Dynamic Data-Flow Testing

Mattia Vivanti
University of Lugano
Faculty of Informatics
Lugano, Switzerland 6904
mattia.vivanti@usi.ch

ABSTRACT

Data-flow testing techniques have long been discussed in the literature, yet to date they are still of little practical relevance. The applicability of data-flow testing is limited by the complexity and the imprecision of the approach: writing a test suite that satisfy a data-flow criterion is challenging due to the presence of many test objectives that include infeasible elements in the coverage domain and exclude feasible ones that depend on aliasing and dynamic constructs.

To improve the applicability and effectiveness of data-flow testing we need both to augment the precision of the coverage domain by including data-flow elements dependent on aliasing and to exclude infeasible ones that reduce the total coverage.

In my PhD research I plan to address these two problems by designing a new data-flow testing approach that combines automatic test generation and dynamic identification of data-flow elements that can identify precise test targets by monitoring the program executions.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing)*

General Terms

Measurement, Verification

Keywords

Data-flow testing, dynamic data-flow analysis

1. PROBLEM STATEMENT

Data-flow testing has been investigated since the late seventies as an alternative approach to classic control flow testing criteria. Aiming at more thorough measures of the adequacy of test suites, in the 1976 Herman firstly proposed to estimate the effectiveness of a test suites as the fraction of definitions and uses of variables exercised by test cases [8].

Since then, data-flow testing has been widely investigated and developed, and has shown great potentiality especially in the context of object oriented systems [14, 4, 2, 18, 7, 15, 3]. Despite the amount of work on the topic, data-flow testing is rarely used in practice and the experimental data about its applicability are still limited and inconclusive.

This is mainly due to both the difficulty of generating test suites that guarantee good coverage and the lack of understanding of the scalability of the approach. Data-flow testing criteria require to cover many more elements than control-flow based criteria, and include many test objectives that are either difficult to satisfy or infeasible.

The complex inter-procedural control-flow and the usage of dynamic constructs such as aliasing and polymorphism directly affect the precision of the computation of the coverage domain. Static data-flow analysis techniques that are used to identify data-flow coverage targets both include infeasible elements in the coverage domain, and exclude feasible ones that depend on aliasing and dynamic constructs. As a result, the testing effort for data-flow testing is diverted by the presence of many infeasible elements, and the meaningfulness of the coverage metrics is weakened both by the inclusion of infeasible elements and the exclusion of feasible ones.

Up to date, the few attempts towards automatic data-flow testing tackle the problems of *complexity* and *imprecision* by either targeting simple programs or relying on some strong approximations, like excluding inter-procedural data relations and aliasing [10, 6, 16], thus limiting the scalability and the effectiveness of the approaches.

In my PhD research I want to directly tackle the *complexity* and *imprecision* problems of data-flow testing. I plan to address these two problems by defining a new hybrid technique that combines static and dynamic analysis to obtain more precise data-flow coverage indicators and test cases that satisfy them. In particular, I propose to use dynamic analysis to identify data-flow elements during the execution of tests, and re-combine the dynamically observed elements using small steps of static analysis to obtain not-yet-covered and more precise test targets to be fed to an automatic test case generator.

Identifying test targets dynamically can both help a test case generator to easy the generation of test cases, by providing additional information on the feasibility of methods and elements, thus avoiding to be stuck in the impossible attempt of covering infeasible element; and strengthen the precision of the coverage measurements, by limiting the presence of infeasible elements and including elements that depend on aliasing and dynamic constructs.

2. RELATED WORK

Different techniques for automatic test case generation have been designed to cover control-flow criteria, but little attention has been dedicated to automatic data-flow testing.

A formal approach has been proposed by Hong et al. [9], who applied model checking to derive test cases for data-flow criteria. They represent the data-flow relations in the program using a Kirpke structure, and express the coverage criterion as a set of CTL properties. Then, a counterexample for such properties gives an input value that covers a def-use pair encoded in the property. The paper proposes different CTL properties for different criteria, but applies the approach only to a single flow graph, and does not support inter procedural analysis.

Buy et al. [1] and Martena et al. [11] defined data-flow testing approaches for intra- and inter-class testing, respectively. Their approaches employ symbolic execution to obtain the method pre-conditions that must be satisfied to traverse definition-clear path for each def-use pair, and then they use automated deduction to determine the order of method invocations that allows satisfying the preconditions of interest and the execution of a subsequent use. Current limitations of static analysis and theorem provers limit the applicability of the approach in practice.

Search-based techniques has been investigated for data-flow testing by some researchers, but all of the proposed approaches either have been demonstrated only on small programs or mostly focused on intra-procedural data-flow relations. Wegener et al. [17] firstly expressed data-flow coverage requirements as a "node-node" fitness functions, where the search is guided towards reaching the definition node, and then from there towards reaching the use node. Some experimental results on small example classes have been presented later by Liaskos and Roper [10], and Ghiduk et al. [6]. More extensive results have been presented in the work by Vivanti et al. [16] that extended the data-flow fitness function for evolving a whole test suite. The technique was successfully applied on a large set of programs, but it used a simplified data-flow analysis that excluded alias information and approximated inter class relations.

3. RESEARCH GOAL AND CHALLENGES

In my PhD research I plan to investigate the possibility of automatically generating effective test cases for object oriented systems, exploiting dynamic data-flow information. As discussed above, existing work suggests that data-flow testing can be particularly effective for testing the inter procedural interactions of object oriented systems, but current approaches based on static analysis seem to be strongly limited by their complexity and imprecision. These limits are currently mostly witnesses by examples or demonstrated on small set of simple programs. We then deal with the first research question **RQ1**: what are the limits of static data-flow testing approaches for object oriented systems?

The underling idea of my research is that we can overcome the limitation of static approaches, and strengthen the confidence on the data-flow coverage domain, by employing dynamic analysis. This leads to the second research question, **RQ2**: can dynamic data-flow analysis identify a reasonable set of coverage elements for data-flow testing?

Finally, the last research question investigates the possibility of using the dynamically obtained data-flow information for test case generation, **RQ3**: how effective are test cases generated using dynamic data-flow analysis?

Table 1: *ExperimentA*, metrics and coverage data for each project.

		Test Cases			Coverage (median)		
Prj	#eloc	Bundl.	Gen.	Total	eloc	branch	defuse
JFreeChart	55 k	2022	24462	26484	0.93	0.83	0.50
JGAP	15 k	1398	5411	6809	0.86	0.78	0.29
Collections	13 k	12836	7768	20604	1.0	1.0	0.33
Lang	11 k	2051	9225	11276	1.0	1.0	0.0
XmlSecurity	10 k	89	7520	7609	0.78	0.65	0.12
JTopas	3 k	209	1583	1792	0.98	0.90	0.27

4. RESEARCH APPROACH AND EXPECTED CONTRIBUTIONS

I plan to answer **RQ1** empirically with a set of experiments that measure the impact of infeasible and hard to execute elements identified with state of the art data-flow approaches. I expect to collect quantitative information about the effectiveness of data-flow criteria and confirm the general hypothesis of a big impact of infeasibility and complexity on covering complex software systems.

I plan to answer **RQ2** by defining a technique for dynamic data-flow analysis. Such analysis, that we call DReaDs (Dynamic REAcing Definition analysiS), will identify data-flow elements revealed while executing object oriented software systems. To this end, DReaDs will monitor the definition, usage and propagation of values while the application is executed (for example, with a test suite), and then abstract and merge the information collected on multiple traces to extrapolate data-flow test targets.

DReaDs will resolve alias relationships and identify the data-flow elements according to the state of the references in the system. For instance, DReaDs will both identify all the objects impacted by the definition of a value and precisely monitor data structures like arrays and collections.

To identify not-yet-covered test targets, DReaDs will include a small step of static analysis. The test targets that I am interest into are pairs of reachable definitions and uses of the same variables that occur in different methods [7]. DReaDs will compute such elements by pairing (dynamically observed) definitions that can reach the exit points of some methods with (dynamically observed) uses that can be reached from the entry point of some other methods.

The main contribution of my PhD thesis will be the study of the possibility of merging dynamic analysis and data-flow analysis, to provide more precise data-flow information than existing approaches.

The test targets identified with DReaDs will be fed to the new approach that I am going to define to answer **RQ3**. I will study the interplay between DReaDs analysis and different coverage-driven test case generation approaches, aiming to design a new technique for data-flow testing. DReaDs and test generation tools have complementary requirements, the dynamic analysis needs test cases to identify test targets, and the test case generator needs test targets to steer the test generation process. I will investigate if and how a synergic combination of the two techniques is possible, by defining a technique that alternates steps of dynamic analysis and test case generation to iteratively generate new test targets and test cases that cover them.

The study of a new framework for data-flow testing will contribute to the state of the research providing techniques for automatic data-flow testing, and data about the effec-

Table 2: `defs@exit` identified with DaTeC and DReaDs, with statistics per class and cumulative size of the difference sets

	d^{DT}: `defs@exit` with DaTeC				d^{DR}: `defs@exit` with DReaDs				Statically missed:	Never observed:
Application	Total	Q_1	Median	Q_3	Total	Q_1	Median	Q_3	$\#(\in d^{DR} \wedge \notin d^{DT})$	$\#(\in d^{DT} \wedge \notin d^{DR})$
Jfreechart	20,513	2	9	38	89,415	3	18	75	85,079 (95%)	3,480 (17%)
Collections	3,908	2	4	12	63,460	4	26	81	62,169 (98%)	1,779 (46%)
Lang	1,227	2	3	14	1,638	2	5	13	1,122 (69%)	409 (33%)
Jtopas	1,481	3	12	16	8,380	6	39	320	8,026 (96%)	600 (41%)
JgraphT	1,800	2	4	16	6,602	1	7	44	6,080 (92%)	505 (28%)
	28,929	2	6	18	169,495	3	17	68	162,476 (96%)	6,773 (23%)

tiveness of (dynamic) data-flow testing with respect to state of the art approaches for testing object oriented systems.

5. CURRENT STATUS

In the first two years of my PhD I mainly focused on **RQ1** and **RQ2**. In this section I describe the current status of my research and the preliminary results.

5.1 Limits of Static Data-Flow Criteria

So far, I evaluated the effectiveness of static data-flow testing criteria with two experiments. In the first experiment, I checked the ability of consistently achieving high data-flow coverage. I selected five Java projects coming with an existing (good) test suite. I augmented the existing suites with the test cases generated with state of the art tools, Randoop [13], CodePro AnalytiX[1] and EvoSuite [5]. I computed both branch and data-flow coverage, and I compared the results. Table 1 reports the data on the size of the subject applications (column *eloc*), the size of the test suites (columns *Test Cases*) and the obtained coverage (columns *Coverage*). We observe that the enhanced test suites consistently achieve high control-flow coverage, but fairly low data-flow one. We confirmed the data statistically: a paired Student's t-test supports that the mean of data-flow coverage per class significantly differs from the mean of statement and branch coverage that achieved on average 44%, 85% and 78% coverage respectively. These results confirm that achieving high data-flow coverage is much more complex that achieving high control-flow coverage.

To further investigate the reasons for the low data-flow coverage, I first defined DReaDs to collect dynamic data-flow information, as discussed below, and then performed a second experiment to compare dynamically and statically identified data-flow elements.

I selected five Java projects, and I computed both static and dynamic data-flow information. I expressed the data-flow information in terms of definitions that can reach an exit point of a method (`defs@exit`). I collected the information with DaTeC, a tool to collect data-flow coverage targets statically [3], and DReaDs, a first prototype of the dynamic technique that I will describe below.

Table 2 reports the data of the experiments. The first two columns (`defs@exit` with DaTeC and `defs@exit` with DReaDs) report the goal number of `defs@exit` in the two cases (static and dynamic data-flow analysis) and the median and quartile per class. The number of definitions dynamically identified is consistently higher than the one statically identified. The last two columns of Table 2 report the amount of `defs@exit` that were revealed with DReaDs but missed with DaTeC, and the amount of `defs@exit` that were statically identified but not dynamically observed, respectively. The data indicate that the imprecision of the outcome of static analysis is dominated by the statically missed relations (false negatives, 96%) over the not observed relations (potential false positives, 20%), and support the hypothesis that statically-identified data-flow elements miss several data-flow relations that could be relevant to cover while performing data-flow testing.

[1]https://developers.google.com/java-dev-tools/download-codepro

5.2 Dynamic Data-Flow Analysis

The core work of my PhD so far has been the definition of DReaDs, a new dynamic data-flow analysis approach for object oriented systems. DReaDs is a technique to dynamically perform reaching definition analysis and compute data-flow coverage.

The underlaying idea of DReaDs analysis is to use dynamic information about references between objects to precisely identify data-flow elements. One of the main limitations of static approaches is that they identify data-flow elements looking at the static declarations of variables in the source code, thus introducing a strong approximation: the propagation of the internal state of an object its not statically seen when the object is declared using an interface or a superclass, and in general all the alias relationships are not captured. DReaDs exploits dynamic analysis to overcome these limitations.

DReaDs identifies data-flow elements more precisely by intercepting data events during the program execution, and identifying data-flow elements affected by those events using a model of the relations between active objects in memory that is built and maintained at runtime.

The model is a graph that represents the existing instances in memory (nodes) and the references between them (edges). DReaDs monitors write and read events in memory and when observes one of such events, it navigates the model to retrieve the set of objects whose (possibly nested) internal state is defined or read in that particular instant. For instance, if a modified object is in that moment part of the state of two other objects, DReaDs registers two definitions, one of the internal state of the first and one of the internal state of the second object.

DReaDs aims not only to identify data-flow elements but also to analyze the propagation of the assigned values through the code that can be executed thereafter (reaching definition analysis). To this end DReaDs maintains a map of active definitions in memory and computes the dynamic reaching definitions incrementally for each basic block executed along a program execution according to the classical data-flow analysis equation [12].

At runtime, DReaDs distinguishes data-flow elements depending on the different instances that are affected by a data event. At the end of each execution, DReaDs abstracts the information on instances identifying the data-flow elements using only their class type, and merges the abstracted information collected on multiple traces in a single report. DReaDs abstract the collected information in a format compatible to classes to be comparable with the data-flow information computed with static analysis tools.

So far, I defined the main features of dynamic data-flow analysis, and implemented a first version of DReaDs to experiment with Java programs. I used the prototype in the experiments described above and I am currently working on additional experiments to understand advantages and limitations of the approach and tune it.

5.3 Automatic Dynamic Data-Flow Testing

In the reminder of my PhD I plan to work on a technique to generate test cases that cover dynamically-identified data-flow targets. In a nutshell, the new technique will combine DReaDs analysis, static analysis and a coverage-driven test case generation approach, alternating phases of dynamic and static analysis to identify coverage elements, with phases of test case generation to cover them and discover new data-flow targets. To identify coverage elements using DReaDs, I extend DReaDs with a step of static analysis. DReaDs dynamically computes a set of definitions and uses from the execution of the program with an initial test suite. It then statically couples definitions that reach the exist of the methods with uses that are reachable from the beginning of the methods, identifying a new set of test targets (i.e. never covered definition use pairs) that focus on inter class testing, to be covered with a test case generation approach.

In my work, I plan to study the interplay between dynamic data-flow analysis and test generation techniques incrementally. I plan to start with a simplified framework based on a simple generation technique to investigate experimentally possible problems of the new approach. I am currently modifying a feedback random testing approach to pair higher probabilities methods that have been seen to propagate to the exit definitions of variables, with methods that use those variables. I plan to extend the current prototype to highlight strengths and weaknesses of the combination of the two approaches and identify new directions of improvement.

6. EVALUATION PLAN

To evaluate my approach I need to evaluate the failure detection ability of the test suites generated with dynamic data-flow testing. I plan to design several experiments of increasing complexity using both known and seeded bugs, measuring the effectiveness of the approach as the amount of real or seeded faults found in target applications.

Known faults require considerable effort to be reproduced and detected in absence of assertions, so I can likely use only a limited set of them. I plan to complement known faults with faults seeded with mutation analysis and to use mutation score as a measure of the effectiveness of the approach.

I will compare the results with test suites written by the developers and test suites obtained with state of the art automatic techniques that use different strategies.

As a side effect I plan to provide a thorough evaluation of data-flow testing and comparison with control flow testing.

7. REFERENCES

[1] U. Buy, A. Orso, and M. Pezzè. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 39–48. ACM, 2000.

[2] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15:272–278, 1989.

[3] G. Denaro, A. Gorla, and M. Pezzè. Contextual integration testing of classes. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pages 246–260. Springer-Verlag, 2008.

[4] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14:1483–1498, 1988.

[5] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*, pages 416–419. ACM, 2011.

[6] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis. Using genetic algorithms to aid test-data generation for data-flow coverage. In *Asia-Pacific Software Engineering Conference*, pages 41–48. IEEE, 2007.

[7] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Proceedings of Symposium on Foundations of Software Engineering*, pages 154–163. ACM, 1994.

[8] P. M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8:92–96, 1976.

[9] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proceedings of the International Conference on Software Engineering*, pages 232–242. IEEE, 2003.

[10] K. Liaskos and M. Roper. Hybridizing evolutionary testing with artificial immune systems and local search. In *Software Testing Verification and Validation Workshop*, pages 211 –220, april 2008.

[11] V. Martena, A. Orso, and M. Pezzè. Interclass testing of object oriented software. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 135–144. IEEE, 2002.

[12] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[13] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering*, pages 75–84. IEEE, 2007.

[14] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11:367–375, 1985.

[15] A. L. Souter and L. L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering*, 29:1005–1018, 2003.

[16] M. Vivanti, A. Mis, A. Gorla, and G. Fraser. Search-based data-flow test generation. In *Proceedings of the International Symposium on Software Reliability Engineering.* IEEE, 2013.

[17] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43:841–854, 2001.

[18] E. J. Weyuker. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering*, 16:121–128, 1990.