A Graphical Class Representation for Integrated Black- and White-Box Testing

Sami Beydeda, Volker Gruhn
University of Dortmund
Computer Science Department
Software Technology
44221 Dortmund, Germany
{sami.beydeda, volker.gruhn}@uni-dortmund.de

Michael Stachorski
adesso AG
Stockholmer Allee 24
44269 Dortmund, Germany
stachorski@adesso.de

Abstract

Although both black- and white-box testing have the same objective, namely detecting faults in a program, they are often conducted separately. In our opinion, the reason is the lack of techniques and tools integrating both strategies, although an integration can substantially decrease testing costs. Specifically, an integrated technique can generate a reduced test suite, as single test cases can cover both specification and implementation at the same time.

This paper proposes a new graphical representation of classes, which can be used for integrated class-level black- and white-box testing. Its distinguishing feature from existing representations is that each method of a class is shown from two perspectives, namely the specification and implementation view. Both the specification of a method and its implementation are represented as control flow graphs, which allows black- and white-box testing by structural techniques. Moreover, a test suite reduction technique has been developed for adjusting white-box test cases to black-box testing.

1 Introduction

This article proposes a new graphical representation for class-level testing. Its distinguishing feature from existing class representations is that it combines the specification and implementation of a class. Each method is represented by two control flow graphs in possibly different abstraction levels, i.e. control flow as specified and control flow as implemented. We refer to the former as the specification view and the latter as the implementation view of a method. Therefore, this representation is called the class specification implementation graph (CSIG) of a class to emphasize the combination of the two different views. Although the method views can differ in abstraction level, the difference does not affect the integration, as the integration is carried out at control flow graph level.

As control flow graphs are used to model specification and implementation, structural techniques, such as coverage criteria-based testing techniques, can be used for test case generation. An important feature of a CSIG is that generated test cases can cover both specification and implementation. The CSIG construction algorithm has been enhanced with a test suite reduction technique to decrease the number of test cases required. The underlying idea of the test suite reduction technique is to adjust test cases generated for white-box testing to also cover the specification. Obviously, a reduced test suite can decrease test execution costs as well as test case maintenance costs.

2 Related work

2.1 Class-level testing

Object-oriented methods and languages can improve the software process and the software produced by that process in many ways [5]. For instance, software can be developed more efficiently by using class libraries which are promoted through object-oriented mechanisms. Additionally, the quality of the developed software can be increased. Software quality depends, among other properties, on the modularity and on the architecture of the software, which can be improved by object-oriented analysis and design. These reasons contributed to the misperception that object-oriented software requires less testing than procedural software and their testing can be done using traditional methods [6, 4].

However, object-oriented methods and languages do not guarantee fewer errors in the developed software [1, 4]. It is commonly accepted that testing of object-oriented software can only be done to some limited extent with traditional methods, because object-oriented mechanisms, namely en-
capsulation, inheritance and polymorphism, introduce problems which cannot be tackled through methods for testing structured software [18, 22, 1]. Furthermore, reuse, which is one of the objectives of the object-oriented paradigm, requires intensive testing, since developed software is more likely to be reused in other contexts.

The process of testing usually starts by validating the smallest testable entities of the software which are referred to as basic units of testing. Although procedures, which are considered as being the basic units of testing in traditional software, correspond to methods in object-oriented languages, classes are considered to be the basic units of testing rather than methods. There are various reasons [23, 1]:

- The use of encapsulation as one of the object-oriented mechanisms hinders observing the state of an object, i.e. the values of its attributes. Since the behavior of a method is usually not only dependent on its inputs but also on the current state of the object, encapsulation complicates the testing of methods.

- Another point which complicates the testing of methods is their close interaction with each other. During information processing within a class, the methods interact with each other through mutual invocations and shared data which is exchanged through the attributes. Separately testing each method would imply the simulation of this interaction by a test driver.

Several techniques have been proposed for object-oriented unit testing, i.e. for testing classes. Some of them are based on finite state machines which can be generated either on the basis of the source code [14] or can be assumed to be available as the specification of the class [24, 16]. It is also possible to generated a required finite state machine from a given formal specification [15]. A given finite state machine can be used, for example, to identify test cases which fulfill some coverage criteria [12]. Other techniques require a formal specification of the class. Examples of techniques basing on formal specifications are given in [9, 13].

2.2 Integrated black- and white-box testing

All techniques thus far mentioned for object-oriented unit testing are either black- or white-box approaches. Few have investigated the integration of black- and white-box techniques. In [7] an approach is presented based on algebraic specifications. Test cases are mainly identified on the basis of specification, while implementation is required to decide whether two objects are observationally equivalent, which provably cannot be decided solely on the basis of the specification.

A different approach to integrating black- and white-box testing can be found in [2]. This approach relies on a graphical representation combining both specification and implementation of the class under test. The underlying combination idea is to identify definition-use pairs [19] (def-use pairs) based on the specification and to represent these def-use pairs in a control flow graph of the implementation by special data flow edges. Having generated this graphical representation of the class, structural testing technique can be used for test case generation. An idea similar to the one in [2] is employed in [3] for integrating the black- and white-box approach. In [3], we have introduced a class representation, called class specification implementation graph (CSIG), having a similar underlying idea for combining black- and white-box testing. The main difference is that the construction process can be automated.

In this article, CSIGs first introduced in [3] are further improved by incorporating a test suite reduction strategy to obtain an integration benefit measurable in terms of costs. The construction strategy takes into account that black- and white-box testing can be carried out to some extent with the same test cases, which allows testing with less test cases.

3 Integration of black- and white-box testing

3.1 Integration benefits

Sufficient testing requires both black- and white-box techniques. Although both tasks are similar in that they have the same objective, namely detecting faults within a program, often black- and white-box techniques are applied separately using different tools. The reason for this is the lack of techniques and tools integrating both tasks. An integrated technique supported by a single tool can have several benefits:

1. The tester has only to be familiar with the concepts underlying one technique and needs training only for one tool.
2. Less maintenance effort is required, since only one tool has to be maintained.
3. Testing can be carried out more efficiently, since an integrated technique can generate test cases covering both the specification and the source code at the same time.

3.2 A demonstrative example

The example consists of a class, called account, which simulates a bank account. This class provides the appropriate methods for making bank account deposits (deposit()) and withdrawals (withdraw()).
Furthermore, it provides methods for paying interest (payInterest()) and for printing bank statements (printBankStatement()).

Figure 1 shows the specification of class account in form of a class state machine (CSM) [12]. In this figure each state of class account is represented by a circle, while each transition is depicted by an arrow leading from its source state to its target state. These transitions are formally specified through 5-tuples (source, target, event, guard, action) below this figure. A transition consists – besides a source and a target state – of an event causing the transition, a predicate guard which has to be fulfilled before the transition can occur, and an action defining operations on the attributes during the transition. There are also two special circles labeled initial and final. These two circles represent the state of an object before its creation and after its destruction, respectively. Thus, they represent states in which the attributes and their values are not defined, meaning that these two states are not concrete states of the object. Furthermore, a CSM also possesses an error state indicating that an error has occurred. Since an error state is not required for further explanations and the example should not be unnecessarily complicated, the error state and the corresponding error transitions are omitted.

The source code of the class account, which is built according to the specification in figure 1, is given in figure 2.

### 3.3 Class specification implementation graph

#### 3.3.1 CSIG constituents

Each method of a class is represented by two control flow graphs in its CSIG. One of them is a control flow graph generated on the basis of the method specification (method specification graph), whereas the other is a control flow graph determined using the method implementation (method implementation graph). For convenience, the two control flow graphs are called method graphs, if they do not have to be distinguished. Thus, the CSIG of a class shows each method from two different perspectives, namely what the method should do and what the method actually does.

The two method graphs of each method are embedded within a frame structure called a class control flow graph frame (CCFG frame). Generally, a class cannot be tested within a frame structure called a control flow graph of the class implementation by special data dependencies. Therefore, method graphs only connect nodes within method specification graphs. The reason for this is explained later.

#### 3.3.2 CSIG construction

**Step 1: Generating method implementation graphs**

The first step consists of generating a control flow graph for each method based on its implementation. A control flow graph consists of nodes, representing statements, and edges, representing control flow among the statements. Moreover, the generated control flow graph can be augmented with data flow information for data flow testing.

**Step 2: Generating a prototype for each specified method**

The second step consists of automatically generating a prototype for each specified method, i.e. for each event type occurring in transitions of the corresponding CSM.

As mentioned before, the main integration idea is similar to that in [2]. In [2], we have proposed to combine black- and white-box class testing by identifying definitions and uses of class attributes within the specification, associating them with each other according to a data flow criterion, and representing the identified def-use pairs within a control flow graph of the class implementation by special data flow edges.

The last step in the above approach requires identifying the statements implementing the definition as well as the use of each def-use pair within the implementation. Generally,
this cannot be fully automated, as it is not possible to decide whether two statements are semantically equivalent.

Therefore, the approach in this article does not require the identification of definitions and uses of attributes within the source code of a method. Instead, a prototype is generated for the corresponding method on the basis of its specification. As the prototype has a predefined structure, identification of definitions and uses of attributes becomes trivial. Note that we do not intend to test a prototype. The prototype is used as a basis for test case identification as it contains the necessary information of the corresponding method's specification.

Generating a method prototype consists of two steps:

1. A prototype is generated for each event of the CSM.

2. Prototypes of events having the same type are combined to one prototype.

These steps are explained using the CSM of class account. During the first step, a prototype is generated for each transition \( t = (source, target, event, guard, action) \) in the form of a nested if-then-else construct as shown in figure 3.

\[
\text{if } \{source\} \\
\quad \text{if } \{guard\} \\
\quad \quad \text{action;}
\quad \text{else} \text{ throw new ErrorStateException();}
\quad \text{else} \text{ throw new ErrorStateException();}
\]

\[\text{Figure 3. The prototype of a transition}\]

source refers to the predicate of the source state. For instance, the predicate of state \( \text{inCredit} \) is defined as \( balance \geq 0 \).

After generating these prototypes, those having the

---

2. Of course, heuristics such as textual comparison can be used to decide whether two statements are probably equivalent. Unfortunately, such heuristics have often limited applicability such as not being applicable to existing software.
same event type are combined. For instance, transitions $t_2$, $t_7$, $t_{14}$, $t_{15}$ and $t_{19}$ share the event $\text{deposit()}$. Their prototypes can be merged to the prototype depicted in figure 4.

```
  if (balance > 0) {
    balance += amount;
  } else if (balance < 0) {
    if (balance + amount < 0) {
      balance += amount;
    } else if (balance + amount >= 0) {
      balance += amount;
    } else {
      throw new ErrorStateException();
    }
  }
```

**Figure 4. The prototype of method deposit()**

**Step 3: Generating method specification graphs**

As the third step, a control flow graph is generated for each generated method prototype and is augmented with data flow edges.

**Step 4: Generating a CCFG frame**

The next step consists of constructing a class control flow graph frame (CCFG frame), which acts as an abstract test driver [11]. A CCFG frame consists of five different nodes, namely $\text{frame entry}$, $\text{frame exit}$, $\text{frame call}$ and $\text{frame return}$. The first two nodes, $\text{frame entry}$ and $\text{frame exit}$, represent the entry and the exit node of the abstract test driver. The other nodes, $\text{frame loop}$, $\text{frame call}$ and $\text{frame return}$, represent call to and return from any public method in an arbitrary sequence. A detailed description of a CCFG frame and a generation algorithm can be found in [11].

**Step 5: Inserting the method graphs into a CCFG frame**

After having generated the CCFG frame, the method graphs are connected to the corresponding nodes of the frame. This means each control flow graph is connected to the $\text{frame call}$ node and the $\text{frame return}$ node of the frame.

**Step 6: Adding data flow edges for black-box testing**

As the last step, a CSIG is augmented with data flow edges for black-box testing. Hong et al. [12] have suggested generating test cases for class-level black-box testing by identifying definitions and uses of each attribute and associating them with each other according to a data flow criterion. Test cases covering these definitions-use pairs are then generated for black-box testing.

Our approach to black-box testing is based on their idea of generating test cases covering a certain data flow criterion on class attributes. However, we use a different technique to obtain the necessary def-use pairs in order to reduce the overall number of test cases and decrease the ef-
forth required for test execution and test case maintenance. Contrary to existing approaches to test suite reduction, such as \[10, 17, 8\], our approach does not assume a fixed set of test requirements, such as given def-use pairs to cover in data flow testing. The basic idea is to address test suite reduction when defining new test requirements, in our case as def-use pairs. For instance, the *all-definitions criterion* \[19\] requires that each definition has to be associated with an arbitrary use. The only constraint is that the use is reachable after the execution of the definition before the variable is re-defined. Therefore, if we already have a test case covering a particular definition and a suitable use, we can associate them with each other as a new test requirement for which we do not need to generate new test cases. Unfortunately, often existing test cases do not appropriately cover definitions and uses to associate them with each other as new test requirements. In these cases, we can either try to adjust an existing test case to retain the total number of test cases or we can generate new test cases.

Generally, a test case for intra-method white-box testing consists of a sequence \(< m_1(x_1), m_2(x_2), \ldots, m_{n-2}(x_{n-2}), m_{n-1}(x_{n-1}), m_n(\top) >\) of method invocations with \(x_i\) being the input of method \(m_i\). In such a sequence, \(m_1\) and \(m_n\) represent the constructor and the destructor of the class, respectively. The sequence \(< m_2(x_2), \ldots, m_{n-2}(x_{n-2}) >\) aims at setting a certain state of the object, whereas \(m_{n-1}(x_{n-1})\) represents the invocation of the method to be tested.

Adjusting a test case \(< m_1(x_1), \ldots, m_{i-1}(x_{i-1}), m_i(x_i), \ldots, m_n(\top) >\) for white-box testing to cover a black-box def-use pair is carried out by inserting a method invocation \(m(x)\) at a particular position \(i\), i.e. between method invocations \(m_{i-1}(x_{i-1})\) and \(m_i(x_i)\). A method invocation \(m(x)\) is inserted at position \(i\), when it contains a use of an attribute which is defined within the sequence \(< m_1(x_1), \ldots, m_{i-1}(x_{i-1}) >\) without being re-defined, or when it contains a definition of an attribute used within the sequence \(< m_i(x_i), \ldots, m_n(\top) >\) without being re-defined before. An important constraint for the method invocation \(m(x)\) is that it may not change the state of the object if it is inserted at a position \(i \leq n-1\), as otherwise it cannot be guaranteed that the test case fulfills the initial white-box test requirement.

In the following description of our approach we assume that definitions and uses of attributes are associated with each other according to the all-definitions criterion. The identification of def-use pairs for black-box testing can be carried out in nine steps:

1. The user is asked to enter the required test cases for white-box testing. Generally, test case generation cannot be automated as it is even not possible to decide whether a program will halt on a particular input. Therefore, test cases have to be generated by a human tester when it is not possible to apply some heuristics. Assume the tester has determined \(t =\langle account(), deposit(1000) >\)\(^3\) as a suitable test case for white-box testing of method \(deposit()\).

2. The generated prototypes are instrumented to track the path taken and the state after each method invocation within a test case. This information obtained after executing a particular test case is called its *test history*.

3. Each prototype is executed with test cases generated by the user for white-box testing of the corresponding method. The result of this step is the test history of each test case. For instance, execution of prototype \(deposit()\) with the above input shows that the definition at line \(a2\) and the uses at line \(a2\) and \(a2\) of attribute \(balance\) are executed (figure 4). Furthermore, we can observe that the object is in state \(inCredit\) after execution.

4. Definitions which have to be associated with a suitable use are identified as the next step. In our case, the definition at line \(a2\) has to be covered.

5. The test history of each test case is analyzed to determine whether the test case, originally generated for white-box testing, also covers a valid def-use pair for black-box testing. Although the test history of the above test case contains a definition and uses of attribute \(balance\), they cannot be associated with each other. The reason is that the use is executed before the definition.

6. Next, a suitable method invocation has to be determined together with the position \(i\) at which it will be inserted. In the case of the all-definitions criterion, the position \(i\) at which the method invocation is inserted is obvious, directly after invoking \(deposit(1000)\). To identify a suitable method invocation, we have to consider the state of the object after invoking \(deposit(1000)\) which is \(inCredit\). Therefore, test histories of other test cases are queried as to whether they contain a method invocation in state \(inCredit\) executing a use of attribute \(balance\). We assume that the user has entered \(t =\langle account(), withdraw(750) >\) as a test case for white-box testing of method \(withdraw()\). Based on its test history we can determine that \(withdraw(750)\) is a suitable method invocation. Thus, \(t =\langle account(), deposit(1000), withdraw(750) >\) can be identified as a test case covering both the specification and the implementation at the same time.

7. The definition of attribute \(balance\) in node \(g18\) is connected with the use in node \(g18\) by an inter-method invocation.
data flow edge. Due to the predefined structure of prototypes and thus specification graphs, localization of both definitions and uses is trivial.

8. Unfortunately, some attribute definitions within the specification cannot be covered by adjusting test cases generated for white-box testing. Therefore, the next step consists of associating these definitions with corresponding uses and generating the appropriate test cases.

9. Finally, the associated definitions-uses pairs are depicted within the CSIG by inter-method data flow edges. These edges are drawn in figure 5 as gray arrows.

4 Tool support

4.1 Using CSIGs for regression testing

The CSIG of a class can be used in various ways for testing. One obvious application is the use of CSIGs for test case generation based on coverage criteria. Another application is the use for selective regression testing.

The objective of regression testing is to provide confidence that modifications have the intended effect and do not affect other unchanged parts of the program adversely. One approach to regression testing relies on a given test suite which has been used to test a prior version of the program. The underlying idea of this approach, called selective regression testing, is to select those test cases from a given test suite covering changed parts of the program. There are various techniques applying different strategies for selecting test cases. A comparison of methods following the selective retest strategy can be found in [20]. Unfortunately, all selective regression testing techniques are white-box techniques to our knowledge, i.e. they do not consider specification changes.

The tool we have implemented for regression testing uses a modified version of the selective regression testing technique proposed by Rothermel et al. [21]. The main idea of their technique is to compare two versions of a class and to analyze the changes between these versions. Based on this analysis, all test cases covering changed statements are selected from a given test set. The two versions consist of a previous version and a changed version, which is to be tested. In the original approach, class control flow graphs (CCFG) are used to compare the two versions. As a CCFG does not consider specification changes, we have adjusted their selection algorithm to be applicable to CSIGs. As the changes made to the algorithm are straightforward and due to space limitations we do not explain them in this paper. However, the use of CSIGs (initial version) for regression testing is explained in [3].

4.2 Sample test process

The sample testing process supported by the tool developed consists of nine activities:

1. The testing process starts by entering the specification of the class to be tested.

2. After defining the specification, first an executable test oracle is generated. An important feature of the tool is generation of executable test oracles. In this context, a test oracle has the objective of determining results and states after each method invocation within a test case. As explained before, we assume a specification of the class under test in the form of a class state machine (CSM). In order to allow executability, transition guards and events are augmented by Java constructs.

3. The CSIG of the class is generated as explained before.

4. Test cases necessary for the testing of the first version of the class have to be entered by the user.

5. Then, the entered test cases are executed and states and results after each method invocation are compared to those obtained by the test oracle. Furthermore, the tool also determines adequacy of the entered test cases. Figure 6 shows the user interface of our test tool. Test cases drawn gray cause a failure of the tested class. Figure 6 shows that the object gets into a wrong state after invoking withdraw(). For instance, the second invocation of withdraw() within the first test case changes the state of the object to inCredit and not, as expected, to blocked.

6. Suppose next that two modifications are made to the class. The first modification concerns its implementation. Generally, valid withdrawal requests are carried out by decreasing the balance of the corresponding account by a particular amount. Unfortunately at line 41, the balance is set to the amount which it should be decreased by, which caused the above failure. This fault is corrected by changing line 41 to 'balance -= amount;'. The other modification concerns the specification. The guards of transactions t13 and t18 have been changed to 'balance-amount < limit & & balance-amount >= 2*limit' to ensure that the balance of an account never becomes less than 2*limit after carrying out a withdrawal request.

7. Similar to the first version of class account, an executable test oracle and the CSIG are generated.

8. The two CSIGs are then compared using the modified selection algorithm. The algorithm selects 6 of the 27
Figure 5. Class specification implementation graph of class account.
(22%) test cases used for testing the previous version for black-box testing and 25 (93%) for white-box testing. The number of test cases selected for white-box testing is quite high as most of them include an invocation of withdraw() and have to be selected for regression testing.

9. The test cases selected are executed. As the fault causing the failure of the first pass has been removed and modifications of the implementation conform to the modification of the specification, no failure can be observed.

Summarizing the results of the example, the main observations are the following:

1. Changes of the specification and the implementation are detected in one pass, i.e. test cases selected cover both types of changes.

2. Although 6 test cases have been selected for black-box testing and 25 for white-box testing, the whole number of test cases is 25, as the white-box test cases selected included the black-box test cases. For instance, the selection algorithm has selected <account(), deposit(1000), withdraw(750)> for testing the guard of transition $t_{18}$, as it covers the definition pair $a_2g_{18}$. The same test case is also selected for testing the changes of method withdraw().

5 Conclusions

We have presented a new graphical representation for class-level testing, called the class specification implementation graph (CSIG). Contrary to existing representation of classes, a CSIG is not restricted to the specification or implementation of a class. It rather combines both by representing them through control flow graphs. Therefore, well known structural testing techniques can be used to obtain test cases for black- and white-box testing. Furthermore, the CSIG generation algorithm has been enhanced with a test suite reduction technique to allow the generation of test cases covering both the specification and implementation, making fewer test cases necessary for adequate testing.

Our future work will focus on two open issues. Firstly, we will conduct empirical studies on selective regression testing using CSIGs. The objective of the empirical studies will be to show whether an integrated approach to testing can really save time and costs. Although initial experience shows that it does, empirical validation is required to underline this claim. A important issue related to the efficiency is effectiveness. We will conduct studies to investigate the ability of the approach to detect faults in comparison to other approaches in literature. Secondly, we will extend CSIGs to also cover dependencies among several classes to allow, for instance, regression testing of object-oriented programs. Existing approaches in literature will be investigated for their applicability for CSIGs.
References


