Contextual Integration Testing of Classes*

Giovanni Denaro[1], Alessandra Gorla[2], and Mauro Pezzè[1,2]

[1] University of Milano-Bicocca, Dipartimento di Informatica, Sistemistica e Comunicazione, Via Bicocca degli Arcimboldi 8, 20126, Milano, Italy
denaro@disco.unimib.it
[2] University of Lugano, Faculty of Informatics, via Buffi 13, 6900, Lugano, Switzerland
alessandra.gorla@lu.unisi.ch, mauro.pezze@unisi.ch

Abstract. This paper tackles the problem of structural integration testing of stateful classes. Previous work on structural testing of object-oriented software exploits data flow analysis to derive test requirements for class testing and defines contextual def-use associations to characterize inter-method relations. Non-contextual data flow testing of classes works well for unit testing, but not for integration testing, since it misses definitions and uses when properly encapsulated. Contextual data flow analysis approaches investigated so far either do not focus on state dependent behavior, or have limited applicability due to high complexity. This paper proposes an efficient structural technique based on contextual data flow analysis to test state-dependent behavior of classes that aggregate other classes as part of their state.

1 Introduction

Object-oriented programs are characterized by classes and objects, which enforce encapsulation and behave according to their internal state. Object-oriented features discipline programming practice, and reduce the impact of some critical classes of faults, for instance those that derive from excessive use of non-local information or from unexpected access to hidden details. However, they introduce new behaviors that cannot be checked satisfactorily with classic testing techniques, which assume procedural models of software [1]. In this paper, we focus on structural testing of state-based behavior, which impacts on both unit and integration testing of classes.

The most promising structural approaches to testing object oriented software exploit data flow analysis to implicitly capture state-based interactions. Harrold and Rothermel proposed data flow analysis for structural testing of classes in 1994 [2]. In their early work, Harrold and Rothermel define a class control flow graph to model data flow interactions within classes, and apply data flow analysis to characterize such interactions in terms of flow relations of class state

* This work has been partially funded by the European Commission through the project SHADOWS, by the Italian Government through the project COMMUTA and by the Swiss National Fund through the project PerSeoS.

variables. This analysis supports well unit testing, but does not apply satisfactorily to integration testing of classes. In fact, when accesses to state variables are properly encapsulated, standard data flow analysis does not distinguish chains of interactions that flow through different methods, thus missing several integration dependencies, and in particular dependencies of classes that aggregate other classes as part of their state. In 2003, Souter and Pollock proposed a contextual data flow analysis algorithm for object oriented software [3]. Souter and Pollock's algorithm distinguishes accesses to the same variables through different chains of method invocations, and thus captures inter-method interactions even when variables are encapsulated in classes. Differently from our work, the work of Souter and Pollock does not focus on state dependent behavior. Moreover, Souter and Pollock's algorithm is very accurate, but quite expensive ($O(N^4)$ in the size of the program). The complexity of the algorithm limits the scalability of the approach and the development of efficient tools.

This paper proposes an approach to state based integration testing of classes that properly extends the early approach by Harrold and Rothermel: We use contextual information about method invocations to analyze state-dependent behavior of classes that can aggregate other classes as part of their state. We share the main concept of contextual def-use associations with Souter and Pollock, but with different goals: Souter and Pollock pursue exhaustive analysis of single methods, while we focus on state based interactions between methods that can be independently invoked from outside the classes. Moreover, differently from Souter and Pollock, we extend the algorithm proposed by Harrold and Soffa to capture inter-procedural propagations of definitions and uses by including contextual information [4]. Our algorithm for contextual data flow analysis is more efficient, albeit less accurate, than the algorithm of Souter and Pollock: it is quadratic in the size of the program in the worst case, and more efficient in many practical cases.

This paper describes an efficient contextual data flow analysis approach for object oriented software, discusses the computational complexity, and proposes structural coverage criteria for class integration testing (Sect. 2). It introduces a prototype implementation that we used for evaluating the suitability of the proposed approach (Sect. 3). It presents empirical data that support the applicability of the proposed coverage, and discusses the limits of the approach (Sect. 4). It surveys the main related work, and acknowledges the contribution of previous research (Sect. 5). It concludes by summarizing the contribution of the paper, and by describing ongoing research work (Sect. 6).

2 Contextual Data Flow Testing of Classes

A class is a (possibly) stateful module that encapsulates data and exports operations (aka methods). At runtime, a method takes a class instance (aka an object) as part of its input, and reads and manipulates the object state as part of its action. Thus in general, the behavior of the methods depends on the state of the objects, and the set of reachable states of the objects depends on the

```
1  public class Msg {
2    private byte info;
3    public Msg(){info = 0;}
4    public void setInfo(byte b){
5      info=b;
6    }
7    public byte getInfo(){
8      return info;
9    }
10 }
11 public class Storage {
12   private Msg msg;
13   private byte stored;
14   public Storage(){
15     msg = new Msg();
16     stored = 0;
17   }
18   public void setStored(byte b){
19     stored = b;
20   }
21   public byte getStored(){
22     return stored;
23   }
24   public void recvMsg(Msg m){
25     byte recv = m.getInfo();
26     msg.setInfo(recv);
27   }
28   public void storeMsg(){
29     byte b = msg.getInfo();
30     setStored(b);
31   }
32 }
```

Fig. 1. A sample Java program

methods. The states of the classes are often composed of instances of other classes. To thoroughly test classes, we need to identify test cases, i.e., sequences of method calls, that exercise the relevant state dependencies of the methods in the reachable (composed) states.

Data flow analysis can assist testing of classes: it identifies the relevant dependencies between methods by capturing definitions and uses of state variables across methods [2]. Section 2.1 briefly recaps the minimal background on data flow testing. Section 2.2 discusses the limits of classic data flow analysis in identifying chains of interactions that flow through different classes. As a consequence, classic data flow analysis misses interactions through structured variables properly encapsulated in classes, which are extremely relevant during integration testing. Section 2.3 introduces our approach that leverages previous work on data flow testing of classes, making it amenable for the purpose of class integration testing.

2.1 Def-Use Associations

Given a program variable v, a standard (context-free) def-use association (d, u) is a pair of program locations, definition (d) and use (u) locations for v, where v is assigned a new value at d, and has its value read and used at u. Testing a def-use association (d, u) executes a program path that traverses first d and then u, without traversing statements that define v between d and u (the subpath between d and u is *def-free*). Def-use associations lead to many testing criteria [5].

Methods of a class interact through local variables, parameters, state variables and global objects. Def-use associations capture such interactions by identifying methods that use values set by other methods. For example the def-use association for variable `stored` at lines $(19, 22)$ of Fig. 1 captures the dependency between methods `setStored()` and `getStored()`.

Def-use associations that involve local variables and parameters characterize method interactions through these elements, and are exploited by many tools, e.g., Coverlipse [6], and are not further considered in this paper.

2.2 Contextual Def-Use Associations

Contextual def-use associations extend def-use associations with the *context* of the method invocations. A context is the chain of (nested) method invocations that leads to the definition or the use [3][3]. A contextual def-use association for a variable v is a tuple (d, u, cd, cu), where (d, u) is a def-use association for v, and cd and cu are the contexts of d and u, respectively. This concept is illustrated in Fig. 1: The context-free def-use association for variable `stored` at lines (19,22) corresponds to two contextual def-use associations ($\rightarrow$ indicates method calls):
(19, 22, `Storage::setStored()`, `Storage::getStored()`) and
(19, 22, `Storage::storeMsg()` $\rightarrow$ `Storage::setStored()`, `Storage::getStored()`)

In general, in absence of context information, def-use associations are satisfied by test cases that focus on trivial rather than complex interactions, while context information identifies a more thorough set of test cases. For example, the context-free def-use associations at lines $(19, 22)$ can be satisfied with a simple test case that invokes methods `setStored()` at line 18 and `getStored()` at line 21, while the corresponding contextual def-use associations illustrated above require also the (more interesting) invocations of methods `storeMsg()` at line 28 and `getStored()` (line 21). We clarify the concept through some common design practice: accessor methods and object aggregation.

Accessor methods are used to guarantee controlled access to state variables. A common design style is to define *get* methods (e.g., `getStored()`) to access state variables, and *set* methods (e.g., `setStored()`) to define state variables. Context-free definitions and uses of variables with accessors are always located within the accessor methods themselves, by construction. Thus, all state interactions that involve these variables are characterized by the few (context-free) def-use associations that derive from the accessor methods, while the many interactions mediated by the accessors are not captured. A test suite that covers all (context-free) def-use associations involving accessors would focus on trivial interactions only, missing the relevant ones. Contextual def-use associations distinguish between direct and mediated invocations of accessors, thus capturing also interactions between methods that access state variables through accessors. In the previous example the test cases derived from context-free def-use associations would focus on the trivial interaction between `setStored()` and `getStored()` only, missing the more relevant interaction between `storeMsg()` and `getStored()`, while the test cases derived from contextual def-use associations would capture all interactions through variable `stored`.

[3] In presence of multiple invocations from the same method, context may or may not distinguish the different invocation points. For the goals of this paper we do not need to distinguish different invocation points in the same method.

Table 1. Definitions and uses computed for the sample program in Fig. 1

Method	Line (state var)	Context
	Defs@exit	
Msg::Msg	3 (info)	Msg::Msg
Msg::setInfo	5 (info)	Msg::setInfo
Storage::Storage	16 (stored)	Storage::Storage
Storage::setStored	19 (stored)	Storage::setStored
Storage::storeMsg	19 (stored)	Storage::storeMsg→Storage::setStored
Storage::Storage	15 (msg)	Storage::Storage
Storage::Storage	3 (msg.info)	Storage::Storage→Msg::Msg
Storage::recvMsg	5 (msg.info)	Storage::recvMsg→Msg::setInfo
	Uses@entry	
Msg::getInfo	8 (info)	Msg::getInfo
Storage::getStored	22 (stored)	Storage::getStored
Storage::recvMsg	26 (msg)	Storage::recvMsg
Storage::storeMsg	29 (msg)	Storage::storeMsg
Storage::storeMsg	8 (msg.info)	Storage::storeMsg→Msg::getInfo

Object aggregation indicates the use of an object as part of the data structure of another object. Since it is a good practice to encapsulate the state of an aggregated object within its methods, definitions and uses of the internal state variables are located within these methods. State interactions that involve internal variables of aggregated objects are then characterized by context-free def-use associations that involve methods of the aggregated object only. Test cases that cover context-free def-use associations focus on single objects and not the aggregated ones, thus missing complex and usually semantically more relevant interactions, while contextual def-use associations identify interactions of simple as well as aggregated objects and lead to a more thorough set of test cases. For example the context-free def-use association for variable `info` at lines $(5, 8)$ in Fig. 1 characterizes both the interaction between methods `setInfo()` and `getInfo()` in class `Msg`, and the interaction between methods `recvMsg()` and `storeMsg()` in class `Storage` (which aggregates a `Msg` object as part of its state).

2.3 Deriving Contextual Associations

We compute state-based def-use associations by first identifying contextual definitions and uses that reach method boundaries, and then pairing definitions and uses of the same state variables across methods. In this phase, our main contribution is the redefinition of the classic algorithms for data flow testing: Differently from Harrold and Rothermel, we compute contextual information of definitions and uses, and thus we capture important inter-procedural properties of object oriented software; We adapt the classic Harrold and Soffa's inter-procedural algorithm to contextual definitions and uses in the context of object oriented software; We borrow the definitions of contextual information by Souter and Pollock, but we use a more efficient algorithm that focuses on definitions and uses of state variables that reach method boundaries as illustrated in the next paragraphs.

Table 2. Def-use associations for the sample program in Fig. 1

Class (state var)	Assoc.	Def context	Use context
Msg (info)	(3, 8)	Msg::Msg	Msg::getInfo
Msg (info)	(5, 8)	Msg::setInfo	Msg::getInfo
Storage (stored)	(16, 22)	Storage::Storage	Storage::getStored
Storage (stored)	(19, 22)	Storage::setStored	Storage::getStored
Storage (stored)	(19, 22)	Storage::storeMsg→Storage::setStored	Storage::getStored
Storage (msg)	(15, 26)	Storage::Storage	Storage::recvMsg
Storage (msg)	(15, 29)	Storage::Storage	Storage::storeMsg
Storage (msg.info)	(3, 8)	Storage::Storage→Msg::Msg	Storage::storeMsg→Msg::getInfo
Storage (msg.info)	(5, 8)	Storage::recvMsg→Msg::setInfo	Storage::storeMsg→Msg::getInfo

We illustrate the algorithm through the example of Fig. 1. In the first step, we statically analyze the methods of the classes under test, and we compute two sets of data, *defs@exit* and *uses@entry*. The *defs@exit* set includes all contextual definitions of state variables that can reach the end of the method, i.e., for which there exists at least a (statically identified) def-free path from the definition to an exit of the method. The *uses@entry* set includes all contextual uses of state variables that can be reached from the entry of the method, i.e., for which there exists at least one def-free path from the entry of the method to the use. For each element in the two sets, we record location, related state variable, and context information. Table 1 shows the *defs@exit* and *uses@entry* sets for the code in Fig. 1.

In the second step, we match the information computed in the first step by combining definitions in *defs@exit* and uses in *uses@entry* that relate to same state variables. In this way, we compute the set of contextual def-use associations for the class under analysis. Table 2 shows the complete set of def-use associations for the classes in Fig. 1.

Our data flow analysis implements intra-procedural analysis according to the classic reaching definition algorithm [7, 8]. We then compute inter-procedural relationships by elaborating the inter-procedural flow graph (IFG), as proposed by Harrold and Soffa [4]. We extended the algorithm to propagate the context information on the control-flow edges that represent inter-procedural relationships.

Context tracking in presence of recursive calls and programs with recursive data structures requires specific handling. Recursion of method calls may generate infinitely many new contexts, which may cause the algorithm to diverge. To avoid divergence, at each node of the IFG our algorithm distinguishes only one level of nested calls, by merging contexts that contain repeated subsequences of method calls. Recursive data structures define aggregations of possibly infinite state variables, which may generate unbounded sets of definitions and uses. Our algorithm expands recursive data structures up to one level of recursion. While limiting the depth of recursion and recursive data structures may threaten the completeness of the analysis, we conjecture that one level of depth is enough to test at least once all interactions between distinct modules, and thus this limit should not impact significantly on integration testing.

The extended algorithm works with the same number of propagation steps as the original rapid data flow algorithm, and thus has a temporal complexity of the

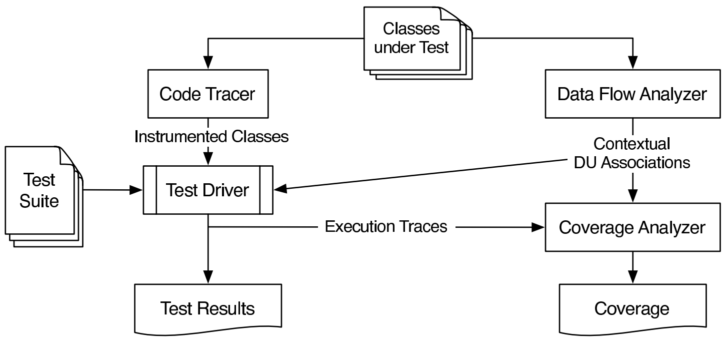

Fig. 2. Logical structure of the **DaTeC** prototype

same order of magnitude as the original one ($O(n^2)$ worst case complexity and linear in most practical situations[4, 7, 8], while Souter and Pollock's algorithm is $O(n^4)$.) Space complexity slightly increases because of the extra space for memorizing context information, and because definitions and uses that reach IFG nodes through different contexts are now tracked as distinct items. The details of the algorithm, omitted here for space limitations, are discussed in [9].

In our experiments, we completed static data flow analysis in few minutes, even for large programs (see Sect. 4 for details.)

3 Evaluation Framework

Contextual def-use analysis captures many non-trivial object interactions that depend on program state. Thus testing all contextual def-use associations should increase the possibility of revealing state-dependent faults, and the confidence in the proper behavior of the program. However, complex interaction patterns may result in many contextual def-use associations, which may be expensive to compute. Moreover, the contextual information may increase the amount of infeasible associations, thus weakening the strength of testing all contextual def-use associations.

To empirically evaluate our approach, we built a prototype. Fig. 2 illustrates the fundamental components of the **DaTeC** (Data flow Testing of Classes) prototype and the information flow to and from a classic testing environment. In a classic testing environment, a *test driver* (for instance *JUnit*), executes a set of *test cases* for the *classes under test*, and produces a set of *test results*. **DaTeC** extends a classic testing environment by computing the contextual def-use associations of the classes under test, and by identifying the contextual def-use associations exercised during testing. The prototype is composed of a *data flow analyzer* that statically computes the contextual def-use associations for the classes under test, a *code tracer* that inserts probes into the classes under test to identify the executed definitions, uses and contexts, and a *coverage analyzer* that identifies the associations executed by a test suite. The *code tracer* has been implemented by instantiating a general purpose monitor. In the next paragraphs, we describe the data flow and the coverage analyzers.

The *Data Flow Analyzer* instantiates contextual data flow analysis for analyzing Java bytecode. Our implementation relies on the JABA API (Java Architecture for Bytecode Analysis [10]) for exploring the control flow graphs of the methods. Following the schema presented in Sect. 2.3, we adapted the traditional algorithms (reaching definitions and its inter-procedural counterpart) to compute contextual def-use associations of state variables for given sets of classes under test.

The *Coverage Analyzer* computes the contextual def-use associations that belong to a set of program traces. A trace T is a sequence of program locations that correspond to a program execution. For each location, a trace records the class and the object involved in the execution at that point.[4] A specific trace $\overline{T}$ covers a contextual def-use association $(\overline{d}, \overline{u}, \overline{cd}, \overline{cu})$ for a state variable $\overline{v}$ if and only if:

- the locations $\overline{d}$ and $\overline{u}$ occur in $\overline{T}$ in this order;
- the variable $\overline{v}$ is instantiated within the same object;
- no killing definition for $\overline{v}$ occurs in $\overline{T}$ between $\overline{d}$ and $\overline{u}$;
- the trace $\overline{T}$ satisfies the contexts $\overline{cd}$ and $\overline{cu}$ at locations $\overline{d}$ and $\overline{u}$, respectively. A generic trace T satisfies a context c at a location l, if, in T, l is reached through a chain c of method invocations. Notice that if T satisfies c at l, it satisfies also all contexts obtained by considering the tails of c.

The algorithm for coverage analysis is computation intensive. The trivial algorithm that scans all traces and checks all def-use associations for each trace has a complexity linear in the product of the number of traces, the length of the traces, and the number of def-use associations. To reduce the execution time and memory consumption, we compare contexts incrementally, and we index the def-use associations by the traversed locations. In this way, we consider only the associations related to the locations in the trace currently analyzed, thus reducing the impact of the total number of associations on the complexity. In our experiments, we processed programs with traces containing up to 10^8 entries, and a total of up to 10^5 contextual def-use associations without problems.

Although the technique is applicable to general Java programs, the prototype does not currently handle exceptions and polymorphism; it does not statically analyze the code for aliases; and it treats arrays with some imprecision. Here we briefly discuss the impact of these limits on the experiments.

Exceptions require special treatment as discussed by Sinha and Harrold [11], Chatterjee and Ryder [12], and Chatterjee et al. [13]. Exceptions are part of the plans for the next release of our prototype. In the current release, the data flow analyzer ignores exception handlers and the coverage analyzer does not report the related coverage.

Polymorphism and dynamic binding can cause an exponential explosion of combinations. Rountev et al. propose a technique to efficiently treat polymorphism based on class analysis that can be integrated with our approach [14]. The

[4] Object identifiers are ignored for `static` methods.

Table 3. Statistics of our contextual data flow analysis on a set of sample programs

	Number of classes	SLOC	State vars	Max assoc. for 95% of classes	Time (in sec)
Jedit	910	92,213	2,975	381	70
Ant	785	80,454	4,081	331	60
BCEL	383	23,631	929	792	24
Lucene	287	19,337	1,013	410	17
JTopas	63	5,359	196	49	8
NanoXML	25	3,279	39	4	5
Siena	27	2,162	66	35	5

SLOC indicates the number of lines of source code, excluding blank and comment lines. SLOC has been computed using SLOCCount (under Linux).
All benchmarks are public open-source software: Ant, BCEL and Lucene are available at apache.org; JEdit is available at jedit.org; Siena, NanoXML and JTopas are available from the SIR repository [19].

current prototype does not solve the bindings, but considers the ones indicated by the user.

Aliases widen the set of possible interactions. We are currently evaluating alias analysis algorithms for inclusion in the next release [15, 16].

Data flow analysis, as most static analysis techniques, cannot handle well arrays, since array elements are often accessed through indexes whose values are computed dynamically. The Java dynamic initialization of arrays worsen the problem. Forgács and Hamlet et al. present some techniques for handling arrays efficiently [17, 18]. The current prototype approximates arrays as a whole, without distinguishing accesses to single element.

The limitations discussed in this section characterize most static analysis techniques, but none of them prevents the applicability of data flow analysis.

4 Empirical Data

We tested the technique proposed in this paper to check for efficiency (the ability of dealing with large programs), feasibility (the portion of infeasible contextual def-use associations), and effectiveness (the ability of revealing failures).

4.1 Scalability

The size of the programs that can be analyzed may be limited by the complexity of data flow and coverage analysis. As discussed in Sects. 2 and 3, our data flow analysis is quadratic in the worst case and linear in many practical cases, while coverage analysis depends on both the length of the analyzed traces and the amount of contextual def-use associations.

To appreciate the impact of the complexity of data flow and coverage analysis, we analyzed a set of sample open-source programs of increasing complexity with our prototype. Table 3 reports the amount of associations and the time for computing them. The first four columns identify the analyzed programs and their complexity (program name, number of classes, lines of code and number of state variables). Column *Time* indicates the overall time for completing the analysis with our prototype running on a Dell PowerEdge 2900 server with two 3.0 GHz Dual-Core Intel Xeon processors and 8 GB of RAM.

Column *Max Assoc. for 95% of classes* indicates the maximum amount of associations computed for a single class, excluding the outliers (the 5% of the classes with the highest number of contextual def-use associations.) We decided to exclude the outliers because most classes present a relatively small number of contextual def-use associations with few exceptions, as illustrated in Fig. 3 that plots the relation between the lines of code and the number of associations in a class for all the sample programs. In the cases considered so far, the outliers are algorithm intensive classes, *tokenizers* or *parsers* for Jedit, BCEL, Lucene and JTopas, which use state variables for storing intermediate results of the computation. Most of these classes are automatically produced with parser generators, and are of little interest from the class testing viewpoint. Thus, we can obtain good testing results even ignoring the associations of these classes.

As shown in the example of Sect. 3, a single test case may cover several associations, thus we can cover the associations with a number of test cases smaller than the amount of associations.

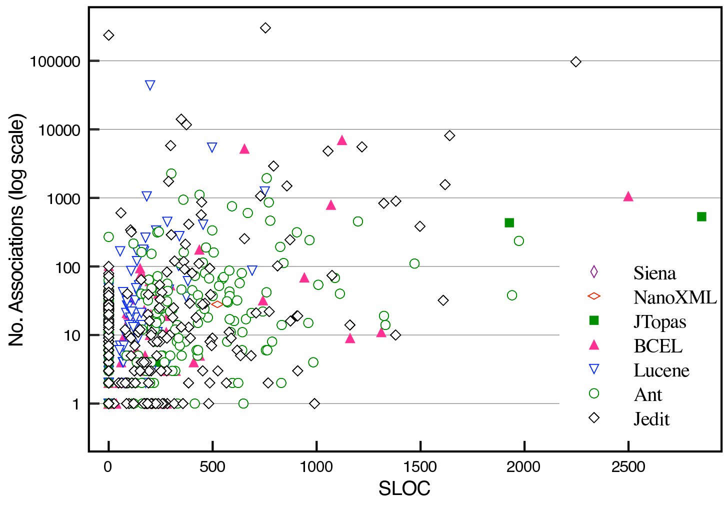

Fig. 3. SLOC and number of associations for all sample programs classes

4.2 Feasibility

Testing criteria may not be effective when they statically identify too many infeasible elements, thus resulting in highly variable coverages. Typical industrially-relevant criteria, such as statement, branch and MC/DC coverage, result in 75% to 95% of feasible elements [1]. To evaluate the impact of infeasible associations, we inspected all contextual def-use associations derived from the packages `junit.framework` of JUnit and a `lucene.document` of Lucene (all the details can be found in [9]). The package `junit.framework` presents a total of 102 contextual def-use associations, and only 3 of them are infeasible. Thus 96% contextual def-use associations are feasible in `junit.framework`. Three associations are infeasible because they involve uses in unreachable code, and none is infeasible due to data flow limitations. The classes of the Lucene package that we inspected contain a total of 354 contextual def-use associations, 280 (80%) of which are feasible. All the infeasible contextual def-use associations derive from the impossibility of executing all associations in all possible contexts, since some classes invoke some methods with a subset of parameter values that restrict the amount of code that can be executed in the specific contexts.

4.3 Effectiveness

We studied the ability of exposing failures by comparing the performance of standard test suites with test suites that guarantee coverage of feasible contextual def-use associations. We ran the experiments on the 3.8.1 distribution of the package `junit.framework` of JUnit, since it is distributed with a standard test suite. We first augmented the test suite provided with the distribution with additional test cases that guarantee 100% coverage of the feasible contextual def-use associations. We then compared the effectiveness of the standard and the augmented test suites by measuring the ability of revealing a set of seeded faults.

The 58 test cases of the standard test suite cover only 52% of the contextual def-use associations (54 out of a total of 102 associations, 99 of which are feasible). We augmented the default test suite with 19 additional test cases to cover all 99 feasible associations. Neither of the suites find any faults, as expected, since JUnit is a mature tool, and the package `junit.framework`, which we analyzed, is the core of the tool.

A qualitative inspection of the additional test cases indicates that the original test suite omits some relevant checks: The test cases of the original suite (1) do not check for the consistency of the initial state of most objects, (2) do not check some functionalities, and (3) do not check the behavior of some methods when invoked in states that violate the preconditions of the methods. For example, in JUnit, test suites can be given a name, but no original test case checks for the correct treatment of this case. Moreover JUnit requires test cases to be defined in public classes, but the behavior in presence of test suites included in private classes is not checked by the standard test suite.

To further assess the performance of the augmented test suite, we evaluated the original and the augmented suites with respect to a set of 83 mutants generated with mutant operators applied to constructs involving the state of the classes. The original suite identifies only 56 out of 83 seeded faults, the original suite augmented with 9 test cases to reach 100% coverage of the feasible statements identifies 65 seeded faults, while the test suite augmented to cover all feasible contextual def-use associations identifies all 83 faults.

This preliminary result suggests that the increment of test cases required to execute all feasible contextual def-use associations is less than the increment in the number of revealed faults. To execute all feasible contextual def-use associations we ran 77 test cases, 33% more than the original suite (58) and 15% more than the suite that executes all statements (67), but we revealed all 83 seeded faults, 48% more than the seeded faults revealed by the original suite, and 28% more than the suite that covers all feasible statements.

4.4 Limitations and Threats to Validity

The main threats to the validity of the empirical results reported in this section derive from the limitations of the prototype used in the experimental work: the language issues that are not addressed by the prototype (exception handling, polymorphism, and reference aliasing) might affect the precision of the computed sets of contextual def-use associations, thus biasing our figures. Alias analysis algorithms, such us the ones proposed by Liang, Pennings and Harrold [15] and Milanova, Rountev and Ryder [16] should improve the precision of the results. We are currently evaluating such algorithms to include alias analysis in the next release of the prototype. Exception handlers may worsen the problem of infeasible associations, but we believe that they should be addressed independently.

The results on fault-detection effectiveness based on mutation analysis may be biased by the set of injected faults, even if they confirm the preliminary feedback from comparing different test suites. We are currently experimenting with known faults in publicly available applications to further confirm the preliminary results of mutation analysis.

Additional validity threats may derive from having experimented only with open-source software, which could not adequately represent software produced in industrial settings. We are working with our industrial partners to confirm the preliminary results obtained with open-source software.

5 Related Work

The problem of testing state-dependent behavior has been originally addressed in the domain of communication protocols, and then further extended to cope with object oriented software [1, 20]. Most work on testing the state-dependent behavior of object oriented software has focused on deriving test cases from state-based specifications, often UML (e.g., [21, 22]).

The most relevant code-based approaches to testing the state-dependent behavior of object-oriented software have exploited data flow analysis techniques. Harrold and Rothermel first, and Souter and Pollock later laid down the foundations [2–4].

Harrold and Rothermel introduced a suitable data flow model of class interactions, and defined intra- and inter-class testing. We based our analysis on this model, and our prototype on the JABA library [10], which extracts the model from Java bytecode. Our work properly extends Harrold and Rothermel's approach by computing contextual information of definitions and uses.

Souter and Pollock introduced contextual def-use associations for better characterizing the interactions within object-oriented software, and proposed an algorithm that analyzes method interactions by examining complete chains of method invocations. Our framework adapts the notion of contextual def-use associations defined by Souter and Pollock, to provide a framework for defining intra-class integration testing strategies.

Souter and Pollock instantiated their approach in the tool TATOO [23] that analyzes inter-method interactions, and used this tool to explore the use contexts of different granularity. TATOO was not publicly available at the time of writing, thus we could not make a direct comparison with our prototype. Based on what published in the literature [3, 23], our prototype seems to scale up better than TATOO: uses of TATOO are reported for programs up to 10 KLOC and 100 classes, while we have successfully analyzed programs up to 100 KLOC and 1,000 classes. On the other hand, our static analysis for building contextual def-use associations is less precise than Pollock and Souter's one. Their algorithm analyzes each method separately for each context in which it can be invoked, and partially accounts for reference aliasing that can differ across different contexts. We ignore aliasing for the moment.

Other tools that address data flow coverage for Java programs, e.g., Coverlipse [6] and JaBUTi [24], consider only intra-method interactions, thus they are not suitable for integration testing.

Several papers propose approaches to increase the precision of the data flow analysis for Java by accounting for reference aliasing, exception handling and polymorphism [13–16, 25, 26]. We are currently extending our prototype to include alias analysis, and improve the precision of the results.

The presence of libraries or components available without source code impacts on the precision of data flow analysis. Rountev et al. propose an approach to inter-procedural data flow analysis that relies on summary information provided with external libraries, and does not require access to the source code [27]. Since our analysis works at the bytecode level, we face this problem only in presence of native code in Java libraries.

6 Conclusions

Classic structural testing approaches do not adequately address subtle failures that may depend on state-dependent behavior of object-oriented classes. Data flow analysis techniques have been recently extended to capture dependencies from instance variables that determine the state of the classes, taking into account the context in which methods that access instance variables are invoked.

In this paper, we propose a framework that adapts and extends previous results to structural integration testing of classes [2], aiming to better support integration testing of classes. Our approach exploits contextual def-use associations defined by Souter and Pollock, but differently from them focuses on state dependent behavior and proposes a more efficient algorithm, thus improving scalability. We report a preliminary set of empirical data obtained though a prototype implementation of the approach on a set of open-source programs.

The analysis proposed in this paper can be performed incrementally on single classes, since definition and use sets can be combined at integration time, while Souter and Pollock's analysis combines the points-to-graphs of the different methods for each method invocation, and thus gains precision at the expense of scalability. Souter and Pollock's approach applies well to the analysis of complete programs and single methods, while our approach fits better the analysis of the interactions of methods that can be invoked independently from outside the class.

The empirical results confirm the complexity results and suggests that the technique proposed in this paper scales up well to mid-size programs (we analyzed programs of up to 100,000 lines of code and 1000 classes in few minutes). The results indicate also a high percentage of feasible contextual def-use associations, thus sustaining the usefulness of the proposed structural coverage. Finally, the results suggest that the coverage proposed in this paper includes relevant state-dependent behavior that are ignored by classic structural coverage criteria.

References

1. Pezzè, M., Young, M.: Software Test and Analysis: Process, Principles and Techniques. John Wiley and Sons (2008)
2. Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. In: Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM Press (1994) 154–163
3. Souter, A.L., Pollock, L.L.: The construction of contextual def-use associations for object-oriented systems. IEEE Transaction on Software Engineering **29**(11) (2003) 1005–1018
4. Harrold, M.J., Soffa, M.L.: Efficient computation of interprocedural definition-use chains. ACM Transactions on Programming Languages and Systems **16**(2) (1994) 175–204
5. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering **SE-11**(4) (1985) 367–375
6. Kempka, M.: Coverlipse: Eclipse plugin that visualizes the code coverage of JUnit tests Open source project on SourceForge.net, http://coverlipse.sourceforge.net.
7. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM Press (1973) 194–206
8. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
9. Denaro, G., Gorla, A., Pezzè, M.: An empirical evaluation of data flow testing of Java classes. Technical Report 2007/03, University of Lugano, Faculty of Informatics (2007)
10. JABA: Aristotele Research Group. Java Architecture for Bytecode Analysis (2005)
11. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Transactions on Software Engineering **26**(9) (2000) 849–871

12. Chatterjee, R., Ryder, B.G.: Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-433, Department of Computer Science, Rutgers University (2001)
13. Chatterjee, R., Ryder, B.G., Landi, W.A.: Complexity of points-to analysis of Java in the presence of exceptions. IEEE Transactions on Software Engineering **27**(6) (June 2001) 481–512
14. Rountev, A., Milanova, A., Ryder, B.G.: Fragment class analysis for testing of polymorphism in Java software. IEEE Transactions on Software Engineering **30**(6) (2004) 372–387
15. Liang, D., Pennings, M., Harrold, M.J.: Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. In: Proceedings of the ACM Workshop on Program Analysis For Software Tools and Engineering, ACM Press (2005) 6–12
16. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transactions on Software Engineering and Methodology **14**(1) (January 2005) 1–41
17. Forgács, I.: An exact array reference analysis for data flow testing. In: Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society (1996) 565–574
18. Hamlet, D., Gifford, B., Nikolik, B.: Exploring dataflow testing of arrays. In: Proceedings of the 15th International Conference on Software Engineering, IEEE Computer Society Press (1993) 118–129
19. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering: An International Journal **10**(4) (2005) 405–435
20. Binder, R.V.: Testing Object-Oriented Systems, Models, Patterns, and Tools. Addison-Wesley (2000)
21. Briand, L.C., Penta, M.D., Labiche, Y.: Assessing and improving state-based class testing: A series of experiments. IEEE Transactions on Software Engineering **30**(11) (2004) 770–793
22. Hartmann, J., Imoberdorf, C., Meisinger, M.: Uml-based integration testing. In: Proceedings of the 2000 International Symposium on Software Testing and Analysis, ACM Press (2000) 60–70
23. Souter, A., Wong, T., Shindo, S., Pollock, L.: TATOO: Testing and analysis tool for object-oriented software. In: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS 2031, Springer (2001)
24. Vincenzi, A.M.R., Maldonado, J.C., Wong, W.E., Delamaro, M.E.: Coverage testing of Java programs and components. Science of Computer Programming **56**(1-2) (2005) 211–230
25. Liang, D., Pennings, M., Harrold, M.J.: Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, ACM Press (2001) 73–79
26. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java using annotated constraints. In: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, ACM Press (2001) 43–55
27. Rountev, A., Kagan, S., Marlowe, T.: Interprocedural dataflow analysis in the presence of large libraries. In: Proceedings of the 15th International Conference on Compiler Construction. LNCS 3923, Springer (2006) 2–16