LA (USA), December 1994.

[10] L. Hatton. *Safer C : Developing Software for High-integrity and Safety-critical Systems.* McGraw-Hill, 1995.

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

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

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

[14] 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.

[15] J. D. McGregor. Constructing functional test cases using incrementally defined state machines. In *Proceedings of the 11th International Conference on Testing Computer Software*, Washington DC (USA), June 1994.

[16] 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.

[17] A. Orso. *Integration Testing of Object-Oriented Software.* PhD thesis, Politecnico di Milano, Milano, Italy, 1998.

[18] A. Orso. A framework for testing object-oriented classes. Technical report, Politecnico di Milano, 1999.

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

[20] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[21] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[22] G. Rothermel and M. J. Harrold. Selecting regression tests for object-oriented software. In *International Conference on Software Maintenance (ICSM94)*, pages 14–25, Victoria, British Columbia (Canada), September 1994.

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

[24] A. L. Souter, L. 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 PASTE'99*, Toulouse (France), September 1999.

[25] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *International Conference on Software Maintenance*, pages 302–310, Montréal, Quebec (Canada), September 1993. IEEE Society Press.