Toward a Comprehensive and Systematic Methodology for Class Integration Testing

L.C. Briand, Y. Labiche, Y. Wang
Software Quality Engineering Laboratory
Carleton University
Systems and Computer Engineering Department
1125 Colonel By Drive
Ottawa, ON K1S 5B6, Canada
{briand, labiche}@sce.carleton.ca

ABSTRACT

This report is a first attempt towards a comprehensive, systematic methodology for class interface testing in the context of client/server relationships. The proposed approach builds on and combines existing techniques. It first consists in selecting a subset of the method sequences defined for the class testing of the client class, based on an analysis of the interactions between the client and the server methods. Coupling information is then used to determine the conditions, i.e., values for parameters and data members, under which the selected client method sequences are to be executed so as to exercise the interaction. The approach is illustrated by means of an abstract example and its cost-effectiveness is evaluated through two case studies.
# CONTENTS

Abstract ........................................................................................................................................... 1

Contents .......................................................................................................................................... 2

1 Introduction ....................................................................................................................................... 4

2 Related work ................................................................................................................................... 4

3 Class integration Testing .................................................................................................................. 7
   3.1 Interprocedural Control Flow Graph for client methods ............................................................. 9
   3.2 Server method sequences triggered by client methods ................................................................. 11
   3.3 Server method sequences triggered by client method sequences ................................................. 12
   3.4 Removing redundancy ............................................................................................................... 14
   3.5 Test sequences .......................................................................................................................... 18
   3.6 Client interactions with several server classes ............................................................................ 18

4 Coupling-based testing of method interactions ................................................................................. 19
   4.1 Data-Flow Analysis Definitions ............................................................................................... 20
   4.2 Coupling Classification ............................................................................................................. 21
   4.3 Coupling-based integration criteria ............................................................................................ 24
   4.4 Issues ........................................................................................................................................ 24

5 Automation ..................................................................................................................................... 25
   5.1 Reverse-engineering Java source code ....................................................................................... 26
   5.2 Automating the derivation of method sequences ....................................................................... 27
   5.3 Automating the identification of coupling coverage ..................................................................... 28

6 Case studies ..................................................................................................................................... 30
   6.1 Case studies .............................................................................................................................. 31
      6.1.1 Jadvisor ............................................................................................................................... 31
      6.1.2 Linked List .......................................................................................................................... 32
   6.2 Testing Sequences ...................................................................................................................... 33
      6.2.1 Jadvisor ............................................................................................................................... 33
      6.2.2 Linked List .......................................................................................................................... 36
   6.3 Mutants ....................................................................................................................................... 37
      6.3.1 Jadvisor ............................................................................................................................... 38
      6.3.2 Linked List .......................................................................................................................... 39
   6.4 Results ....................................................................................................................................... 40
      6.4.1 Jadvisor ............................................................................................................................... 40
      6.4.2 Linked List .......................................................................................................................... 41
   6.5 Results from the case studies ..................................................................................................... 43

7 Conclusion ....................................................................................................................................... 43

References .......................................................................................................................................... 45

Appendix A  More details on the running example ............................................................................... 48

Appendix B  Instantiation of the tuple metamodel ............................................................................... 50
Appendix C  Algorithms for the generation of tuples ................................................................. 52
C.1  Procedure combineMethodTuples .................................................................................. 52
C.2  Procedure addTuples ..................................................................................................... 53
C.3  Procedure removeRedundancyOne ............................................................................... 54
C.4  Procedure removeRedundancyTwo ............................................................................... 55
C.5  Procedure removeRedundancyThree .......................................................................... 55
C.6  Procedure checkPendingTuples ................................................................................... 56
C.7  Function checkIncludedTuple ....................................................................................... 57

Appendix D  Instantiation of the coupling information metamodel ........................................ 59

Appendix E  Algorithms for coupling based integration test requirements ............................ 62
E.1  Function computeCallCouplingPath .......................................................................... 62
E.2  Function computeParameterCouplingPaths ............................................................... 62
E.3  Function computeSharedAttributeCouplingPaths ...................................................... 64
E.4  Function computeExternalDeviceCouplingPath ......................................................... 65

Appendix F  Jadvisor - Details .............................................................................................. 67

Appendix G  Linked List – Details ........................................................................................ 69

Appendix H  Tuples for Linked List ....................................................................................... 71
1 INTRODUCTION

Integration testing explores how components (i.e., modules, sub-systems) interact with each other, assuming that the components have all passed their local tests [5]. This paper’s focus is on integrating classes, though many of the ideas can be generalized to testing cooperating subsystems, which are simply an encapsulation mechanism to hide class public interfaces in a subsystem from other subsystems. Class integration testing aims at detecting faults that cause inter-class failures [6]. Any class integration strategy must answer the two following questions: (i) In what order will classes be integrated and class interfaces be exercised? (ii) Which test design technique(s) should be used to exercise each interface? Regarding (i), there exist several techniques for the derivation of an optimal test order of classes according to different criteria to minimize, such as the number of stubs [8] or the coupling of class relationships [7]. This report addresses problem (ii) and investigates ways to perform class interface testing, in the context of client/server relationships.

Testing client/server class interactions has received little attention in the literature and existing works only partially address it. A precise but comprehensive methodology, supported by automatable algorithms, is still to be defined and investigated.

Section 2 reports on related work regarding class integration testing. Our approach, which builds on and combines existing techniques is then introduced in Sections 3 and 4, where we focus in turn on the selection of test method sequences and the determination of test data. Section 5 presents how this new approach has been automated in a prototype tool and case studies are described in Section 6. Conclusions are drawn in Section 7.

2 RELATED WORK

A first set of existing integration testing strategies can be described as functional and grey-box since they both concentrate on the methods involved in interactions between classes/objects (without considering the details of the methods though) and associate a functional description with the interacting classes. For instance, in [14], Atomic System Functions (ASF), which
involve system level inputs and outputs and exercise Method/Message paths between objects, drive the integration test of classes. These ASFs correspond to a functional decomposition of the system and are similar to use cases. The objective of the strategy is to execute complete, end-user functionalities, in an incremental manner during integration. Similar strategies based on UML use cases can be found in [6, 17]. They all address the selection and integration of classes which are required in a new release to provide new end-user functionalities in an incremental manner.

Other existing integration strategies are purely white-box. The first one to be mentioned addresses the selection of client method sequences that exercise the interaction between a client class and one of its server classes [4] but does not address the generation of test data. Bashir&Goel’s strategy for class integration testing consists of reusing method sequences defined for the class testing of the client class and re-executes a subset of these sequences, using the actual server classes instead of stubs, as it is normally the case during class testing. Selecting a subset of client sequences to re-test is based on analyzing the server methods called by client methods and devising the sequences of server methods that are triggered by the sequences of client methods. Client method sequences that trigger the same server method sequences are then considered redundant as it is deemed that they exercise the client/server interface in the same way, and all but one are eliminated. After this trimming process, remaining sequences have then to be executed to complete the integration testing of the client/server class pair.

It is worth noting that, though Bashir&Goel describe it in the context of their specific class testing technique, their integration testing strategy is independent from it and can be used in combination with other class testing techniques: e.g., using a state-based representation of the class [3, 20], using constraints based on methods’ contracts [9].

Bashir&Goel’s strategy is interesting since all its steps can be automated and it specifies how a subset of class testing sequences can be reused, thus reducing drastically the effort of class integration testing. However, it suffers major limitations that we address in this work as a first step to specify a comprehensive methodology for class integration testing.

First, the approach does not consider the control flow of both client and server methods. For example, a client method can call server methods several times, possibly not always in the same order (depending on the control flow in the client method). As a consequence, one sequence of
client methods can trigger more than one sequence of server methods, whereas [4] assumes that there is a one to one mapping between them.

Second, when analyzing sequences of server methods triggered by sequences of client methods, the approach in [4] does not account for the instance(s) on which server methods are invoked. For instance, saying that sequence \( m_1.m_2 \) (methods of the server class) is triggered by a given sequence of client methods is not correct when methods \( m_1 \) and \( m_2 \) are called on two different instances of the server class. In this case the sequence of client methods triggers two sequences of server methods: \( m_1 \) and \( m_2 \).

Another recent work consists in identifying data flow interactions between pairs of client methods through attributes that are instances of a server class [16]: the first method in the pair modifies the attribute, and the second uses the attribute. The data-flow analysis in [16] uses an adaptation of Harrold et al.’s notion of intra-class def-use pairs [11]. They only consider attributes (data members) that are object references: a definition/use of an attribute does not only apply to the attribute value, but is also a modification/access of the state of the object referenced by the attribute. Our approach furthers this work in two ways. First, since integration testing refers, in our context, to testing interfaces between classes to assure that they have consistent assumptions and communicate correctly [5], we are also interested in the details of the class methods that interact (e.g., where are located the uses and definitions in the corresponding control-flow graphs). Second, we do not restrict the data-flow analysis to attributes of the client class: We also want to consider interactions through method parameters and local variables (which may not be object references).

Note that the functional and white-box methodologies above are complementary. The former allow you to select a subset of classes to be integrated in the next release so as to be able to deliver end-user functional increments. The latter allows us to test class interactions in this subset in a stepwise manner, once an integration order has been determined [7, 8]. Next, once method sequences are determined, the conditions (values for parameters and data members) under which pairs of methods (client and server methods) are executed remains to be determined. This is the focus of [13] in a procedural programming context. The authors define four coupling criteria that are aimed at triggering the four different coupling types between modules they have
identified (e.g., parameter coupling that refers to parameter passing). These criteria are based on traditional data-flow criteria, and have been extended to address polymorphic relationships [1, 2].

Our approach to class integration testing adapts and extends Bashir&Goel’s strategy (addressing its drawbacks), and combines it with an adaptation of the above coupling criteria, so as to address the issues mentioned above and obtain an automatable, comprehensive class integration testing methodology.

3 CLASS INTEGRATION TESTING

Recall that to test the interaction between pairs of classes during integration testing, we first need to determine an adequate integration order, i.e., an order in which classes will be integrated to form subsystems [7, 8]. This is necessary to determine in which order pairs of classes undergo integration testing. Furthermore, as discussed in the previous section, we assume a class testing suite for the client class in each pair is available. Though the way this test suite was generated does not impact our class integration testing strategy, it is obvious that the more complete this test suite, the more effective integration testing as our integration test sequences will be a subset of the class testing sequences. The motivation at this stage is to reuse the client class test sequences to test the interaction(s) between this client and its server(s). This lowers the cost of integration testing and it is clear that at this stage, while testing class interactions, we usually cannot afford to perform as many tests as during class testing. Furthermore, fully testing the client class again would probably not be cost effective for testing its interaction with a given server class.

Having fulfilled the above pre-requisites, we have, for each pair of interacting classes, to derive the sequences of server methods that the client class method sequences trigger. Our motivation is to determine how the client class exercises, for each test sequence, the server class we are interested in. Recall from Section 2 that, to do so, we have to consider the control flow of client methods and must identify the objects on which calls to server methods are performed, as these two aspects drive the method server sequences actually triggered by client method sequences.
Our strategy first consists in building, for each client method, an annotated control flow graph that accounts for calls between client methods (Section 3.1). Paths in this control flow graph are used to derive the sequences of server methods that are triggered by the corresponding client method (Section 3.2). These sequences are then used to derive the server method sequences that are triggered by client method sequences (Section 3.3). The next steps are then to identify redundant server sequences triggered by different client sequences and reduce (or eliminate) this redundancy by removing some of the server sequences (Section 3.4). Last, we identify the client sequences that trigger the subset of remaining server sequences and retain them to test the interaction between classes (Section 3.5). Note that although a client class can interact with several server classes, we only consider one client/server pair in Sections 3.1 to 3.5. We then discuss the impact of client interactions with more than one server class in Section 3.6. The different steps of our strategy are illustrated using the examples in Figure 1, where class A is the client class and class B is the server class.
public class A {
    private int dA1, dA2, dA3;
    public A() {
        b1 = new B(); b2 = new B();
        dA1 = 0; dA2 = 0; dA3 = 0;
    }
    public int mA1(int i, B b) {
        i++;
        if (i > 0) {
            dA1 = b1.mB1(i);
            dA2 = b1.mB2(i);
        } else {
            dA1 = b2.mB2(dA1);
            dA2 = b2.mB1(dA2);
        }
        dA3 = b.mB3(b1.mB3(dA1));
        return dA3;
    }
    public void mA2(B bb) {
        dA2 = bb.mB1(dA1);
        mA1(dA2, bb);
    }
    public int mA3(int i) {
        B ab1 = new B();
        B ab2 = new B();
        int j = ab1.mB1(i);
        if (j > i) {
            return ab1.mB1(j);
        } else {
            return ab2.mB2(j);
        }
    }
    public int mA4(int i) {
        B ab3 = new B();
        // Class C, not shown, has
        // a static attribute VAR
        // of type int.
        C.VAR = 12
        int j = b1.mB1(i);
        int k = ab3.mB2(dA2);
        if (j > k) {
            return ab3.mB2(j);
        } else {
            return b1.mB1(k);
        }
    }
    public void mA5() {
        B ab = new B();
        dA2 = ab.mB2(2);
        mA1(1, b2);
    }
}

Figure 1 Running example (source code)

3.1 Interprocedural Control Flow Graph for client methods

Each client method is associated with an annotated Control Flow Graph (CFG). The nodes in the CFG are blocks of consecutive statements (two specific nodes being the entry and exit nodes), and edges represent the flow of control between the blocks. Nodes are annotated with the sequence of calls to server methods that are performed in the corresponding blocks of consecutive statements. This is not sufficient in our context as we are interested in sequences of calls performed on individual server instances. We thus have to uniquely identify the server instances, involved in a CFG, on which calls are performed. Symbolic names qualifying these calls are used to that end, and calls can then be described by (symbolic name, method name)
pairs. The requirements in terms of source code reverse engineering, and in particular the way symbolic names are determined so as to uniquely identify the server instances involved, are described in Section 5, where we show that there exist techniques and tools that can be used for that purpose.

In addition, the entry and exit nodes indicate the objects—only those of the type of the server class we are interested in—that are passed to (and returned by) the method. Again, symbolic names are used and correspond to the symbolic names involved in the method annotated CFG. Figure 2.a shows the annotated CFG for method mA1() in class A: mA1() has one formal parameter of type B, the server class, named b. Note that we use attribute and parameter names instead of different symbolic names to improve the clarity of the examples by making the mapping between the code and ICFGs straightforward. This is, however, not possible on real code as object references are assigned to variables/attributes and are changed during execution.

These symbolic names can be of different kinds. First, calls can be performed on parameters and local variables of the method for which we build the CFG. Calls can also be performed on attributes, provided that their class is the server class of interest: the attributes of the current object, but also any accessible attribute of objects other than the current one. This notion of accessible attribute is programming language dependent, and corresponds, in the case Java, to public or protected (when in the same package), either static or not, attributes. Last, method calls can be performed on references returned by other calls.

Since client methods can call each other, thus resulting in more complicated sequences of server methods than those triggered by a single client method, we combine the control-flow of client methods. We then build an Interprocedural Control Flow Graph (ICFG) for each client method. This notion of ICFG has been introduced in [24] in the context of C programs, and was adapted in [11] to an object-oriented context (though for class testing instead of class integration testing). In an ICFG, specific nodes are created to connect the control flow graphs of methods that call each other. Call sites to other methods of the client class are split into call and return nodes. Call nodes are connected to the entry nodes of the methods they invoke, and exit nodes are connected to the return node of all call sites that invoke the method (i.e., if a method is called several times, its CFG appears only once in the ICFG). When their type is the server class, call and return
nodes indicate the actual parameters (using symbolic names) used in the call, and the variable (using a symbolic name) to which the return value of the call is assigned. This is necessary to determine the sequences of server methods executed on the different objects of the server class involved in the interaction.

Figure 2.b shows the ICFG for method mA2() in class A. We can see that during the call to mA1() in mA2(), actual parameter bb is mapped to formal parameter b in method mA1(), implying that the last call to an instance of class B performed in mA1() is performed on parameter bb.

![CFG for method mA1()](image1.png)

![ICFG for method mA2()](image2.png)

Figure 2 Examples of annotated CFG and ICFG from Figure 1

Note that client/server interactions can also consist of calls to static methods of the server class. Such calls, including calls to constructors, are reported in the ICFG, and are all qualified by a unique symbolic name: the server class name.

3.2 Server method sequences triggered by client methods

The annotated ICFG is used to derive the sequences of server methods that are triggered by the corresponding client method. For each client method, paths in the corresponding ICFG are then determined. Note that the number of paths in an ICFG can be infinite because of loops and recursive calls. In this situation, we assume that each loop (recursive call) is bypassed (if possible), taken only once, a representative or average number above 1, and a maximum number of times. Note that this may still produce unfeasible paths (the undecidable path sensitization problem [5, 6]). How unfeasible paths are identified is out of the scope of this report and we may resort, in practice, to getting the user to verify the feasibility of paths.
While determining paths in the ICFG, the mapping between actual parameters and formal parameters (symbolic names) is used to replace any occurrence of a formal parameter name with the corresponding actual parameter name. For instance (Figure 2.b), any occurrence of \( b \) is replaced with \( bb \). The result of these replacements is that the symbolic names in the server sequences no longer correspond to the formal parameter names of the methods involved in the ICFG other than the ones for which we have built the ICFG. Again, we have to ensure that different symbolic names really correspond to different server instances. These issues are discussed in Section 5.

From each path in a client method’s ICFG we extract the possible sequences of calls to server methods, and sort out the sequences corresponding to the different symbolic names involved. These are the sequences of server methods triggered by a single client method. As an example, Table 1 shows the sequences of server methods triggered by method \( mA2() \) (see Figure 1 and Figure 2.b).

<table>
<thead>
<tr>
<th>Server methods called by the two paths in ( mA2() )'s ICFG</th>
<th>Corresponding server method sequences for each symbolic name (server instance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((bb, mA1()), (b1, mA1()), (b1, mA2()), (b1, mA3()), (bb, mA3()))</td>
<td>( bb: mA1().mA3() )</td>
</tr>
<tr>
<td>((b1, mA2()), (bb, mA3()))</td>
<td>( b1: mA1().mA2().mA3() )</td>
</tr>
<tr>
<td>((bb, mA1()), (b2, mA2()), (b2, mA1()), (b1, mA3()), (bb, mA3()))</td>
<td>( bb: mA1().mA3() )</td>
</tr>
<tr>
<td>( b2: mA2().mA1() )</td>
<td>( b1: mA3() )</td>
</tr>
</tbody>
</table>

Table 1 Server method sequences triggered by method \( mA2() \) in Figure 1

### 3.3 Server method sequences triggered by client method sequences

Sequences of server methods triggered by sequences of client methods can now be determined by concatenating the server method sequences, identified in the previous step, based on client method sequences. We also have to ensure, as it has been done previously, that symbolic names uniquely identify server instances.

In particular, the user input is required to decide whether the parameters used in the sequence of client methods reference identical objects. Let us assume that client method sequence \( mA1().mA2().mA5() \) has been defined and run during the class testing of class \( A \), and is
considered for reuse during the integration testing of classes A and B. In this sequence, both methods mA1() and mA5() require a parameter of type B (Figure 1) and we need the tester's help to determine (e.g., based on the semantics of the two methods and purpose of the sequence) whether these two parameters are equal. This is an important practical issue as it determines which server method sequences are possible. However, since we expect class test sequences to be already defined and executed when starting class integration testing, this information is readily available in the test plans and drivers for the client class.

Using the example sequence above (the ICFG for method mA5() in class A as well as the server method sequences triggered by mA5() can be found in Appendix A), if we make the assumption that attributes b1 and b2, and parameters b and bb are referencing distinct instances, then there is a maximum of 2 possible server method sequences, resulting from concatenating 1 possible server sequence in each of the methods mA1(), and mA2() and 2 possible server sequences in mA5() (Table 2: Hypothesis 1). If we make a different assumption, for example that b2=b=bb, in mA1() and mA2(), we now obtain 8 longer server method sequences (Table 2: Hypothesis 2).

<table>
<thead>
<tr>
<th>Hypothesis 1</th>
<th>Client method sequence</th>
<th>Server method sequences triggered by individual client methods on b2.</th>
<th>mA1().mA2().mA5()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mB2().mB1()</td>
<td>mB2().mB1().mB3()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mB2().</td>
<td>mB2().mB1().mB3()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mB3()</td>
<td>mB3()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hypothesis 2</th>
<th>Client method sequence</th>
<th>Server method sequences triggered by individual client methods on b2.</th>
<th>mA1().mA2().mA5()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mB2().mB1().mB3()</td>
<td>mB2().mB1().mB3()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mB1().mB2().mB1().mB3()</td>
<td>mB2().mB1().mB3()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mB1().mB2().mB3()</td>
<td>mB3()</td>
</tr>
</tbody>
</table>

Table 2 Concatenations of Server Sequences on attribute b2 for Client Sequence mA1().mA2().mA5()

As a result, each sequence of client methods triggers zero or more sequences of server methods, these sequences being pairs of the form (symbName, seqServ) where symbName is a symbolic name identifying the object on which the sequence of server methods seqServ is executed. It can also refer to the server class name when seqServ is a sequence of static methods (including constructors) of the server class.
3.4 Removing redundancy

There may exist some redundancy in the way different client method sequences trigger server method sequences. In other words, identical or similar server method sequences may be triggered by different client method sequences. Our objective here is to remove (or reduce) such a redundancy to avoid testing several times identical or similar interactions between a client and a server and make class integration testing more cost-effective.

A preliminary step is to ignore sequences of client methods that do not trigger any sequence of server methods, since we are only interested in the interaction between a client and a server class. Next, when the methods involved in client method sequence S₁ all appear in the same order (in the sequence) in client method sequence S₂, then the server method sequences triggered by S₁ are included in the server method sequences triggered by S₂. In such a situation we ignore the client method sequence (i.e., S₁), which is included in another client method sequence (i.e., S₂). This is the case when client method sequences are derived using Bashir&Goel’s strategy [4]: if two slices contain the exact same transformers, or there is a subset relationship between the set of transformers involved into two different slices, we select only one of the two slices (the one that includes the other, in case of a subset relationship).

Other simplifications are possible. Two different client method sequences may trigger identical or overlapping server method sequences (i.e., there is a subsequence relationship between the triggered sequences), though on different symbolic names. Our goal is to have a systematic strategy to take advantage of those redundancies to decrease the number of client class test sequences to be executed. It would not be realistic to expect that all test cases executed during class testing would be re-run during class integration testing.

Recall that we are in a situation where we have a set of client-server method sequences pairs for each symbolic name, denoted as a tuple: (cseqᵢ, sseqⱼ, nameₖ), where cseqᵢ triggers sseqⱼ on server instance nameₖ. Based on such information, we can define three criteria to remove redundant tuples from our test plan, which result in different testing effort (i.e., number of class test sequences that are re-run), and, as one can expect, in different fault detection capabilities. Those criteria are defined below and illustrated with our running example (Figure 1).
**Criterion 1: Server method sequences** (removal of all the redundant server method sequences)

For any two tuples \((cseq_i, sseq_j, name_k)\) and \((cseq_m, sseq_n, name_l)\), such that \(sseq_j \subseteq sseq_n\), \((cseq_i, sseq_j, name_k)\) can be removed\(^1\), unless one or more methods in \(cseq_i\) do not appear in other tuples.

**Criterion 2: Server method sequences and server instance origin** (removal when redundant server method sequences and same instance kind)

For any two tuples \((cseq_i, sseq_j, name_k)\) and \((cseq_m, sseq_n, name_l)\), such that \(sseq_j \subseteq sseq_n\) and both \(name_k\) and \(name_l\) have the same kind, e.g., an attribute (see Section 3.1), \((cseq_i, sseq_j, name_k)\) can be removed, unless one or more methods in \(cseq_i\) do not appear in other tuples.

**Criterion 3: Server method sequences and server instance** (removal when redundant server method sequences and identical instance name)

For any two tuples \((cseq_i, sseq_j, name_k)\) and \((cseq_m, sseq_n, name_l)\), such that \(sseq_j \subseteq sseq_n\) and \(name_k = name_l\), \((cseq_i, sseq_j, name_k)\) can be removed, unless one or more methods in \(cseq_i\) do not appear in other tuples.

Note that, for all three criteria, every client method that appears in the original client method sequences is present in at least one tuple, and is thus executed at least once. The rationale is that the interaction between the client and the server classes be exercised at least once by every client method.

From these definitions, it is clear that criterion 3 subsumes criterion 2, as, according to the way symbolic names are derived (see the previous sections), when symbolic names involved in two different tuples are equal (criterion 3), they have the same kind (criterion 2). Also, since criterion 1 does not put any constraints on the symbolic names (as opposed to criteria 2 and 3), criteria 2 and 3 subsume criterion 1.

Table 3 lists the tuples (except those containing calls to static methods of the server class) for four of the methods of client class \(A\) in Figure 1, assuming parameters \(b\) and \(bbb\) reference the

\(^1\) \(s_1 \subseteq s_2\) means that \(s_1\) is a subsequence of \(s_2\).
same object. It also indicates the tuples for four client method sequences, assuming these sequences come from the class testing of class A.

<table>
<thead>
<tr>
<th>Client method sequences</th>
<th>Tuples for client method sequences for each server instance symbolic name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(mA1, mB1.mB2.mB3, b1)</td>
<td>(mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1)</td>
</tr>
<tr>
<td>(mA1, mB2.mB1, b2)</td>
<td>(mA1.mA2, mB2.mB1.mB2.mB1, b2)</td>
</tr>
<tr>
<td>(mA1, mB3, b)</td>
<td>(mA1.mA2, mB3.mB1.mB3, b)</td>
</tr>
<tr>
<td>(mA2, mB1.mB3, b)</td>
<td>(mA2.mA1, mB1.mB2.mB3.mB1.mB2.mB3, b1)</td>
</tr>
<tr>
<td>(mA2, mB1.mB2.mB3, b1)</td>
<td>(mA2.mA1, mB2.mB1.mB2.mB1, b2)</td>
</tr>
<tr>
<td>(mA2, mB2.mB1, b2)</td>
<td>(mA2.mA1, mB1.mB3.mB3, b)</td>
</tr>
<tr>
<td>(mA3, mB1.mB1, aB1)</td>
<td>(mA3, mB1.mB1, aB1)</td>
</tr>
<tr>
<td>(mA3, mB1, aB1)</td>
<td>(mA3, mB1, aB1)</td>
</tr>
<tr>
<td>(mA3, mB2, aB2)</td>
<td>(mA3, mB2, aB2)</td>
</tr>
<tr>
<td>(mA4, mB1, b1)</td>
<td>(mA4, mB1, b1)</td>
</tr>
<tr>
<td>(mA4, mB1.mB1, b1)</td>
<td>(mA4, mB1.mB1, b1)</td>
</tr>
<tr>
<td>(mA4, mB2, aB3)</td>
<td>(mA4, mB2, aB3)</td>
</tr>
<tr>
<td>(mA4, mB2.mB2, aB3)</td>
<td>(mA4, mB2.mB2, aB3)</td>
</tr>
</tbody>
</table>

Table 3 Tuples for client methods and client method sequences from Figure 1

Table 4 indicates the sequences that are selected from Table 3 when using the three different redundancy criteria. Five sequences are selected according to redundancy criterion 1. Server sequences for client sequences mA1.mA2 and mA2.mA1 are identical: the former are selected and the latter discarded. Server sequence mB1.mB1 for client sequence mA3 does not appear in previously selected server sequences, and is thus selected too. Similarly, server sequence mB2.mB2 for client sequence mA4 is selected. Then, the other server sequences for client sequences mA3 and mA4 all appear in already selected sequences and are thus discarded, thus resulting in the five sequences in Table 4. Criterion 2 would yield a larger subset of tuples. Most of the selections described above apply for criterion 2, except that since criterion 2 accounts for the different kinds of symbolic names, tuples (mA3, mB1.mB1, aB1) and (mA4, mB1.mB1, b1) are different (different symbolic name kinds) though server sequences are identical. Tuple (mA4, mB1.mB1, b1) is then selected in addition to the ones selected for criterion 1, thus resulting in the six sequences in Table 4 (the new tuple is underlined). Last, criterion 3 would yield the largest tuple subset: tuple (mA3, mB2, aB2) is added to the tuples selected according to criterion 2 since criterion 3 accounts for distinct symbolic names and none of the sequences selected by criterion 2 contain a call to mB2 on aB2 (again, the new tuple is underlined in Table 4).
Assuming that the call at line 44 in \texttt{mA4()} is erroneously coded as \texttt{b1.mB1(j)}, instead of \texttt{b1.mB1(k)}, then this fault would be detected with criteria 2 and 3: tuple \((\texttt{mA4, mB1.mB1, b1})\). This would not be the case with criterion 1 though, thus illustrating why criterion 2 strengthens criterion 1. Similarly, a fault in the call at line 31 in \texttt{mA3()}'s implementation would not be detected by the tuples produced according to criteria 1 and 2, but would be detected by the tuples produced according to criterion 3: tuple \((\texttt{mA3, mB2, aB2})\).

<table>
<thead>
<tr>
<th>Redundancy criterion 1</th>
<th>Redundancy criterion 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\texttt{mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1}))</td>
<td>((\texttt{mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3.mB1.mB2.mB3, b1}))</td>
</tr>
<tr>
<td>((\texttt{mA1.mA2, mB2.mB1.mB2.mB1, b2}))</td>
<td>((\texttt{mA1.mA2, mB2.mB1.mB2.mB1.mB2.mB3, b2}))</td>
</tr>
<tr>
<td>((\texttt{mA1.mA2, mB3.mB1.mB3, b}))</td>
<td>((\texttt{mA1.mA2, mB3.mB1.mB3, b}))</td>
</tr>
<tr>
<td>((\texttt{mA3, mB1.mB1, aB1}))</td>
<td>((\texttt{mA3, mB1.mB1, aB1}))</td>
</tr>
<tr>
<td>((\texttt{mA4, mB2.mB2, aB3}))</td>
<td>((\texttt{mA4, mB2.mB2, aB3}))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Redundancy criterion 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\texttt{mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1}))</td>
</tr>
<tr>
<td>((\texttt{mA1.mA2, mB2.mB1.mB2.mB1, b2}))</td>
</tr>
<tr>
<td>((\texttt{mA1.mA2, mB3.mB1.mB3, b}))</td>
</tr>
<tr>
<td>((\texttt{mA3, mB1.mB1, aB1}))</td>
</tr>
<tr>
<td>((\texttt{mA3, mB2, aB2}))</td>
</tr>
<tr>
<td>((\texttt{mA4, mB1.mB1, b1}))</td>
</tr>
<tr>
<td>((\texttt{mA4, mB2.mB2, aB3}))</td>
</tr>
</tbody>
</table>

**Table 4 Applying the three redundancy criteria from Table 3**

An additional simplification can be performed. Given two tuples generated from the same client method sequence \((\texttt{cseq, sseq}_j, \texttt{name}_1)\) and \((\texttt{cseq, sseq}_m, \texttt{name}_2)\), if \texttt{name}_1 and \texttt{name}_2 are accessed in the same control flow path in the \texttt{cseq} methods' ICFG, then either of the two pairs can be removed. This simplification does not reduce the fault detection effectiveness, and can be used independently from the selected redundancy criterion. It requires to examine each ICFG control flow path and decide which tuple is to be removed, a task that can be easily automated.

Continuing with our working example, and assuming we were using criterion 1, tuple \((\texttt{mA1.mA2, mB3.mB1.mB3, b})\) can be removed as it is executed in the same control flow path as \((\texttt{mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1})\), and only four tuples remain to be tested: \((\texttt{mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1})\), \((\texttt{mA1.mA2, mB2.mB1.mB2.mB1, b2})\), \((\texttt{mA3, mB1.mB1, aB1})\), and \((\texttt{mA4, mB2.mB2, aB3})\).
3.5 Test sequences

At this stage, the test sequences to be re-run are the method sequences on the client class in the tuples we obtained in the previous simplification stages. Each one of these test sequences may then correspond to one or several test cases, depending on the strategy used to exercise the interactions between client and server methods (e.g., coupling information as discussed in Section 4). Going back to our example, assuming we used criterion 1 and removed a sequence based on the analysis of the ICFG’s, we obtain the following client method sequences: mA1.mA2, mA2.mA1, mA3, and mA4. We have not reduced the number of client sequences to be tested, but we have reduced the number of tuples to be tested: from 13 (Table 3) to five (excluding (mA1.mA2, mB3.mB1.mB3, b)). Therefore, regardless of the way we set parameters, this will result into less test cases.

3.6 Client interactions with several server classes

In the previous sections we assumed client/server pairs of interacting classes. However, class interactions in object-oriented systems are not restricted to client/server pairs: A client class can use the services of different server classes at the same time. In such a situation the previous approach can be used for each client/server pair, where server, is one the server classes whose services are used by the client class. The union of all the tuples (cseq, sseq, name) thus derived for each pair provides the set of sequences that have to be executed in order to test the interactions between the client class and its server classes. There may exist some redundancy in the way different client method sequences trigger method sequences in the server classes.

For instance, a given client method sequence can trigger, in one of its control flow paths, method sequences on objects of two (or more) server classes. This results in two tuples of the form (cseq, c1.seq1, name1) and (cseq, c2.seq2, name2) where cseq is the client method sequence that triggers c1.seq1 (respectively c2.seq2), that is sequence seq1 (respectively seq2) on server class c1 (respectively c2), on symbolic name name1 (respectively name2) that is an instance of server class c1 (respectively c2). In such a situation, executing the client method sequence ICFG’s control flow path only once is sufficient to trigger the two server method sequences. Removing
(or reducing) such a redundancy thus requires that the control flows actually triggered by tuples in client method sequences be compared. This will be the subject of future work.

Another situation involving more than one server class is worth mentioning. Because of inheritance, even if one client/server pair is considered, several classes may be involved when the server class is a parent class. Indeed, the interaction between the client and its server is supposed to be consistent when an instance of the server class is replaced by an instance of any one of its child classes [15]. In such a situation, the interaction between the client and all its potential servers must be tested. However, the tuples defined between the client and the parent class (server) can be reused. Furthermore, if the multiplicity of the client/server relationship is 1 to many (more than one instance of the server class might be used by one client instance), the tester has to decide (1) how many server instances have to be instantiated (e.g., 0, 1, a reasonable number above 1 and a maximum number), and (2) what are the classes from which these instances are created (e.g., one instance from each potential server class). Solutions to these issues, likely involving the user’s input, will be investigated in future work.

4 COUPLING-BASED TESTING OF METHOD INTERACTIONS

In the previous sections, our goal was to fully exercise the interactions between two classes, by carefully selecting client class test sequences. In this section, we go one level deeper and propose a procedure to fully exercise the interaction between each pair of interacting methods in an integration client test sequence. The method presented below makes use of the notion of coupling path between two subroutines [13] and adapts it to an object-oriented context to ensure that paths between definitions and uses of parameters, attributes, and files are covered during testing in order to exercise the interactions between a caller method and a callee method. In other words, we are combining two types of techniques to fully exercise the interface between two interacting classes. We provide a short overview of basic data-flow analysis definitions (Section 4.1), propose a coupling type classification (Section 4.2) and coupling-based integration criteria (Section 4.3). All the definitions are illustrated using our running example in Figure 1. Important issues are then discussed in Section 4.4.
4.1 Data-Flow Analysis Definitions

Let us start with basic data flow analysis definitions in a non-OO context. A definition is a location in a subroutine where a variable’s value is stored into memory. A use is a location where a variable’s value is accessed. A computation use (C-use) is a node where a variable is used in a computation, as a functional parameter or in an output statement. A predicate use (P-use) is a node where a variable is used in a conditional expression. A new type of use, namely, indirect-use (I-use), which is defined in [22], is also applied in [13]. An indirect-use occurs when a variable has a C-use first to define another variable, and then the defined variable is later used in a conditional expression. A call-site is a node in a caller’s control flow graph from which a callee is called. A return-site is a node from which some value is returned to the caller.

From Figure 1, line 11 in class A (i.e., method mA1()) is a call-site to method mB1() in class B. Lines 13 and 18 are return-sites from method mB2() (class B). In method mB2() (class B), lines 10 and 11 are a C-use and P-use for parameter j and attribute dB2, respectively, thus resulting in an I-use of j at line 11. Line 10 is also a definition for attribute dB2.

In [13] the authors provide a number of basic definitions regarding coupling that we adapt and refine to our context based on ICFGs. If there is a node in a caller/callee subroutine’s ICFG\(^2\) that contains a definition and there is at least one execution path between this definition and one of its uses in a node in this ICFG, we say this definition is a coupling-def as far as that caller-callee pair is concerned. Similarly, a coupling-use is a node in a caller/callee subroutine’s ICFG that contains a use so that there is at least one execution path between a definition in a node of the caller subroutine’s ICFG and this use.

A coupling path is a control flow path between two subroutines from a coupling-def corresponding to an actual parameter in a caller/callee subroutine to a coupling-use of the corresponding formal parameter in the callee/caller unit. For the in parameters, a coupling path is from the last definition in the caller before calling the callee (last-def-before-call) to the first use of the parameter in the callee (first-use-in-callee). For the out parameters, the coupling path is from the last definition of the parameter before returning control to the caller (last-def-before-

\(^2\) In this ICFG, the call site to the callee subroutine is split into call and return nodes in the caller’s CFG (see Section 3.1).
return) to its first use in the caller after the completion of the call (first-use-after-call). For instance (Figure 1), line 9 in class A (method mA1()) is a first-def-before-call for i, a call being located at line 11 (call to method mB1() in class B), and line 5 in class B (method mB1()) is the first-use-in-callee for formal parameter j, corresponding to actual parameter i in method mA1().

Offutt et al presented four coupling based integration test coverage criteria\(^3\) between subroutines in [13]:

- **Call-coupling** requires that the test cases cover all call-sites of the callee method in the caller method. This is the weakest criterion.

- **All-coupling-defs** requires that, for each coupling-def of a variable in the caller, the test cases cover at least one coupling path to at least one reachable coupling-use.

- **All-coupling-uses** requires that, for each coupling-def of a variable in the caller, the test cases contains at least one coupling path to each reachable coupling-use.

- **All-coupling-paths** requires that the test cases covers all coupling paths from each coupling-def of a variable to all reachable coupling-uses.

### 4.2 Coupling Classification

Given the above definitions, we present a modified coupling type classification, based on [13] and tailored to Object – Oriented programming languages.

**Call Coupling**

There is a **Call coupling** between mA and mB when mA invokes mB.

**Parameter coupling**

There is a **Parameter coupling** between mA and mB when mA invokes mB, and mB has (at least) one parameter (or return value).

---

\(^3\) Note when these criteria involve control flow paths and there is a loop along these paths, then typical techniques for testing loops can be applied to make the number of paths finite.
**Shared Attribute coupling**

There is a *Shared Attribute coupling* between \( m_A \) and \( m_B \) when \( m_A \) invokes \( m_B \) and when they both reference one or several attribute of any class such that \( m_A \) and \( m_B \) update and read that attribute, respectively. We define *shared-def* and *shared-use* nodes as ICFG nodes where a shared attribute is defined and read, respectively. Such attributes can be static, public (though this is considered poor practice), or accessible through specific mechanisms such as Friend classes in C++. This type is an adaptation of *Shared data coupling* defined in [13].

**External device coupling**

There is an *External device coupling* between \( m_A \) and \( m_B \) when \( m_A \) invokes \( m_B \) and when they both access an external medium (e.g., file) such that \( m_A \) and \( m_B \) update and read that medium, respectively. We define *external-def* and *external-use* nodes as ICFG nodes where an external device is updated and read, respectively.

Note that following these definitions, we do not consider definition-uses when the definition and the use are both either in the client method or in the server method, as this should be the focus of the unit testing of the client class in the former case (only a stub of the server class is then required), or the unit testing of the server class in the latter case. In other words, we do not consider the definition of a reference whose type is the server class (e.g., the assignment of a new reference to a local variable) and its use in a following call on that reference\(^4\): former case. We do not consider interactions through shared attributes\(^5\) when a definition is performed in a call to the server object (i.e., the method modifies an attribute of the server object) and the use is also in a call to the server object\(^6\) (i.e., the method uses the attribute modified in the previous method): latter case.

The next step is to define the notion of coupling path, for each coupling type. A coupling path is a path in the ICFG of the caller and callee methods that exercises a particular pair of coupling definition and use. As in [13], those paths will constitute the basic coverage elements that will be

\(^4\) Definition of local variable \( a_B \) at line 47 in class \( A \) and its use through a call to \( m_B2() \) at line 48 (Figure 1).

\(^5\) This corresponds to the definition-use pairs of references mentioned in [16], where the definition is a call to a client method modifier (i.e., a method that modifies the state of the object on which it is called) and the use is a call to a client method inspector (i.e., a method that uses the state of the object on which it is called).

\(^6\) Definition of attribute \( d_B1 \) at line 5 in class \( B \) as a result of call to \( m_B1() \) at line 11 in class \( A \), and its use at line 12 in class \( B \) as a result of call to \( m_B2() \) at line 12 in class \( A \) (Figure 1).
used to elaborate our testing strategy. We then define four different types of coupling paths, corresponding to the four previous coupling types.

**Call coupling paths**

For a given method invocation on an instance of the server class, a call coupling path begins with the corresponding call-site and ends with a return-site.

**Parameter coupling paths**

For each actual in parameter involved in an invocation, a parameter coupling path starts with a last-def-before-call (of the actual parameter), continues through a call-site, and ends with a first-use-in-callee (of the formal parameter corresponding to the actual parameter). For each actual out parameter (and return parameter), a coupling path starts with a last-def-before-return, continues through a return-site, and ends with a first-use-after-call (of the returned parameter value).

**Shared Attribute coupling paths**

For each attribute shared by a caller and callee method, the shared attribute coupling path begins with a shared-def in the caller/callee, continues through a call-site/return-site, and ends with a shared-use in the callee/caller.

**External device coupling path**

For each external device shared by a caller and callee method, the shared external device coupling path begins with a external-def in the caller/callee, continues through a call-site/return-site, and ends with a external-use in the callee/caller.

Terms coupling-def and coupling-use can then be defined as any one of the “-def” (last-def-before-call, last-def-before-return, shared-def, external-def) and “-use” (first-use-in-callee, first-use-after-call, shared-use, external-use) above, respectively.

Figure 1 allows us to illustrate these different coupling paths. Considering, in method mA1 (class A), the call to method mB2 (class B) (line 12 in class A), there is a call coupling path starting at line 12 in class A (call-site) and ending at line 13 in class B (return-site). Other coupling paths for the same call includes the loop in line 16 and end at line 18 (return-site). A parameter coupling path involving actual parameter i during the call to method mB2 () in method
mA1() starts at line 9 (last-def-before-call), continues through line 12 (call-site), and ends at line 10 in class B (first-use-in-callee). Attribute VAR in class C is shared by methods mA4() and mB1() in classes A (lines 38 and 39) and B (line 6) respectively. The corresponding shared attribute coupling path starts at line 38 (shared-def) and continues through line 39 (call-site) in class A, and ends at line 6 (shared-use) in class B. Two external device coupling paths appear in the example: external-def at lines 55 and 57 and call-site at line 59 in class A, call to mB4() at line 36 (in mB5()) and external-use at line 27 in class B.

4.3 Coupling-based integration criteria

Offutt et al. presented four coupling-based integration test criteria in [13]. We adapted them here to define strategies, in increasing cost order, for covering all coupling paths defined above3:

- **All-Call-Sites** requires that the test cases cover all call-sites in the caller.

- **All-coupling-defs** requires for each coupling-def that the test cases contain at least one coupling path to at least one reachable coupling-use.

- **All-coupling-uses** requires that for each coupling-def that the test cases contain at least one coupling path to each reachable coupling-use.

- **All-coupling-paths** requires that the test cases cover all coupling paths from the coupling-def to all reachable coupling-uses.

4.4 Issues

The above definitions allow us to exercise client/server interactions, in a systematic way, by means of coupling paths: there is a coupling-def in mA() (respectively mB()) and a coupling-use in mB() (respectively mA()), and methods mA() and mB() are in the same tuple (mA() and mB() being in the client and server sequences, respectively). However, when mA() and mB() are in two different tuples, or one of the two methods is not in any of the tuples, the coupling cannot be exercised, as it is not part of the original class testing sequences. Figure 1 illustrates the former situation, assuming method mA6() is not in the sequences, reused from the class testing of class A, where methods mA1() to mA5() appear. In other words, one of the sequences reused from the
class testing of class $A$ contains only method $mA6()$. In such a case, $mA6()$ contains a definition of shared attribute $VAR$ (class $C$) and method $mB1()$ in class $B$ contains a use of that shared attribute, thus resulting in a potential coupling between $mA6()$ and $mB1()$. But methods $mA6()$ and $mB1()$ are not together in any of the tuples and that coupling cannot be exercised by integration test cases. Such situations can happen when two client classes $A$ and $A'$ use the services of a same server class $B$. Methods in $A$ and $A'$ can interact (and exhibit coupling) through the state of $B$ or external devices (e.g., files) it accesses.

In order to exercise these couplings, a first solution is to ask the user to explicitly add sequence(s) so that the generated tuple(s) contain(s) both methods. Another solution is to wait for a subsequent integration step in the integration plan in which class $C$, a client class of both $A$ and $A'$ in the example above, is integrated right after $A$ and $A'$. Such an integration step would allow us to exercise couplings between the methods of $A$ and $A'$ that were not exercised during their own integration.

## 5 AUTOMATION

In this section we present how the new approach presented in Sections 3 and 4 has been implemented in a prototype tool. We first summarize the requirements in terms of source code reverse engineering of the approach (Section 5.1): Recall that we need to derive ICFGs and annotate them with symbolic names, and that we have to identify coupling paths. We show that there exist techniques and tools that can be used for that purpose, and we do not further discuss this issue in this report, though future work will address the integration of these techniques/tools into our prototype tool. We then identify the information relevant to the two different steps of the approach: deriving tuples for client method sequences and using redundancy criteria (Section 5.2), and applying the different coupling-based criteria (Section 5.3). This information, provided by reverse-engineering techniques is modeled by means of UML class diagrams (or metamodels). These metamodels first help us define the requirements in terms of information we need to retrieve from reverse-engineering. In turn, the algorithms that implement the approach (they are provided in Appendices) can be expressed in terms of classes and relationships in these metamodels. As a consequence, they make use of the Object Constraint Language (OCL) [26]. OCL in this context helps understand the algorithms as, no additional data structure is required.
(the metamodel describes what the structure is), and the mapping between the algorithms and the metamodel is straightforward. Last, we limit our description of the prototype tool to these two metamodels (and the corresponding algorithms) as they form the core subsystems of the tool.

5.1 Reverse-engineering Java source code

One important aspect of the approach is the amount of information that needs to be reverse-engineered. First, ICFGs must be built for each method in the client class. There exist approaches to gather control-flow, local data-flow, and symbol table information for Java programs [12]. In addition, we need information on the (possibly different) server instances that are used in these ICFGs, so as to assign symbolic names. Here again some techniques exist and can be reused. For instance, [18, 19, 25] propose techniques, referred to as points-to analysis, to determine the set of objects whose addresses may be stored in reference variables (e.g., method local variables) and attributes. These techniques can be applied in our context to ICFGs, thus identifying objects used in ICFGs’ paths. Once objects have been identified, choosing a unique symbolic name for each one is straightforward.

However, we have to ensure that, when applying these techniques the symbolic names produced are different if and only if they represent two different server instances. Let us take two example situations that illustrate why the choice of symbolic names may be difficult. First, two different symbolic names may correspond to the same attribute name, parameter name, local variable name, or return value name, as indicated in the source code. For instance, two calls on the same attribute name but in two different statements in the same method (same CFG) require two different symbolic names when the attribute is assigned a new value in a statement between the two calls, as two different objects are then involved. Also, when a specific CFG is used several times in an ICFG (i.e., different calls to the corresponding method are performed by the method for which we build the ICFG), symbolic names corresponding to local variables in that CFG must be renamed when we derive sequences of server methods as they (may) correspond to different objects. Also, a given attribute name may correspond to different symbolic names in different CFG, as it is not possible when we build the CFGs to know whether the different names represent the same object.
As for coupling paths, tools have been built to measure coupling coverage of test cases: e.g., the coverage of the coupling criteria defined in [13] can be determined with the tool described in [21] in the case of Java programs. Again, such a tool can be reused but needs to be adapted since our definitions of coupling differ from the ones in [13].

The requirements of our class integration testing approach in terms of reverse-engineered information do seem realistic, as there exist techniques and tools that can be reused or adapted. As a consequence, we do not further discuss this issue in this report, though future work will address the integration of these techniques/tools into our prototype tool.

5.2 Automating the derivation of method sequences

Figure 3 shows the metamodel for the derivation of tuples. The first step of the approach consists in reusing client method sequences from the class testing of the client class: a TestRequirement instance is created along with clientSetNum client method sequences (qualifier sequenceIndex is used to access these clientSetNum sequences). The reverse engineering step provides a set of original tuples for each client method in these sequences (role name originalTuples). Each tuple consists in a client method sequence (role name clientSeq), a server method sequence (role name serverSeq), and a symbolic name. A symbolic name can have four different kinds, as defined in Section 3.1, and symbolic names are uniquely identified by their name. Then, applying any of the three redundancy criteria produces a new TestRequirement instance that is associated with a set of tuples (aggregation between TestRequirement and Tuple).
Note the user inputs, e.g., deciding whether actual parameters in a client method sequence are equal or not, are not part of our descriptions in this section, as we focus on the class diagram that models tuples, and its use by redundancy criteria.

Appendix B provides examples of instantiations of this metamodel for our running example (Figure 1, Table 3 and Table 4), and the algorithms that implement the three redundancy criteria from instances of this metamodel can be found in Appendix C.

5.3 Automating the identification of coupling coverage

Figure 4 is the metamodel (class diagram) for coupling information between a caller and callee methods. A Method may have several CallSite (0 or more), i.e., a node in its control flow graph where a method of the class is invoked, and several ReturnSite where the callee returns control to the caller (at least one return site corresponding to the last closing curly bracket). Since a CallSite corresponds to a method call, it is associated with one Method, called the callee. The data flow information used for coupling-based integration testing criteria concerns definitions and/or uses before/after call-sites (e.g., last definition before call), return-sites (e.g., last definition before return) and method declaration (e.g., first use in callee). This is modeled by means of class Site (parent class for CallSite, ReturnSite and Method) and its associations with class UseDef (role names lastDefs and firstUses). Each definition or use is for an InteractingElement which can be either a SharedAttribute, an ExternalDevice, a FormalParam or an ActualParam (each ActualParam being associated with the corresponding FormalParam). Note that the fact that a callee method returns a value to a caller is modeled by specific instances of classes FormalParam and ActualParam in the callee and caller methods respectively. Last class Path is used by the algorithms that compute the test requirements according to the different coupling-based criteria we defined in Section 4.3: e.g., a parameter coupling path starts with a last-def-before-call, continues through a call-site and ends at a first-use-in-callee.

It is worth noting that abstracting associations to definitions and uses from classes CallSite, ReturnSite and Method to class Site, which greatly simplifies the metamodel, implies that

---

7 since the return parameter with its corresponding return value can be viewed as an out parameter Ada???
constraints exist between classes ClassSite, ReturnSite, Method and classes UseDef and InteractingElement (and its subclasses). For instance, role name lastDefs does not make sense for any instance of class Method, since it refers to “last definitions before” and, we are only interested in the first uses (i.e., first use in callee) in the case of a method declaration. Also, it does not make sense for a CallSite to have definitions and uses of formal parameters since in such a case we are only interested in the actual parameters used in the call, and the corresponding formal parameters in the called method.

These constraints are modeled below using the Object Constraint Language [26]:

- Instances of class Method cannot have definitions (i.e., last definition before):
  \[\text{Method.allInstances}\rightarrow\forall (m:\text{Method}|m.\text{lastDefs}\rightarrow\text{Empty})\]

- Instances of class ReturnSite cannot have uses (i.e., first use after):
  \[\text{ReturnSite.allInstances}\rightarrow\forall (rs:\text{ReturnSite}|rs.\text{firstUses}\rightarrow\text{Empty})\]

- Instances of CallSite cannot have definitions or uses of formal parameters:
  \[\text{CallSite.allInstances}\rightarrow\forall (cs:\text{CallSite}|cs.\text{firstUses.}\text{interactingElement}
    \rightarrow\text{select}(ie:\text{InteractingElement}|ie.\text{oclType}\rightarrow\text{FormalParam})\rightarrow\text{Empty})\]
  \[\text{CallSite.allInstances}\rightarrow\forall (cs:\text{CallSite}|cs.\text{lastDefs.}\text{interactingElement}
    \rightarrow\text{select}(ie:\text{InteractingElement}|ie.\text{oclType}\rightarrow\text{FormalParam})\rightarrow\text{Empty})\]

- Instances of ReturnSite cannot have definitions or uses of actual parameters:
  \[\text{ReturnSite.allInstances}\rightarrow\forall (rs:\text{ReturnSite}|rs.\text{firstUses.}\text{interactingElement}
    \rightarrow\text{select}(ie:\text{InteractingElement}|ie.\text{oclType}\rightarrow\text{ActualParam})\rightarrow\text{Empty})\]
  \[\text{ReturnSite.allInstances}\rightarrow\forall (rs:\text{ReturnSite}|rs.\text{lastDefs.}\text{interactingElement}
    \rightarrow\text{select}(ie:\text{InteractingElement}|ie.\text{oclType}\rightarrow\text{ActualParam})\rightarrow\text{Empty})\]
Appendix D provides examples of object diagrams that are instantiations of this metamodel for different coupling in our running example (Figure 1), and the algorithms that derive coupling paths from instances of this metamodel, according to the different criteria we defined in Section 4.3 can be found in Appendix E.

### 6 CASE STUDIES

We report here on the use of our approach on two case studies implemented in Java: (1) a class scheduler and course planner for students (Jadvisor), (2) an implementation of a linked list. The case studies, and the class interactions we were interested in are detailed in Section 6.1. We did not exercise all the client-server pairs of classes in each case study but rather selected the most complex ones for investigation in this report. Section 6.2 reports on the application of our strategy to derive sequences to be tested. The mutation operators, defined specifically to focus on the interactions between modules, that we used in the case studies are then described in Section 6.3. Section 6.4 reports on the results of our approach and compares its cost (in terms of test cases) and effectiveness (in terms of killed mutants) with a black-box technique, i.e., Category-
Partition [23]: Recall that a subset of client method sequences defined during the class testing of the client class are reused for integration testing. We compare our approach (removing redundant sequences and using coupling path information to set input parameter values) with the use of Category-Partition to select input parameter values. In other words, we assess the benefit of using white-box information (i.e., coupling paths) to reduce the number of test cases and select appropriate input parameter values over simply considering the specification of the client methods. Note that no other integration testing methodology really compares to what we propose here, and we therefore cannot really compare our approach with any alternative that has been designed for the same purpose. Section 6.5 summarizes the conclusions that can be drawn from these case studies.

6.1 Case studies

6.1.1 Jadvisor

Jadvisor is a class scheduler and course planner for students, written in Java (http://jadvisor.sourceforge.net). Figure 5 shows the four subsystems of Jadvisor, as well as the class diagram in subsystem planner. The client and server classes we selected for the case study, are StudentPlan and StudentSemesterPlan respectively. The corresponding Java source code can be found in Appendix F. The aggregation between StudentPlan and StudentSemesterPlan is implemented with a two-dimensional array, called _plan, with four lines and three columns, corresponding to years (students graduate in 4 years) and semesters (3 semesters per year) respectively.
6.1.2 Linked List

The second case study is an implementation of a bi-directional linked list, involving client class List and server class Node (Figure 6): each Node instance has references to the previous and next Node instances in the list. An excerpt of the corresponding Java source code can be found in Appendix G. In this implementation, Node instances hold a data (an int value) and are uniquely identified by a key (int value). Note that a List instance is not responsible for ordering the Node instances it contains (e.g., in increasing order of the keys). Rather, this is the responsibility of clients of List instances, as methods insertAfter and insertBefore have a parameter of type Node that determines where that new Node instance (whose data value is also provided as a parameter) is to be inserted (e.g., after the node passed as a parameter in the case of insertAfter). The key attribute is uniquely set in the constructor of Node using static attribute public_key.
6.2 Testing Sequences

Bashir & Goel’s class unit testing strategy [4] has been used to derive client method sequences for the two case studies. The techniques first identifies slices where a slice is defined as a quantum of a class including only a single attribute and the set of methods that manipulate this attribute. Methods in slices are classified as reporters (return the value of an attribute), transformers (modify one or more attributes) and others (do not fall into the two previous categories). Then the rationale is to devise sequences for each slice: (1) for each other method, one sequence containing only the method is produced; (2) all possible permutations of the transformers in the slice are considered, such that each transformer appears only once in the sequence.

6.2.1 Jadvisor

Following Bashir & Goel’s strategy [4], we derive the following simple sequences: add.remove, remove.add, contains, and satisfiesPrerequisites. Note that we have omitted sequences derived for the test of the constructor and “get” methods (also called reporters in [4]) of class StudentPlan, as they do not interact with server class StudentSemesterPlan. ICFGs for methods add and remove both contain two paths that exercise the interaction with server class StudentSemesterPlan: server method sequences triggered are contains and contains.add, and contains and contains.remove for client methods add and remove, respectively. ICFGs for methods contains and satisfiesPrerequisites have more than one path as those
methods have nested loops\textsuperscript{8}, and exercise the interaction in the same way: call(s) to method \texttt{contains} of class \texttt{StudentSemesterPlan} (Appendix F).

The actual server instances on which these calls are performed depend on the number of times each loop is executed: for instance, if the first loop in \texttt{contains} is executed once, and the second loop is executed twice, two instances of the server class are used, i.e., the server instance in the first line (index 0) and first column (index 0), and the server instance in the first line (index 0) and second column (index 1) of array \texttt{_plan}. We thus decided, in order to ease the presentation of the case study, to associate symbolic names of the form \texttt{_plan(i,j)} where \(i\) and \(j\) are the line and column of the server instance in array \texttt{_plan} of the objects involved in client method sequences.

Table 5 uses this convention for representing symbolic names, and shows the tuples generated according to our approach from the four client sequences above. Note that it has been decided that client methods \texttt{add} and \texttt{remove} will be executed with the same parameters: server instance \texttt{_plan(1,2)}. Similarly, one of the symbolic names used in \texttt{satisfiesPrerequisites} is directly driven by its parameters, and has been chosen to correspond to server instance \texttt{_plan(1,0)}. We also decided that each loop be taken once and twice (we selected these numbers so as to limit the number of tuples). Last, when a given tuple is exercised by different paths in an ICFG, only one is shown in Table 5, so as to simplify the table. Table 5 shows that 14 tuples are produced.

\textsuperscript{8} For instance, verifying that a \texttt{StudentPlan} contains a particular course (method \texttt{contains}) amounts to looking at all the cells of array \texttt{_plan}: four lines and three columns.
Table 5 Tuples for Jadvisor

Table 6 shows the result of applying the three redundancy criteria we defined previously on tuples in Table 5. Server sequences for tuples 5 to 10 are included in tuples 11 to 14 with the same symbolic names, and are thus removed according to criterion 3, except tuple 5 that has been selected to ensure there is at least one execution of client method `contains`. In addition to these removals, server sequences in tuples 11 to 14 are included in tuples 2 and 4 with symbolic names of the same kind, and are thus removed according to criterion 2, except tuple 11 that has been selected so has to ensure there is at least one execution of client method `satisfiesPrerequisites`. Criterion 1 provides the same results as criterion 2 since we do not have different kinds of symbolic names.
We then identify the coupling paths for every client-method/server-method pair and used the All-Coupling-Path criterion to cover method interactions.

### 6.2.2 Linked List

Following Bashir&Goel’s class unit testing strategy [4], we derive the seven client method sequences in Table 7. As for the previous case study, we have omitted sequences derived for the test of the constructor and “get” methods of class `List`, as they do not interact with server class `Node`. CFGs for methods `insertAfter`, `insertBefore` and `deleteNode` are simple. The complexity of the corresponding ICFGs is due to the fact that these methods call `contains`, which mainly consists of a loop (`contains` traverses the list to find the `Node` instance which is passed as a parameter). We decided that the loop in each ICFG be bypassed (if possible), taken once and twice (we selected these numbers so as to limit the number of tuples). As a result methods `insertAfter`, `insertBefore`, `deleteNode` and `contains` produce 9, 10, 11 and 8 tuples respectively (see Table 13 in Appendix H), and these tuples correspond to different settings: e.g., a tuples can be exercised only when the list contains at least two elements and the second element is the parameter passed to the method. Every possible combination of those tuples has to be considered when determining tuples for client method sequences. In order to reduce the number of possible combinations, we decided that methods in a client sequence be executed with the same `Node` parameter. The total number of tuples generated from client method sequence is then 89 (the second column in Table 7 indicates the number of possible
combinations for each client method sequence). Those tuples for client method sequences can be found in Appendix H (Table 14).

<table>
<thead>
<tr>
<th>Client method sequences</th>
<th>Number of tuples produced</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 insertAfter.insertBefore.deleteNode</td>
<td>12</td>
</tr>
<tr>
<td>2 insertAfter.deleteNode.insertBefore</td>
<td>13</td>
</tr>
<tr>
<td>3 insertBefore.insertAfter.deleteNode</td>
<td>14</td>
</tr>
<tr>
<td>4 insertBefore.deleteNode.insertAfter</td>
<td>15</td>
</tr>
<tr>
<td>5 deleteNode.insertAfter.insertBefore</td>
<td>16</td>
</tr>
<tr>
<td>6 deleteNode.insertBefore.insertAfter</td>
<td>16</td>
</tr>
<tr>
<td>7 contains</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 7 Client method sequences for Linked List

When applying the three different redundancy criteria, we go from 89 tuples to 21, 21 and 17 for criteria 3, 2 and 1 respectively (see Table 15 in Appendix H for details). We then identify the coupling paths for every client-method/server-method pair and used the All-Coupling-Path criterion to cover method interactions.

6.3 Mutants

Our intent was to select mutation operators that focus on interface faults between modules so as to emulate, as realistically as possible, the situation where class testing has been performed and most remaining faults are interface faults to be found during integration. In [10] the authors propose a set of such operators, to be applied to both the caller and callee modules, that are specifically defined for the C language. However, the same concepts can be applied to other similar languages (e.g., C++ and Java), e.g., a mutation operator that modifies a global variable in a C program can be used in a Java program to modify a shared attribute. As usual, for a given system under study, all applicable mutation operators must be covered, and the distribution of mutants across operators is driven by the program characteristics (e.g., use of parameters in calls).
6.3.1 Jadvisor

Seven mutation operators were used. The first two are specific to the caller method: *FunCalDel* deletes the call to the callee method in the callsite\(^9\); *ArgRepReq* replaces each actual parameter by a compatible constants. The following five operators are specific to the callee method: *DirVarRepReq* replaces each use or definition of an interface variable (i.e., formal parameter or the shared attribute accessed by the callee method) by another variable or constant; *IndVarRepReq* is similar to *DirVarRepReq*, but applies to the local variable and constant used in the callee method; *IndVarIncDec* inserts a pre-decrement operator (--) or a pre-increment operator (++) at each reference to a local variable or constant used in the callee method; *IndVarAriNeg* inserts an arithmetic negation operator (-) before local variables and constants used in the callee method; *RetStaRep* replaces the expression used in a return statement by one of the expressions used in other return statements in the callee method.

We generated 29 mutants according to these mutation operators (see the distribution of mutants across operators in Figure 7). These mutants were seeded in both client class *StudentPlan* (mutants numbered 11 to 29) and server class *StudentSemesterPlan* (mutants numbered 1 to 10). More precisely, methods remove, add, contains, and satisfiesPrerequisites in client class *StudentPlan*, and in methods add, remove and contains in server class *StudentSemesterPlan* (see the distribution of mutants across methods in Figure 7). These numbers, and the distribution of mutants among classes/methods were deemed sufficient considering the complexity of the different methods (see the source code in Appendix F). For example, class *StudentSemesterPlan* has an attribute of type *java.util.List* for storing *Course* instances, and methods add and remove on *StudentSemesterPlan* instances simply call methods add and remove on this attribute. There is then a small number of opportunities for the seeding of faults. Note that one of these mutants was found to be equivalent (mutant number 21 seeded in method remove in class *StudentPlan*), thus resulting in 28 non-equivalent mutants.

\(^9\) If the callee method returns *void*, then nothing is required to implement the operator; otherwise, a constant is to be set and used to replace the returned value.
6.3.2 Linked List

Seven mutation operators were used for the Linked list. Three of those operators were also used for Jadvisor: FunCallDel, ArgRepNeg, DirVarRepReq. Four new operators were used (the difference between the two case studies is due to differences in the source code). The first one is specific to the caller method: ArgAriNeg inserts an arithmetic negation for an actual parameter. The following three operators are specific to the callee method: DirVarReqGlo replaces an interface variable (i.e., formal parameter or the shared attribute accessed by the callee method) with a global variable accessed by the callee; DirVarIncDec inserts/removes an increment/decrement operation for a use of an interface variable; DirVarAriNeg inserts an arithmetic negation for a use of an interface variable.

We generated 63 mutants according to these mutation operators (see the distribution of mutants across operators in Figure 8). These mutants were seeded in both client class List (38 mutants) and server class Node (25 mutants). More precisely, methods insertBefore, insertAfter, deleteNode, and contains in client class List, and in methods addNodeAfter, removeNodeAfter, getKey, getNext, getPrevious and setPrevious in server class Node (see the distribution of mutants across methods in Figure 8). These numbers, and the distribution of mutants among classes/methods were deemed sufficient considering the complexity of the different methods (see the source code in Appendix G).
6.4 Results

6.4.1 Jadvisor

As stated above, we generated functional test cases following the Category-Partition approach [23], and produced 38 test cases. Following our strategy, that is, combining tuples and coupling paths (test cases satisfy the all-coupling-paths criterion), we produced 18, 12 and 28 test cases for the redundancy criteria 3, 1 and 2, and when using no redundancy criterion (i.e., keeping all sequences), respectively (see Table 8). The number of test cases is twice the number of tuples because methods `contains` and `satisfyPrerequisites` return a Boolean value and both `true` and `false` values are tested when considering coupling paths.

The five test sets, i.e., Category-Partition, redundancy criterion 3, redundancy criteria 1 and 2, and no criterion, killed 23, 28, 25, and 28 non-equivalent mutants (out of 28) respectively, thus resulting in 82%, 100%, 89% and 100% mutation scores (percentage of non-equivalent mutants killed), respectively. Table 8 thus shows that the redundancy criterion 3 has a better cost-effectiveness than criteria 1, 2, no criterion and Category-Partition (half the number of test cases for a slightly higher effectiveness).

Figure 8 Distribution of mutants across operators and methods (Linked List)
The three mutants that are not killed by criteria 1 and 2 were seeded in client method `satisfiesPrerequisites`. After investigation, we found that two different paths in `satisfiesPrerequisites`'s ICFG correspond to tuple `(satisfiesPrerequisites, contains.contains, _plan(0,0))`, i.e., the tuple selected by criteria 1 and 2. In other words, executing server sequence `contains.contains` on symbolic name `_plan(0,0)` from client method `satisfiesPrerequisites` can be achieved in two different paths of `satisfiesPrerequisites`'s ICFG. Only one of these two paths was used when producing test cases from that tuple, thus resulting in not exercising the faulty statements in the other path.

Mutants remain alive when using Category-Partition because of the constraints that we defined among the choices in order to limit the number of resulting test cases [23]. For instance, as a result of our constraints among the choices, we verify prerequisites between courses in the same year, but do not verify them between courses in different years, thus resulting in not killing mutant 16 (seeded in method `satisfiesPrerequisites`). However, though removing those constraints would allow us to kill more mutants (and thus reaching a similar effectiveness as redundancy criterion 3), the cost difference (e.g., measured in terms of number of test cases) between the criterion and category-partition would be much larger (i.e., we go from 38 to 138 test cases).

### 6.4.2 Linked List

As stated above, we generated functional test cases following the Category-Partition approach [23], and produced 38 test cases. Following our strategy, that is, combining tuples and coupling...
paths (test cases satisfy the all-coupling-paths criterion), we produced 13, 13, 12 and 41 test cases for the redundancy criteria 3 and 2, 1 and when using no redundancy criterion (i.e., keeping all sequences), respectively (see Table 9). As opposed to the previous case study, the number of test cases is less than the number of tuples. This is due to the fact that several tuples and coupling paths are executed in the same control flow paths, and thus are exercised by the same test cases.

The five test sets, i.e., Category-Partition, redundancy criteria 3, 2 and 1 and no criterion, killed 63, 57 and 63 non-equivalent mutants (out of 63) respectively, thus resulting in 100%, 90% and 100% mutation scores (percentage of non-equivalent mutants killed), respectively (Table 9).

<table>
<thead>
<tr>
<th>Testing technique (and criterion)</th>
<th>No Criterion</th>
<th>Criterion 1</th>
<th>Criteria 2 &amp; 3</th>
<th>Category Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuples</td>
<td>89</td>
<td>17</td>
<td>21</td>
<td>N/A</td>
</tr>
<tr>
<td>Test Cases</td>
<td>41</td>
<td>12</td>
<td>13</td>
<td>38</td>
</tr>
<tr>
<td>Non-equivalent mutants killed</td>
<td>63</td>
<td>57</td>
<td>57</td>
<td>63</td>
</tr>
<tr>
<td>Mutation Score</td>
<td>100%</td>
<td>90%</td>
<td>90%</td>
<td>100%</td>
</tr>
<tr>
<td>Number of Live Mutants</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Live Mutants</td>
<td>11,12,13,22,36</td>
<td>11,12,13,22,36</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9 Results for Linked List

As in the previous case study, the three redundancy criteria are less costly than Category-Partition (one third of the number of test cases). Note that, as opposed to Jadvisor, we did not use constraints between choices to reduce the number of test cases. However, the three redundancy criteria miss the same 5 mutants, resulting in a smaller effectiveness. These mutants were all seeded in one statement, i.e., a call to the constructor of class `Node` in method `insertBefore` of class `List`: The first actual parameter, which represents the data stored in the node, does not have the correct value (line 18: Table 11 in Appendix G). The corresponding tuple, `(insertBefore, Node, Node)`, was considered redundant with other similar tuples by the three redundancy criteria and thus removed: call to `Node`’s constructor is tested in another method of class `List` (i.e., method `insertAfter` as shown in Appendix H by Table 13 and Table 14). Another selection of redundant tuples would have resulted in test sets that achieve 100% mutation score. Such a situation was expected since, as mentioned in Section 3.4, reducing
redundancy between tuples can result in not exercising control flow paths, and thus not detecting faults along those paths.

After investigation, it appears that these 5 live mutants would have been killed by any class testing technique applied to client class \texttt{List}, as long as the stub used to simulate the behaviour of class \texttt{Node} reports on the value of the parameters that are passed to calls. The question is then whether these 5 mutants should be counted as our strategy assumes that interacting classes have passed their local tests.

### 6.5 Results from the case studies

The two case studies suggest that the three redundancy criteria along with the most expensive coupling based criterion (i.e., the all-coupling-path criterion) produce less expensive test sets than Category-Partition: from half to one third of the number of test cases. Recall that we used Category-Partition as no other integration testing methodology really compares to what we propose here, and we therefore cannot really compare our approach with any alternative that has been designed for the same purpose.

The results of the three redundancy criteria in terms of effectiveness are mixed and vary across case studies: mutation scores in the first and second case studies are 100\% and 90\% (or 100\% if we do not count the mutants that would have been killed by class testing techniques), respectively. However, the mutation score is high in both cases, especially since the five live mutants in the LinkedList case study were seeded in the same statement. The main reason for such a difference in mutation score is that when applying our strategy, some random choices have to be made during the selection of redundant tuples. Other case studies will be performed to better generalize these results.

### 7 CONCLUSION

We have defined a precise and automatable procedure to test class interactions in the context of class integration testing. This is complementary to our previous work on integration test orders [7, 8] and is based on existing works regarding class and integration testing [4, 13]. Assuming a given class integration order, we test each pair of interacting classes by analyzing the patterns of
method executions from clients to servers and their inter-method control and data flows (referred to as coupling paths) when a method call takes place from the client to the server. Our rationale is to define coverage criteria that ensure that interactions between classes are exercised (i.e., all possible server method sequences are executed, coupling paths are covered) while minimizing the number of test cases to be re-executed on the client class during integration. We thus define three criteria and compare them to using a black-box testing technique (category-partition) to set the method parameter values, so as to have a baseline of comparison and assess the benefit of using control and data flow information to optimize integration testing. The case studies we present shows that our strategy and criteria yield smaller test suites and higher fault detection rates. This is particularly true for one case study and one of the criteria we define that produces half the test cases of category-partition while detecting all seeded class interface faults. Category-partition is a general-purpose functional testing technique and it should be expected that it does not perform as well as a technique that is specifically aimed at class integration testing. Our results are consistent with this expectation and thus show that the class integration test technique we propose is particularly efficient and far better than performing standard black-box testing in this context. Future work includes running additional case studies and the refinement of our integration test support environment.
REFERENCES


Appendix A   More details on the running example

The figure below (Figure 9) provides the ICFGs for methods mA1(), mA2(), and mA5() in our running example (Figure 1).

![Diagram of ICFGs for mA1(), mA2(), and mA5() in Figure 1]

Table 10 shows the sequences of server methods triggered by methods mA1() and mA5() (see Figure 1). As we can see, though mA5() calls mA1() in a similar way as mA2() (the ICFGs for mA2() and mA5() are similar), the sequences triggered are different because mA5() calls mA1() using attribute b2 (instead of parameter bb in the case of mA2()).
<table>
<thead>
<tr>
<th>Server methods triggered by the two paths in mA5()’s ICFG</th>
<th>Corresponding server method sequences for each symbolic name (server instance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(aB, mB3()), (b1, mB1()), (b1, mB2()), (b1, mB3()), (b2, mB3())</td>
<td>aB: mB3()</td>
</tr>
<tr>
<td></td>
<td>b1: mB1().mB2().mB3()</td>
</tr>
<tr>
<td></td>
<td>b2: mB3()</td>
</tr>
<tr>
<td>(aB, mB3()), (b2, mB2()), (b2, mB1()), (b1, mB3()), (b2, mB3())</td>
<td>aB: mB2()</td>
</tr>
<tr>
<td></td>
<td>b1: mB3()</td>
</tr>
<tr>
<td></td>
<td>b2: mB2().mB1().mB3()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Server methods triggered by the two paths in mA1()’s ICFG</th>
<th>Corresponding server method sequences for each symbolic name (server instance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b1, mB1()), (b1, mB2()), (b1, mB3()), (b, mB3())</td>
<td>b1: mB1().mB2().mB3()</td>
</tr>
<tr>
<td></td>
<td>b: mB3()</td>
</tr>
<tr>
<td>(b2, mB2()), (b2, mB1()), (b1, mB3()), (b, mB3())</td>
<td>b2: mB2().mB1().mB3()</td>
</tr>
<tr>
<td></td>
<td>b: mB3()</td>
</tr>
</tbody>
</table>

Table 10 Server method sequences triggered by method mA5() in Figure 1
Appendix B  Instantiation of the tuple metamodel

In this appendix, we use our running example (Figure 1) and client and server sequences in Table 3 and Table 4 to illustrate how instances of the tuple metamodel are created. Given the four client method sequences in Table 3, Figure 10 shows an excerpt of the corresponding tuple metamodel instance. Two Method instances appear (names cM1 and cM2). They are both associated with three tuples (called t1, t2, t3 and t4, t5, t6, respectively). Those tuples share the same server methods (called sM1, sM2 and sM3) and symbolic names.

Figure 10 Instance of the tuple metamodel for client methods mA1 and mA2 (Figure 1)

Figure 11 then shows the result of the first algorithm described in Appendix C, i.e., the algorithm that derives server method sequences triggered by client method sequences, given server method sequence triggered by client methods. In this particular case, the server method sequence triggered by methods mA1 and mA2 from Figure 10 are used. Three tuples are created (called tuple1, tuple2 and tuple3). The corresponding client and server method sequences contain methods mA1 and mA2, and mB1, mB2 and mB3 respectively (as described in Table 3). Note that the order is not shown in the figure.
Figure 11 Instance of the tuple metamodel showing tuples for client sequence mA1 . mA2
Appendix C  Algorithms for the generation of tuples

Seven algorithms are provided in this appendix. The first two are used to derive tuples for complete client method sequence from tuples associated to individual client methods. Then three algorithms provide implementations to the three redundancy criteria. The last two algorithms are used by the redundancy criteria algorithms to check whether there is an inclusion relationship between two server method sequences, and whether additional tuples are needed to ensure that every client method appears in at least one tuple when applying the redundancy criteria.

C.1 Procedure combineMethodTuples

This procedure uses tuples for individual client methods to produce tuples for complete client method sequences. The procedure calls addTuples, that combines tuples from individual methods for a given symbolic name, for every symbolic names used in tuples triggered by client methods.

**Input/Output:** origTestReq: TestRequirement

**Algorithm**

```plaintext
procedure combineMethodTuples(origTestReq: TestRequirement)
var: symbNameUsed: Sequence(SymbolicName)
    clientSeq: Sequence(Method)
    serverSeq: Sequence(Method)
    i, j: int
begin
    serverSeq := new Sequence;
    for i:=1 to origTestReq.clientSeqNumber do begin
        clientSeq := origTestReq.method[i]; // one sequence of client methods
        symbNameUsed := clientSeq.originalTuples.symbolicName->asSet->asSequence;
            // all the symbolic names used in client methods
        for j:=1 to symbNameUsed->size do begin
            empty serverSeq;
            addTuples(clientSeq,1,serverSeq,null,symbNameUsed->at(j),origTestReq)
        end
    end
end
```
C.2 Procedure addTuples

This procedure produces tuples for a given client method sequence (first parameter, i.e., \texttt{classTestSeq}) and for a given symbolic name (fifth parameter, i.e., \texttt{sN}) in a recursive manner. Each time a new method in the sequence must be considered, its position in the sequence is provided (second parameter, i.e., \texttt{methodNum}). The third and fourth parameters (i.e., \texttt{serverSeq1} and \texttt{serverSeq2}) are the server sequence triggered by the previous client methods in the sequence (except the last one) and the server sequence triggered by the last previous client method in the sequence. The last parameter (i.e., \texttt{testReq}) is the \texttt{TestRequirement} instance that holds the client method sequences.

\textbf{Input:} \texttt{classTestSeq: Sequence(Method)} // the client method sequence to be used
\texttt{serverSeq1: Sequence (Method)} // the server method sequence triggered by client methods number 1 to methodNum-2 in \texttt{classTestSeq}.
\texttt{serverSeq2: Sequence(Method)} // the server method sequence triggered by client method methodNum-1 in \texttt{classTestReq}
\texttt{sN: SymbolicName} // the symbolic name to be used
\texttt{methodNum: int} // the index in \texttt{classTestSeq} of the method to be considered in the current execution of the procedure
\texttt{testReq: TestRequirement} // the \texttt{TestRequirement} instance to be used

\textbf{Input/Output:} \texttt{testReq: TestRequirement}

\textbf{Algorithm}

\texttt{procedure addTuples( classTestSeq: Sequence (Method), methodNum:int, serverSeq1: Sequence (Method), serverSeq2: Sequence (Method), sN: SymbolicName, testReq: TestRequirement)}
\texttt{var: clientMethod: Method}
\texttt{serverSeq: Sequence (Method)}
\texttt{tuples: Sequence(Tuple)}
\texttt{t: Tuple}
\texttt{i: int}
\texttt{begin}
\texttt{if (serverSeq2 != null) then}
\texttt{serverSeq := serverSeq1->union(serverSeq2);}
\texttt{else}
\texttt{serverSeq := serverSeq1;}
\texttt{end}
\texttt{if methodNum > classTestSeq->size then}
\texttt{t := new Tuple;}
\texttt{t.symbolicName := sN;}
\texttt{t.serverSeq := serverSeq;}
\texttt{t.clientSeq := classTestSeq;}
\texttt{testReq.tuple->including(t);}
\texttt{else}
clientMethod := classTestSeq->at(methodNum);
tuples := clientMethod.originTuples->select(symbolicName = sN)->asSequence;
if tuples->isEmpty then
    addTuples(classTestSeq, methodNum+1, serverSeq, null, sN, testReq);
else
    for i:=1 to tuples->size do
        addTuples(classTestSeq, methodNum+1, serverSeq, tuples->at(i).serverSeq, sN, testReq);
end
end

c3 Procedure removeRedundancyOne

This function implements redundancy criterion 1. It consists in selecting all the tuples that are not contained in other tuples (the server method sequence in not included in a server method sequence from another tuple). Then additional tuples may be added by the call to checkPendingTuples so as to make sure that every client method appears in at least one tuple.

Input: testReq: TestRequirement

Output: Set(Tuple)

Algorithm

function removeRedundancyOne(testReq: TestRequirement): Set(Tuple)
var: tuple1, tuple2: Tuple
    pendingTuples: Bag(Tuple)
    resultTuples: Bag(Tuple)
    i: int
function: checkIncludedTuple
procedure: checkPendingTuples
begin
    resultTuples := testReq.tuple;
    resultTuples->asSequence;
    for i:=1 to testReq.tuple->size do
        begin
            tuple1 := resultTuples->at(i);
            if (resultTuples->exists(tuple2:Tuple| checkIncludedTuple(tuple2,tuple1)) then
                pendingTuples->including(tuple1); //tuple1 is included in another tuple
                resultTuples->excluding(tuple1);
            endif
        end
    // in the end, resultTuples contains the tuples that are not included in other tuples
    checkPendingTuples(pendingTuples, resultTuples);
    removeRedundancyOne = resultTuples->asSet;
end
C.4 Procedure removeRedundancyTwo

This function implements redundancy criterion 2. It consists in selecting all the tuples that are not contained in other tuples, while accounting for the kind of the symbolic name (the server method sequence in not included in a server method sequence from another tuple). Then additional tuples may be added by the call to checkPendingTuples so as to make sure that every client method appears in at least one tuple.

Input: testReq: TestRequirement

Output: Set(Tuple)

Algorithm

function removeRedundancyTwo(testReq: TestRequirement): Set(Tuple)

var: tuple1, tuple2: Tuple
pendingTuples: Bag (Tuple)
resultTuples: Bag(Tuple)
i: int

function: checkIncludedTuple
procedure: checkPendingTuples

begin

resultTuples := testReq.tuple;
resultTuples->asSequence;
for i:= 1 to testReq.tuple->size do
begin

tuple1 := resultTuples->at(i);
if (resultTuples->exists(tuple2:Tuple|
tuple2.symbolicName.kind = tuple1.symbolicName.kind
and checkIncludedTuple(tuple2,tuple1)) then
result->including(tuple1); // tuple1 is included in another tuple
resultTuples->excluding(tuple1);
endif
end

// in the end, resultTuples contains the tuples that are not included in other tuples, accounting for the kind of the associated symbolic names
checkPendingTuples(pendingTuples, resultTuples);
removeRedundancyTwo = resultTuples->asSet;
end

C.5 Procedure removeRedundancyThree

This function implements redundancy criterion 3. It consists in selecting all the tuples that are not contained in other tuples, while accounting for the name of the symbolic name (the server method sequence in not included in a server method sequence from another tuple). Then additional tuples may be added by the call to checkPendingTuples so as to make sure that every client method appears in at least one tuple.
Input: testReq: TestRequirement
Output: Set(Tuple)

Algorithm

function removeRedundancyThree(testReq: TestRequirement): Set(Tuple)
var:
    tuple1, tuple2: Tuple
    pendingTuples: Bag(Tuple)
    resultTuples: Bag(Tuple)
    i: int
begin
    resultTuples := testReq.tuple
    resultTuples->asSequence;
    for i:= 1 to testReq.tuple->size do
        begin
            tuple1 := resultTuples->at(i);
            if (resultTuples->exists(tuple2:Tuple|
                tuple2.symbolicName.name = tuple1.symbolicName.name
                and checkIncludedTuple(tuple2,tuple1))
                then
                    result->including(tuple1); //tuple1 is included in another tuple
                    resultTuples->excluding(tuple1);
        endif
    end
    // in the end, resultTuples contains the tuples that are not included in other
    // tuples, accounting for the name of the associated symbolic names
    checkPendingTuples(pendingTuples, resultTuples);
    removeRedundancyThree = resultTuples->asset;
end

C.6 Procedure checkPendingTuples

This procedure makes sure that every client method appears in at least one of the tuples selected when applying a particular redundancy criterion. Parameters pendingTuples and resultTuples are the tuples that were not selected by the redundancy criterion and the tuples that were selected. If a client method appears in a tuple from pendingTuples does not appear in any tuple in resultTuples, then a tuple from pendingTuples must be added to resultTuples. The procedure identifies the tuple in pendingTuples with the maximum number of client methods that do not appear in the resultTuples, and adds it to that resultTuples. In other words, we make sure that every client method appears in tuples in resultTuples with a minimum number of tuples. This is repeated until every client method appears in at least one tuple from resultTuples.
Input: pendingTuples: Bag (Tuple)

Input/Output: resultTuples: Set(Tuple)

Algorithm

procedure checkPendingTuples(pendingTuples: Bag(Tuple), resultTuples: Set(Tuple))
var: maxDiffTuple, tmpTuple: Tuple
tmpPendingTuples: Bag (Tuple)
maxDiffNumber, diffNumber: int
begin
Tuple maxDiffTuple := new Tuple;
Tuple tmpTuple := new Tuple;
while pendingTuples->size > 0 do
begin
maxDiffNumber := 0;
maxDiffTuple := null;
tmpPendingTuples := pendingTuples;
for i:= tmpPendingTuples->size downto 1 do
begin
//looking for the tuple with the maximum number of different client methods
tmpTuple := tmpPendingTuples->at(i);
if resultTuples.clientSeq->includes(tmpTuple.clientSeq) then
pendingTuple->excluding(tmpTuple);
else
diffNumber := resultTuples.clientSeq->asset
->reject(tmpTuple.clientSeq->asSet)->size;
if diffNumber > maxDiffNumber then
maxDiffNumber := diffNumber;
maxDiffTuple := tmpTuple;
endif
endif
resultTuples->including(maxDiffTuple);
pendingTuples->excluding(maxDiffTuple);
end //end while loop
end

C.7 Function checkIncludedTuple

This function is used to check whether a tuple’s server method sequence is included in another tuple’s server method sequence. If the second parameter (tuple2) is included in the first parameter (tuple1) then the function return true. Server method sequence inclusion means that the same methods appear in the two sequences in the same order.

Input: tuple1, tuple2: Tuple

Output: boolean

Algorithm

function checkIncludedTuple(tuple1, tuple2: Tuple): boolean
var: subSeq: Sequence
i: int
begin
if tuple1.server->size >= tuple2.server->size then
for i:=1 to tuple1.server->size do
begin
if tuple1.server->size > (i + tuple2.server->size) then
    return false;
endif
if tuple1.server->at(i) = tuple.server->first then
    subSeq := tuple1.server->subsequence(i, i+tuple2.server->size-1);
    if subSeq = tuple2.server then
        return true;
    endif
endif
end
return false;
else
    return false;
endif
end
Appendix D  Instantiation of the coupling information metamodel

In this appendix we illustrate the instantiation of the coupling information metamodel in Figure 4 (Section 5.3) using our running example in Figure 1. Note that we split the object diagram, instance of the metamodel in several parts so as to illustrate with simple object diagrams the different coupling between client and server classes (the object diagrams below are not meant to be complete).

First, Figure 12 illustrates the call coupling that exists between client class A (i.e., client method mA1) and server class B (i.e., server method mB1). The call site at line 11 in class A is in client method mA1 (the caller method) and is a call to method mB1 in class B. mB1 has one return site at line 7 in class B.

![Diagram of call coupling](image)

Figure 12 Instance of the metamodel (excerpt) for a call coupling

Figure 13 illustrates parameter coupling due to call to method mB1 (class B) at line 11 in method mA1 (class A): parameter coupling path involving in parameter i (Figure 13.a), and parameter coupling path involving the return value (Figure 13.b). Line 9 in class A is the last definition of actual parameter i (type int) before the call to method mB1 in line 11, thus objects named lastDefBefCall (class UseDef), actParam (class ActualParam), call (class CallSite) and callee (class Method). The first use in callee (i.e., method mB1 in class B) is a use of formal parameter j (type int) at line 5, thus objects named callee (class Method), firstUseInCallee (class UseDef) and formParam (class FormalParam). Similarly (Figure 13.b), call to mB1 returns a value: last definition before return at line 6 in class B (object named lastDefBefRet of class UseDef) and first use after call at line 17 in class A (object named firstUseAftCall of class UseDef).
(a) in parameter  
(b) out parameter

Figure 13 Instance of the metamodel (excerpts) for parameter coupling

Figure 14 illustrates shared attribute coupling (attribute VAR in class C shared by methods mA4 and mB1 in Figure 14.a) and external device coupling (file named File.txt shared by methods mA6 and mA4 in Figure 14.b). Shared attribute VAR (object named sharedAtt of class SharedAttribute in Figure 14.a) is defined (last definition) at line 38 in class A (object named sharedDef of class UseDef) before call to mB1 at line 39 in class A, and used (first use) at line 6 in class B (object named sharedUse of class UseDef). Similarly, external device File.txt (object named extDev of class ExternalDevice in Figure 14.b) is defined (last definition) at line 55 in class A (object named externalDef of class UseDef) before call to mB5 at line 39 in class A, and used (first use) at line 6 in class B (object named externalUse of class UseDef).
Figure 14 Instance of the metamodel (excerpts) for shared attribute and external device coupling
Appendix E  Algorithms for coupling based integration test requirements

Four functions are defined in this appendix for the derivation of call, parameter, shared attribute and external device coupling paths respectively.

E.1 Function computeCallCouplingPath

This function computes the set of call coupling paths for a client method.

**Input:** caller: Method

**Output:** callCouplingPaths: Set(Path)

**Algorithm:**

```plaintext
function computeCallCouplingPaths(caller: Method): Set(Path)

var:
    p: Path;

begin
    for i:=1 to caller.call->size do
        p := new Path;
        p->append(caller.call->at(i))->append(caller.call.callee.return->first);
        callCouplingPaths->including(p);
    end
end
```

E.2 Function computeParameterCouplingPaths

This function computes the parameter coupling paths for a client method. It first consists in deriving coupling paths for in parameters, and then for out parameters.

**Input:** caller: Method

**Output:** parameterCouplingPaths: Set(Path)

**Algorithm:**

```plaintext
function computeParameterCouplingPaths(caller: Method): Set(Path)

var:
    call: CallSite;
    returnSites: Sequence(ReturnSite);
    aReturn: ReturnSite;
    actPs: Set(ActualParam);
    ap: ActualParam;
    fp: FormalParam;
    actDefs, formUses, actUses, formDefs: Set(UseDef);
    p: Path;
    i, j, k, l: int;
```
begin
  for i := 1 to caller.call->size do // for each call in client method
    begin
      call := caller.call->at(i);
      // Devise coupling paths for in parameters
      // get the set of the actual parameters the caller defines before call
      actPs := call.lastDefs.interactingElement
               ->select(ap:InteractingElement|ap.oclType = ActualParameter)->asSequence;
      for j := 1 to actPs->size do // for each actual parameter defined before call
        begin
          ap := actPs->at(j);
          fp := ap.formalParam; // formal parameter corresponding to ap
          // get the sequence of last defs before call of ap
          actDefs := call.lastDefs->select(ld:UseDef| ld.interactingElement = ap)
                     ->asSequence;
          // get the set of first uses in callee of fp
          formUses := call.callee.firstUses->select(fu:UseDef|fu.interactingElement = fp)
                      ->asSequence;
          for k := 1 to actDefs->size do // for each last def before call of ap
            begin
              for l := 1 to formUses->size do // for each first use in callee of fp
                begin
                  p := new Path;
                  path->append(actDefs->at(k))->append(call)->append(formUses->at(l));
                  parameterCouplingPaths->including(p);
                end
            end
        end // for each actual parameter
      end // for each call in client method
    end
end

// Devise coupling paths for out parameters
// get the set of the actual parameters the caller uses after call
actPs := call.firstUses.interactingElement
          ->select(ap:InteractingElement|ap.oclType = ActualParameter)->asSequence;
// get the sequence of return sites of callee
returnSites = call.callee.return
for j := 1 to actPs->size do // for each actual parameter used after call
  begin
    ap := callerUseAtts->at(j);
    fp := ap.formalParam;
    // get the set of first uses (after call) of ap
    actUses := call.firstUses->select(fu:UseDef|fu.interactingElement = ap)
               ->asSequence;
    for k := 1 to actUses->size do // for each first use after call of ap
      begin
        for l := 1 to returnSites->size do // for each return site
          begin
            aReturn := returnSites->at(l);
            // get the set of aReturn’s last defs of fp
            formDefs := aReturn.lastDefs->select(ld:UseDef|ld.interactingElement = fp)
                        ->asSequence;
            for m := 1 to formDefs->size do // for each aReturn’s last def of fp
              begin
                for n := 1 to actUses->size do // for each first use after call of ap
                  begin
                    for p := 1 to returnSites->size do // for each return site
                      begin
                        aReturn := returnSites->at(p);
                        // get the set of aReturn’s last defs of fp
                        formDefs := aReturn.lastDefs->select(ld:UseDef|ld.interactingElement = fp)
                                    ->asSequence;
                        for m := 1 to formDefs->size do // for each aReturn’s last def of fp
                          begin
                            path->append(formDefs->at(m))->append(aReturn)->append(actUses->at(n));
                            parameterCouplingPaths->including(path);
                          end
                      end
                    end
                  end
              end
          end
      end
  end
end // for each actual parameter used after call
E.3 Function computeSharedAttributeCouplingPaths

This function computes the shared attribute coupling paths for a client method.

**Input:** caller: Method

**Output:** sharedAttributeCouplingPaths: Set(Path)

**Algorithm:**

```plaintext
function computeSharedAttributeCouplingPaths(caller: Method): Set(Path)
var:
  call: CallSite
  aReturn: ReturnSite
  returnSites: Sequence(ReturnSite)
  att: SharedAttribute
  callerDefAtts, callerUseAtts: Set(SharedAttribute)
  sharedDefs, sharedUses: Set(UseDef)
  path: Path
  i, j, k, l: int
begin
  for i:= 1 to caller.call->size do // for each call in client method
    begin
      // Compute coupling paths involved in the call
      call := caller.call->at(i);
      // get the set of the shared attribute the caller defines
      callerDefAtts := call.lastDefs.interactingElement
      ->select(sa:InteractingElement|sa.oclType = SharedAttribute)
      ->asSequence;
      for j:=1 to callerDefAtts->size do
        begin // for each shared attribute defined before the call
          att := callerDefAtts->at(j)
          // get the set of last defs of att in caller
          sharedDefs := call.lastDefs->select(ld:UseDef|ld.interactingElement = att)
          ->asSequence;
          // get the set of first uses of att in callee
          sharedUses := call.callee.firstUses->select( fu:UseDef |
            fu.interactingElement = att)->asSequence;
          for k:=1 to sharedDefs->size do // for each last def before call of att
            begin
              for l:=1 to sharedUses->size do // for each first use in callee of att
                begin
                  p := new Path;
                  path->append(sharedDefs->at(k))->append(call)->append(sharedUses->at(l));
                  sharedAttributeCouplingPaths->including(path);
                end
            end
        end
    end
  // Compute coupling paths involved in the return
  // get the set of the shared attribute the caller uses
  callerUseAtts := call.firstUses.interactingElement
  ->select(sa:InteractingElement|sa.oclType = SharedAttribute)
  ->asSequence;
  // get the sequence of return sites of callee
  returnSites = call.callee.return
end // end function
```
for j:=1 to callerUseAtts->size do 
begin // for each shared attribute (first) used after call in caller
    att := callerUseAtts->at(j);
    // get the set of first uses (after call) of att in caller
    sharedUses := call.firstUses->select(fu:UseDef|fu.interactingElement = att)->asSequence
    for k:=1 to sharedUses->size do // for each first use (after call) of att
        begin
            for l:=1 to returnSites->size do // for each return site
                begin
                    aReturn := returnSites->at(l);
                    // get the set of last def of att before aReturn
                    sharedDefs := aReturn.lastDefs->select(ld:UseDef|
                        ld.interactingElement = att)->asSequence
                    for m:=1 to sharedDefs->size do // for each last def of att
                        begin
                            p := new Path;
                            path->append(sharedDefs->at(m))->append(aReturn)->append(sharedUses->at(k));
                            sharedAttributeCouplingPaths->including(path);
                        end
                    end // for each first use (after call) of att
                end // for each return site
        end // for each call in client method
end // end function

E.4 Function computeExternalDeviceCouplingPath

This function computes the external device coupling paths for a client method.

Input: caller: Method

Output: externalDeviceCouplingPaths: Set(Path)

Algorithm:

function computeExternalDeviceCouplingPaths(caller: Method): Set(Path)
var:
    call: CallSite
    aReturn: ReturnSite
    returnSites: Sequence(ReturnSite)
    device: ExternalDevice
    callerDefDevs, callerUseDevs: Set(ExternalDevice)
    extDevDefs, extDevUses: Set(UseDef)
    path: Path
    i, j, k, l: int
begin
    for i:=1 to caller.call->size do // for each call in client method
        begin
            // Compute coupling paths involved in the call
            call := caller.call->at(i);
            // get the set of the external device the caller defines
            callerDefDevs := call.lastDefs.interactingElement
                ->select(ed:InteractingElement|ed.oclType = ExternalDevice)
                ->asSequence;
            for j:=1 to callerDefDevs->size do
                begin // for each external device defined before the call

device := callerDefDevs->at(j)
// get the set of last defs of device in caller
extDevDefs := call.lastDefs->select(ld:UseDef|ld.interactingElement = device)->asSequence;
// get the set of first uses of device in callee
extDevUses := call.callee.firstUses->select(fu:UseDef|
    fu.interactingElement = device)->asSequence;
for k:=1 to extDevDefs->size do
    begin
        // for each last def before call of device
    for l:=1 to extDevUses->size do
        begin
            // for each first use in callee of device
            p := new Path;
            path->append(extDevDefs->at(k))->append(call)->append(extDevUses->at(l));
            externalDeviceCouplingPaths->including(path);
        end
    end
end // for each external device defined before the call

// Compute coupling paths involved in the return
// get the set of the external device the caller uses
callerUseDevs := call.firstUses.interactingElement
    ->select(ed:InteractingElement|ed.oclType = ExternalDevice)
    ->asSequence;
// get the sequence of return sites of callee
returnSites = call.callee.return
for j:=1 to callerUseDevs->size do
    begin // for each external device (first)used after call in caller
        device := callerUseDevs->at(j);
        // get the set of first uses (after call) of device in caller
        extDevUses := call.firstUses->select (fu:UseDef|
            fu.interactingElement = device)->asSequence
        for k:=1 to extDevUses->size do
            begin
                // for each first use (after call) of device
                for l:=1 to returnSites->size do // for each return site
                    begin
                        aReturn := returnSites->at(l);
                        // get the set of last def of device before aReturn
                        extDevDefs := aReturn.lastDefs->select (
                            ld:UseDef|
                            ld.interactingElement = device)->asSequence
                        for m:=1 to extDevDefs->size do //for each last def of device
                            begin
                                p := new Path;
                                path->append(extDevDefs->at(m))->append(aReturn)->append(extDevUses->at(1));
                                externalDeviceCouplingPaths->including(path);
                            end
                        end
                end
            end // for each first use (after call) of device
        end // for each external device (first)used after call in caller
    end // for each call in client method
end // end function
Appendix F  Jadvisor - Details

```java
package jadvisor.planner;
import jadvisor.scheduler.Course;

public class StudentPlan implements Serializable {
  private StudentSemesterPlan[][] _plan;

  public StudentPlan (int numYears, int numSemesters) {
    ...
  }

  public void add (Course course, int year, int semester) {
    if (!_plan[year][semester].contains(course)) {
      _plan[year][semester].add(course);
      fireCourseAdded(course, year, semester); // GUI
    }
  }

  public void remove (Course course, int year, int semester) {
    if (_plan[year][semester].contains(course)) {
      _plan[year][semester].remove(course);
      fireCourseRemoved(course, year, semester); // GUI
    }
  }

  public boolean contains (Course course) {
    for (int i = 0; i < _plan.length; i++)
      for (int j = 0; j < _plan[i].length; j++)
        if (_plan[i][j].contains(course))
          return true;
    return false;
  }

  public boolean satisfiesPrerequisites (Course course, int year, int semester) {
    int counter;
    List prerequisites = course.getPrerequisites();
    for (int c = 0; c < prerequisites.size(); c++) {
      counter = 0;
      for (int i = 0; i < year; i++)
        for (int j = 0; j < semesters(i); j++)
          if (_plan[i][j].contains((Course)prerequisites.get(c)))
            counter++;
      for (int j = 0; j < semester; j++)
        if (_plan[year][j].contains((Course)prerequisites.get(c)))
          counter++;
      if (counter == 0)
        return false;
    }
    return true;
  }
```

Figure 15 Java code chunk for class StudentPlan
package jadvisor.planner;
import jadvisor.schedular.Course;

public class StudentSemesterPlan implements ListModel, Serializable {
    private final List _semesterPlan;

    public StudentSemesterPlan() {_semesterPlan = new LinkedList();}

    public void add (Course course) {
        _semesterPlan.add(course);
    }

    public void remove (Course course) {
        _semesterPlan.remove(course);
    }

    public boolean contains (Course course) {
        for (int i = 0; i < _semesterPlan.size(); i++)
            if (course.equals(_semesterPlan.get(i)))
                return true;
        return false;
    }

    ......
Appendix G  Linked List – Details

```java
public class List{
    private Node head;
    private int size = 0;

    public List() {
    }

    // Inserts a node before the Node n in this list.
    public void insertBefore(int i, Node n) {
        if (size == 0) {
            if (n != null)
                return;
            else
                head = new Node(i, null, null);
        } else {
            if ((n == null) || (!contains(n)))
                return;
            else {
                if (n.getPrevious() == null) {
                    Node newNode = new Node(i, null, n);
                    n.setPrevious(newNode);
                    head = newNode;
                } else
                    n.getPrevious().addNodeAfter(i);
            }
        }
        size++;
    }

    // Inserts a node after the Node n in this list.
    public void insertAfter(int i, Node n) {
        if (size == 0) {
            if (n != null)
                return;
            else {
                head = new Node(i, null, null);
            }
        } else {
            if ((n == null) || (!contains(n)))
                return;
            else {
                if (n.getPrevious() == null) {
                    Node newNode = new Node(i, null, n);
                    n.setPrevious(newNode);
                    head = newNode;
                } else
                    n.getPrevious().addNodeAfter(i);
            }
        }
        size++;
    }

    // Deletes Node n in this list.
    public void deleteNode(Node n) {
        if ((n == null) || (!contains(n)))
            return;
        Node e = n.getPrevious();
        if (e == null) { // n is head
            head = n.getNext();
            if (n.getNext() != null)
                n.getNext().setPrevious(head);
        } else
            e.removeNodeAfter();
        size--;
    }

    ...}
```
public boolean contains(Node n) {
    if (n == null)
        return false;
    Node e = head;
    int key = n.getKey();
    int data = n.getData();
    while (e != null) {
        if ((e.getKey() == key) && (e.getData() == data))
            return true;
        e = e.getNext();
    }
    return false;
}

Table 11 Source code for class List (excerpt)

public class Node {
    static int public_key = 0;
    private int key;
    private int idata;
    private Node previous;
    private Node next;
    public Node(int element, Node previousNode, Node nextNode) {
        key = public_key++;
        idata = element;
        next = nextNode;
        previous = previousNode;
    }
    public void addNodeAfter(int element) {
        Node n = new Node(element, this, next);
        next = n;
        if (n.next != null)
            n.next.previous = n;
    }
    public void removeNodeAfter() {
        if (next != null) {
            next = next.next;
            if (next != null)
                next.previous = this;
        }
    }
    ...
}

Table 12 Source code for class Node (excerpt)
Appendix H  Tuples for Linked List

The table below (Table 13) shows the server sequences, the corresponding symbolic names and the conditions (setting) under which these sequences are triggered (conditions are detailed after the table) for each method in client class `List`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Server sequence</th>
<th>Symbolic name</th>
<th>Setting (see below for details)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertAfter</td>
<td>Node</td>
<td>ClassNode</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td>n</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>getKey.getData.next</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData.addNodeAfter</td>
<td>n</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData.next</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData.addNodeAfter</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertBefore</td>
<td>Node</td>
<td>ClassNode</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td>n</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>getKey.getData.next</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData.</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData.setPrevious</td>
<td>setPrevious</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData.next</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deleteNode</td>
<td>getKey.getData</td>
<td>n</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td>n</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>getKey.getData.next</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData.</td>
<td>n</td>
<td>G</td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData.getPrevious</td>
<td>getPrevious</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData.getPrevious</td>
<td>getNext</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td>n</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>getKey.getData.next</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData.getPrevious</td>
<td>removeNodeAfter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData.getPrevious</td>
<td>getNext</td>
<td></td>
</tr>
<tr>
<td>contains</td>
<td>getKey.getData</td>
<td>n</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td>n</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>getKey.getData.next</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData</td>
<td>n</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td>n</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>getKey.getData.next</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey.getData.getKey.getData</td>
<td>n</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>getKey.getData</td>
<td>n</td>
<td></td>
</tr>
</tbody>
</table>

Table 13 Tuples for methods in class `List`
Conditions on the setting are the following:

A Empty list and parameter n equals n (list.size=0 and n=null);
B Empty list and parameter n doesn’t equal n (list.size=0 and n!=null);
C Non-empty list and parameter n equals n (list.size!=0 and n=null);
D List with one element different from parameter n (list.size=1 and n!=null, n=head);
E List with one element, which is parameter n (list.size=1 and n=head);
F List with two elements and parameter n is not in the list (list.size=2 and n!=null, n is not in the list). In this case, symbolic name e is the second element in the list;
G List with two elements and the first one is parameter n (list.size=2 and n=head). In this case, symbolic name is the second element in the list;
H List with at least two elements and the second one is parameter n (list.size>=2 and n is the second element);
I List with three elements and the second one is parameter n (list.size=3 and n=head);
J List with at least three elements and the second one is parameter n (list.size>=3 and n is the third element).

Given the client method sequences in Table 7 (Section 6.2.2) and the server method sequences they trigger (Table 13), Table 14 below indicates all the tuples that are produced according to our strategy (combination of tuples from each client method). Each cell in the second column shows a client method sequence (first line of the cell) along with one possible triggered server method sequence. The following column provides the corresponding symbolic name. The last column then indicates the conditions under which each client method in the sequence (second column) is executed: these conditions are those determined above. When a setting is not mentioned (we use character ‘-’), no tuple from the corresponding client method was selected: i.e., the previous method in the client method sequence does not leave the list in a state that allows the execution of any tuple from the following method in the sequence. Note that highlighted tuples are those removed according to redundancy criterion 3.

<table>
<thead>
<tr>
<th>Tuples for client method sequences</th>
<th>Symbolic name</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 insertAfter.insertBefore.deleteNode, Node</td>
<td>Node</td>
<td>A,C,C</td>
</tr>
<tr>
<td>2 insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.getKey.getData getNext.getKey.getData.getNext.getKey.getData.getNext</td>
<td>n</td>
<td>D,D,D</td>
</tr>
<tr>
<td>3 insertAfter.insertBefore.deleteNode, getKey.getData.getNext.getKey.getData.getNext.getKey.getData.getNext getKey.getData.getNext.getKey.getData.getNext</td>
<td>head</td>
<td></td>
</tr>
</tbody>
</table>

10 Highlighted tuples are those removed according to redundancy criterion 3.
<table>
<thead>
<tr>
<th></th>
<th>Expression</th>
<th>Type 1</th>
<th>Type 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td><code>insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.getKey. getPrevious.setPrevious.getKey.getData.getKey.getData</code></td>
<td>n</td>
<td>E, G, H</td>
</tr>
<tr>
<td>5</td>
<td><code>insertAfter.insertBefore.deleteNode, Node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.getKey.getData</code></td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><code>insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.getKey.getData.getKey.getData.getData.getPrevious.setPrevious.getKey.getData.getKey.getData</code></td>
<td>head</td>
<td>F, F, F</td>
</tr>
<tr>
<td>8</td>
<td>`insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getData.addNodeAfter.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getDatagetKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getDatagetKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getDatagetKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getDatagetKey得到了无 n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>`insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getData.addNodeAfter.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData得到无 n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>`insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getData.addNodeAfter.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getDatagetKey得到了无 head</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>`insertAfter.insertBefore.deleteNode, getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getData.addNodeAfter.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData.getKey.getData getKey得到无 head</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td>Node</td>
<td>A, C, C</td>
</tr>
<tr>
<td>13</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td><code>insertAfter.deleteNode.insertBefore, node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td>Code</td>
<td>Action</td>
<td>Node 1</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------------------</td>
<td>-----------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>35</td>
<td>`insertBefore.deleteNode.insertAfter,</td>
<td>Node</td>
<td>A, C, C</td>
</tr>
<tr>
<td>36</td>
<td>getNew.getData.getKey.getData</td>
<td></td>
<td>D, D, D</td>
</tr>
<tr>
<td>37</td>
<td>`insertBefore.deleteNode.insertAfter,</td>
<td>head</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>getNew.getData.getKey.getData.next.getNew</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>`deleteNode.insertAfter</td>
<td>Node</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>`insertBefore.deleteNode.insertAfter,</td>
<td>head</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>getNew.getData.getKey.getData.next.getNew</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>`deleteNode.insertAfter</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>`deleteNode.insertAfter</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>`deleteNode.insertAfter</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>`deleteNode.insertAfter</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>`deleteNode.insertAfter</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>`deleteNode.insertAfter</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>`deleteNode.insertAfter</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>`deleteNode.insertAfter</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>`deleteNode.insertAfter</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>`deleteNode.insertAfter</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>`deleteNode.insertAfter</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>`deleteNode.insertAfter</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>`deleteNode.insertAfter</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>`deleteNode.insertAfter</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>`deleteNode.insertAfter</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>`deleteNode.insertAfter</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>`deleteNode.insertAfter</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>`deleteNode.insertAfter</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>`deleteNode.insertAfter</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>`deleteNode.insertAfter</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>`deleteNode.insertAfter</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>`deleteNode.insertAfter</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>`deleteNode.insertAfter</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>`deleteNode.insertAfter</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>`deleteNode.insertAfter</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>`deleteNode.insertAfter</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>`deleteNode.insertAfter</td>
<td>H</td>
<td></td>
</tr>
</tbody>
</table>
Table 14 Tuples for Linked List (from client method sequences in Table 7)

The last table in this appendix (Table 15 below) indicates the tuples selected according to the three redundancy criteria we defined in Section 3.4.

<table>
<thead>
<tr>
<th>Redundancy criterion</th>
<th># of tuples</th>
<th>Actual tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Criterion 3</td>
<td>21</td>
<td>1, 2, 3, 4, 8, 9, 10, 15, 16, 20, 21, 26, 32, 33, 34, 38, 44, 46, 51, 59, 84</td>
</tr>
<tr>
<td>Criterion 2</td>
<td>21</td>
<td>1, 2, 3, 4, 8, 9, 10, 15, 16, 20, 21, 26, 32, 33, 34, 38, 44, 46, 51, 59, 84</td>
</tr>
<tr>
<td>Criterion 1</td>
<td>17</td>
<td>1, 2, 3, 4, 9, 10, 15, 20, 21, 26, 32, 33, 38, 44, 51, 59, 84</td>
</tr>
</tbody>
</table>

Table 15 Selected tuples for Linked List according to redundancy criteria