Dodona: Automated Oracle Data Set Selection

Pablo Loyola[‡], Matt Staats[*], In-Young Ko[†], and Gregg Rothermel[⋆]

[‡]Dept. of Industrial Engineering
University of Chile
Santiago, Chile
ployola@ing.uchile.cl

[*]SnT Centre
University of Luxembourg
Luxembourg
matthew.staats@uni.lu

[†]Div. of Web Science & Technology
KAIST
Daejeon, South Korea
iko@kaist.ac.kr

[⋆]Dept. of Comp. Science
Univ. of Nebraska-Lincoln
Lincoln, NE
grother@cse.unl.edu

ABSTRACT

Software complexity has increased the need for automated software testing. Most research on automating testing, however, has focused on creating test input data. While careful selection of input data is necessary to reach faulty states in a system under test, test oracles are needed to actually detect failures. In this work, we describe DODONA, a system that supports the generation of test oracles. DODONA ranks program variables based on the interactions and dependencies observed between them during program execution. Using this ranking, DODONA proposes a set of variables to be monitored, that can be used by engineers to construct assertion-based oracles. Our empirical study of DODONA reveals that it is more effective and efficient than the current state-of-the-art approach for generating oracle data sets, and can often yield oracles that are almost as effective as oracles hand-crafted by engineers without support.

1. INTRODUCTION

A *test case* is composed of two essential elements: *test input data* and *test oracles*. Test input data consists of values passed or provided to the system under test, while test oracles are the artifacts used to judge the correctness of the system's execution. Both test input data and oracles impact the effectiveness of test cases – test input data determines what behavior the system will exhibit, while test oracles determine what failures (and hence, ultimately, what faults) can be detected [21]. Most work on automating testing, however, focuses on issues related to test inputs, while largely ignoring the impact of test oracles.

A small, but growing body of work has recognized the value of creating test oracles that are tailored to specific test inputs [9, 14, 18, 21]. While several approaches for automatically generating such oracles have been proposed, most of these approaches attempt to completely automate the process. This results in a "generate and fix" approach, whereby the generation process produces effective test oracles, but only if developers can correct the output from the tools, a challenging task [8, 19]. In contrast, in this work, we seek not to completely automate oracle generation, but instead to *support* test engineers in the construction of *expected value test oracles*. Expected value test oracles are oracles that specify, for a single test input, the concrete expected value for one or more program values.

Our interest in expected value test oracles stems from their role in automatic test case generation. When generating test cases, automated test case generation techniques can typically fully generate only test inputs, because without a formal program specification, techniques cannot specify what it is for an execution to be "correct". In practice, it is then up to test engineers to define the expected behavior of the system under test. In this context, manual oracle generation can be difficult, because, having not constructed the test inputs, test engineers may find it difficult to understand expected program behavior for those inputs, or to know where to look for failures [5]. We believe that by providing testers with recommendations as to what oracles should consist of (i.e., what aspects of system state are worth monitoring), we can make oracle construction easier, and maximize the potential return on engineers' efforts.

In prior work, we presented an approach for supporting oracle construction based on specifying an *oracle data set* – a set of variables for which expected values can be defined. While effective in the domain in which it was employed (avionics systems), this approach's generalizability is limited for two reasons: lack of scalability due to its reliance on mutation analysis, and a simple model of program observability that assumes that an oracle can consider a program's entire state. These problems limit the approach's applicability, making it difficult to apply it to other areas such as object-oriented unit testing, where much of the work on automatic test case generation currently exists [20].

In this work, we present DODONA, a system that implements a new approach for specifying oracle data sets when unit testing Java applications. DODONA is applied for each test case. Initially, a test input is executed, and DODONA monitors the relationships that occur between variables during execution (i.e., via dataflow analysis). Following this, DODONA ranks the relevance of each program variable using techniques from network centrality analysis. DODONA then maps variables to observable points, i.e., methods and public variables. Finally, DODONA recommends an oracle data set for the given test input. A test engineer can then define an expected value oracle for the given test input, confident that their effort is directed towards aspects of the system behavior that are relevant under that input.

DODONA overcomes the obstacles preventing prior work on

ISSTA '14 Bay Area, California, USA

oracle data selection from being applied to Java unit testing. This, in turn, addresses a long standing issue with automatic test case generation for such programs; namely, while test case generation tools have become increasingly competent at generating test inputs, they provide little guidance to test engineers concerning how to use such tests, i.e., how to define the necessary test oracles. DODONA fulfills this need, and does so efficiently and independent of the method used for test case generation.

We evaluated DODONA against the current state of the art oracle data set selection approach (based on mutation analysis) using nine open source Java programs drawn from prior work on automatic test generation [9]. Our results indicate that DODONA is both more efficient and more effective than mutation-based previous approach. In the case studied, DODONA required 17.3%-89.8% less time than the mutation-based approach to generate oracle data sets. Furthermore, for four of the nine Java programs studied, the oracle data sets generated via DODONA were clearly more effective at detecting faults producing improvement in fault finding up to 115%, whereas for only two programs was the older mutation-based approach more effective (effectiveness on other programs was comparable). These results represent a clear improvement over the state-of-the-art approach, and bring practical oracle data selection to a much wider set of software domains.

2. ORACLE DATA SET SELECTION

DODONA operates in a context in which test input data has already been generated. Extensive research has been performed on automatic test input generation, and various promising approaches exist [3]. In some cases, these approaches include methods for creating test oracles, but such approaches always — albeit often implicitly — require manual intervention by test engineers to inspect and correct the results [9, 7]. Evidence supporting the effectiveness of these approaches is mixed, with user studies noting a tendency for test engineers to accept incorrect oracles [19, 8].

In this work, our goal is to avoid the "generate-and-fix" paradigm. Thus, with DODONA, we do not attempt to fully automate the construction of test oracles. Instead, DODONA is meant to assist test engineers who use existing test case generation techniques by *supporting* their construction of test oracles. For each test input, DODONA specifies an *oracle data set*: a set of elements to be used to construct a test oracle for that input. DODONA's goal is to select oracle data sets that are likely to reveal faults relative to given test inputs. Using oracle data sets, the test engineer's efforts can be directed to where they are most likely to have impact, saving them the effort required to understand which aspects of program state are most relevant.

In prior work, we developed an approach for oracle data set selection based on mutation analysis; here, we refer to this approach as *mutation-analysis oracle data selection*, or MAODS. As noted in Section 1, this approach suffered from limited generalizability, rendering it difficult to apply in the context of object-oriented unit testing, where much of the work on automated test case generation has occurred.

In this work, we propose a new approach suited for use with object-oriented unit testing – specifically, unit testing for Java programs. We address two issues that prevent deployment of MAODS on general Java programs: observability and scalability. First, MAODS does not consider observability issues relative to the system under test, i.e., it does not distinguish between public/private variables and methods. This is inappropriate when testing Java programs, for which test oracles are typically based on observer methods (e.g., `get` methods) and public variables. Second, mutation analysis has scalability issues in the context of oracle data selection. For the class of systems studied in our prior work — critical avionics systems — these issues were manageable. However, Java systems can be quite large, and in our experience, the number of mutants created for a system must scale with the size of the system in order for MAODS to consistently create effective oracle data. In situations in which mutation analysis is also used to construct test inputs this may be acceptable [9], but in other cases (e.g. symbolic execution, random generation) we do not wish to follow an expensive test input generation process with an expensive oracle data selection process.

2.1 Overview of Dodona

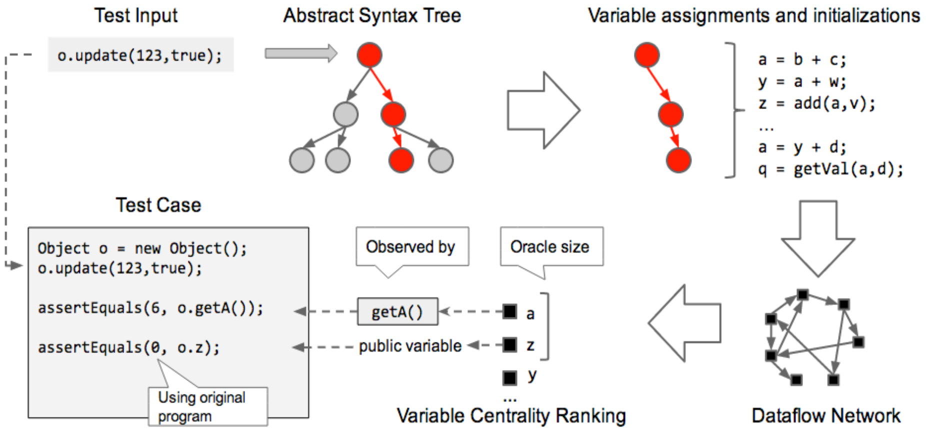
DODONA relies on three assumptions: (1) erroneous values in program variables propagate to further uses (both direct and transitive) during program execution, (2) variables whose values are influenced by many other variables are more likely to contain erroneous values than other variables, and (3) the likelihood that an erroneous value will propagate to a variable decreases as the number of intermediate computations (computations lying between the occurrence of the erroneous value and a later use of that value increases. An effective oracle data set, then, should consist of a small set of variables that are computationally highly related to other program variables.

Figure 1 provides a visualization of the approach used by DODONA. To identify variables that meet these criteria, DODONA utilizes dynamic data flow analysis and network centrality techniques. Dodona first uses data flow analysis to construct a network of program variables for each test input (Algorithm 1, top of Figure 1). Next, Dodona uses network centrality metrics to rank variables in terms of relevance or centrality to the resulting network (Algorithm 2, lower right Figure 1). Dodona uses this ranking to create an oracle data set for each test input. In both of Dodona's algorithms, steps are taken to ensure that the oracle data set is constructed in terms of *observable points*, i.e., public member variables and public method calls (bottom left Figure 1). In the next two subsections we describe the two algorithms, in turn.

2.2 Building a Variable Relationship Network

Given a test case t and program P, DODONA uses Algorithm 1 to map the flow of information from the input data in t to all potentially observable points in P. This is done by executing t (in our case a JUnit test case) and tracking the flow of data from t's inputs (i.e., parameters of method calls in the JUnit test), through intermediate variable assignments and method calls to member variables in objects. When tracking data flow, care is taken to consider the context of the variable, including variable scope and the method call used to reach the variable. This information is used later to determine how to observe the variable.

For example, consider the statement `int c = a + b`. Here, c is initialized using operands a and b, resulting in a unidirectional relationship from a and b to c, that we denote as $c \leftarrow a$ and $c \leftarrow b$ (line 6 of Algorithm 1). In addition, when c is added to the adjacency matrix, its dynamic scope as part



```
o.update(123,true);
```

```
a = b + c;
y = a + w;
z = add(a,v);
...
a = y + d;
q = getVal(a,d);
```

```
Object o = new Object();
o.update(123,true);

assertEquals(6, o.getA());

assertEquals(0, o.z);
```

Figure 1: Visualization of the steps taken by Dodona. Note the final step in which assertions are inserted is to be done manually using the oracle data set produced by Dodona.

Algorithm 1 Test Input Dataflow Recording

```
Require: Test case $t$
Require: Program $P$
 1: $adjMat = \emptyset$
 2: map $obs = \{\}$
 3: while $t$ runs over $P$ do
 4:     for all $y_i$ in $x := y_1$ op $y_2$ ... $y_n$ do
 5:         $s = getCurrentScope()$
 6:         $adjMat = adjMat \cup (s.x, s.y_i)$
 7:         if $s.x \notin obs$ then
 8:             $obs[s.x] = getCurrentMethodCall()$
 9:         end if
10:     end for
11: end while
12: return $adjMat, obs$
```

of a method call (and an object) is also captured, along with the method call's parameters (i.e. the values of *a* and *b*), as shown on lines 5, 6, and 8 of Algorithm 1. As each instruction is executed sequentially, the network is built with each relationship represented as a new edge. At the end of the test case's execution, the algorithm produces, in `adjMat`, an asymmetric adjacency matrix, and a map for each variable to the method call that references that variable.

We have implemented this analysis in Dodona by using Java Pathfinder (JPF) (version six) [24]. JPF is an open-source framework for executing and verifying Java bytecode. The framework consists of an extensible custom Java Virtual Machine (JVM) and listener support for monitoring and influencing JPF's search. Dodona's dynamic data flow analysis is implemented via JPF listeners that monitor Java's execution. Specifically, when running a Java test, Dodona monitors all executions of bytecodes which result in a value being assigned, method calls, and method returns. When an assignment (of any kind) is performed, Dodona extends and updates the Java adjacency matrix accordingly, tracking the relationships between operands and the assigned variable, and recording the scope, the current method call, and what source code variable (if any) the assignment corresponds to. Thus "variable" in this context refers to operands used by bytecode assignment(s), and thus arrays (for example) consist of multiple variables; one for each element in the array.

When a method call occurs, Dodona tracks the flow of information from variables used as method parameters in the current method to the called method. Note that by constructing the adjacency matrix at the bytecode level, Dodona avoids potentially troublesome issues related to, for example, method calls as operands, e.g., `x = a + someMethod(b)`— during compilation, these operations are reduced to assignments to temporary variables.

2.3 Ranking Variables in Terms of Relevance

Dodona's goal in selecting variables is to "cover" all variables, i.e., to ensure that all computed values propagate to the test oracle, and do so in a minimal number of intermediate computations. However, a tradeoff exists when selecting variables for a test oracle: we must often choose between a variable that covers previously uncovered variables, and one that reduces the number of intermediate computations performed on variables already covered.

In prior work on test case prioritization, we developed a metric for measuring how well a set of program variables is covered by a test suite based on a variable adjacency matrix [20]. We have subsequently discovered, however, that our metric is essentially an example of a *network centrality metric*. Network centrality metrics measure the relative importance of nodes within a graph, and are frequently applied in the analysis of social networks to measure the influence of individuals [10]. Increasingly, these metrics are used in software engineering contexts to measure the importance of connected components in software, e.g., to measure the importance of program dependencies [28, 6]. Given that our chief technical challenge in selecting oracle data is identifying the most relevant variables in the flow of program execution, these metrics are a natural fit, and have the benefit of years of careful study behind them.

In this work, therefore, we have used network centrality metrics to allow Dodona to measure the importance — and hopefully, the fault finding ability — of the variables in the variable adjacency matrix. We outline this process in Algorithm 2. After applying a network centrality metric to the adjacency matrix (line 1), Dodona filters and maps the list of variables with centrality scores, retaining only those variables that are either public variables, or that have public scope, i.e., that are referenced by a public method call.

Variables referenced by a public method call are mapped to the appropriate method call (often a "get" method). Finally, Dodona sorts the mapped list by descending centrality score. This is visualized on the bottom of Figure 1, which shows us moving from a network of variables to, in the end, calls inserted into a Java test suite.

Algorithm 2 Ranked Observable Points Computation

```
Require: Adjacency matrix $adjMat$
Require: Observable mapping $obs$
Require: Network centrality metric $ncm$
$varValues$ = $ncm(adjMat)$
$mappedList = []$
for all $(var, value) \in varValues$ do
    if $isPublicVariable(var)$ then
        $mappedList += (var, value)$
    end if
    if $isPublicScope(var)$ then
        $mappedList += (obs[var], value)$
    end if
end for
$ranking = sortByValue(mappedList)$
return $ranking$
```

2.3.1 Network Centrality Metrics

A network centrality metric consists of a function f that computes, for all nodes n in a graph G, a *centrality index* $f(n)$. Many centrality metrics have been proposed, and when implementing Dodona, it was not clear how effective various centrality metrics might be. We therefore allowed the metric to vary to empirically compare the effectiveness of several metrics in the context of oracle data selection.

We explore four network centrality metrics in this work:

Degree centrality. Given graph G, the degree centrality of a node $n \in G$ is defined as $deg(n)$, i.e., the number of other nodes connected to n. In our context, this represents the number of operands used to compute a variable.

Closeness centrality. The closeness of a node $n \in G$ is defined as the inverse of the sum of its distance to all other nodes in G. Thus, as the distance from node n to other nodes decreases, its closeness increases. Closeness is often interpreted as a metric indicating how much time is required for information to propagate. The closeness of a variable v represents, roughly, how far an error must propagate from some variable to reach v.

Betweenness centrality. The betweenness of a node $n \in G$ is the frequency with which n must be traversed when traveling the shortest path between any two nodes $n_1, n_2 \in G$. A high score for a variable v indicates that v often stores an intermediate computation.

Eigenvector centrality. Eigenvector centrality assigns a node $n \in G$ a high score if it is adjacent to nodes that have high scores. A high score for a variable v indicates that v is computed using other influential variables.

The foregoing metrics are discussed in further detail in [10]. Note that in our context, we are concerned with data flowing to a variable, and thus our computations are based on the *in-degree* of a node/variable, i.e. the number of edges directed at the node. We implemented the computation of all four centrality metrics in Dodona using the *JUNG* framework, an open-source Java library for graph-based computations [2].

2.3.2 Mapping Variables to Observable Points

After applying the centrality metric, Dodona must map each variable to an observable point, and filter out any variables that cannot be mapped. This is accomplished using information recorded during dataflow analysis. Public variables do not need to be mapped; they can be referenced directly as `object.variable`. For each non-public variable, Dodona first checks whether the method call in which the variable was observed is public. If so, this method call, with the parameters used when the variable was observed, is used. If the variable was at no point observed in a public method call, it is considered unobservable, and Dodona removes it from the ranking.

In theory, our approach can result in inaccurate mappings due to changes in program state. Specifically, after recording the method call used to observe a variable, it is possible that calling that method a second time may, in fact, not access that variable a second time. In practice, however, many variables are mapped to accessor methods or are otherwise accurately mapped. We considered alternative methods of mapping variables, including static analysis and Java reflection, but concluded that these methods were too expensive to justify using, given the the small number of mistakes that must be corrected.

2.4 Construction of Test Oracles

Using the foregoing analysis, Dodona produces a list of observable points for each test input, ordered by their importance according to a network centrality metric. To construct an oracle data set, a test engineer selects the top n observable points from the ranked list, with n determined by the engineer according to the level of effort he or she believes is warranted. The engineer then constructs a complete test oracle, by defining expected values for each element in the oracle data set and placing them after the test input. In JUnit testing, the engineer will construct an *assertEquals* call for each variable, asserting that the variable has the value he or she expects for the given test input.

In prior work, we provided a method for estimating an effective size n. In this work, we do not use such a method. The prior analysis was based on estimating the point of diminishing returns on testing effort using mutation analysis. While a similar analysis could be performed here, we believe that estimating diminishing returns using a centrality metric — an abstraction of variable importance — is not conceptually sound. Furthermore, in practice, testers typically construct only 1-4 assertions, with the size of the oracle determined via tester judgement [8]. We therefore believe that any suggestion about oracle data size might not only be conceptually unsound, but also likely to be ignored.

3. EVALUATION

We had two goals when evaluating Dodona. We wished to first determine what network centrality metric is typically the most effective with respect to fault finding with Dodona or, failing that, develop a set of guidelines. Second, we wished to assess the effectiveness and the cost of using Dodona to specify oracle data sets.

In this evaluation, we do not yet consider data on human effort. Studies of humans are expensive, and before conducting them, it makes sense to first determine whether techniques possess qualities necessary to enable their effectiveness. This is typical when evaluating testing approaches;

early work refines the approach, after which human studies begin to rigorously assess the human factor (e.g. fault-localization [16], invariant generation [19]).

We designed an empirical study to explore the following research questions:

RQ1. How do different centrality metrics used impact the effectiveness of Dodona?

RQ2. Is Dodona more effective than the existing state-of-the-art approach for specifying oracle data sets?

RQ3. Is Dodona more effective than oracle data specified by developers?

RQ4. What is the cost associated with using Dodona to specify oracle data sets?

3.1 Objects of Study

As stated in Section 2, one of our original goals in developing Dodona was to bring oracle data set specification to Java testing. We therefore wished to apply our technique to programs that: (1) have limited observability, and thus present the a challenge for oracle data set specification; (2) have associated, manually constructed Java unit tests (for comparison); and (3) are amenable to the use of test case generation techniques.

We thus chose as objects of study the set of libraries used by Fraser et al. [9]. These objects exhibit the types of observability issues that motivated the development of Dodona, ranging from APIs to data structures to support libraries. Additionally, each object program has an associated test suite constructed by their developers, together with a set of oracles constructed by their developers. Finally, having been utilized in a prior test case generation study, these objects are known to be well suited to automatic test case generation for object oriented programs.

Table 1: Object Program Characteristics

Object Program	Pckgs	Classes	Lines	Test Cases	Branch Coverage
CLI	1	21	882	187	92%
CDC	6	85	3131	616	93%
COL	16	447	11311	13677	77%
LOG	2	28	1500	26	-
MTH	62	1063	41228	4993	84%
PRI	4	294	5586	4452	96%
JGT	17	264	5775	188	72%
JOT	7	232	13547	4000	81%
GUA	15	1175	>800K	>200K	77%

Ultimately, we chose nine object programs, as follows.[1] *Commons CLI* (CLI) provides an API for parsing command line options. *Commons Codec* (CDC) implements common encoders and decoders such as Base64. *Commons Collections* (COL) is a collection of data structures. *Commons Logging* (LOG) establishes communication between logging systems. *Commons Math* (MTH) provides math and statistics tools for numerical analysis. *Commons Primitives* (PRI) provides utilities for manipulating primitive data types. *JGraphT* (JGT) provides graph-theory objects and algorithms for graph analysis. *Joda Time* (JOT) provides new functionalities for Java time classes. *Guava* (GUA) (formerly Google Collections) is a set of collection types.

Table 1 provides basic data on these object programs, including the numbers of packages, classes, and lines in the code bases for the objects, the numbers of test cases that we utilize, and the branch coverage of the object programs' code achieved by those test cases. Statistics were gathered using the Cobertura tool [2] [1].

3.2 Variables and Measures

3.2.1 Independent Variables

We investigate the impact of three independent variables.

Our first independent variable involves oracle selection techniques. We explore the relative merits of three techniques: Dodona, outlined in Section 2; MAODS, the previous state-of-the-art approach based on mutation testing [18]; and manual oracle specification. MAODS serves as our baseline, because to the best of our knowledge, there are no other automated approaches for specifying oracle data sets. For the purpose of this study, we reimplemented MAODS for use with Java programs, using the MAJOR mutation system for Java programs [12]. To use MAODS in Java, this approach now employs the observability mapping used by Dodona, but otherwise is the same as before [18]. In contrast, manually constructed oracles, being built by developers with a deep understanding of the source code, serves as a representation of the current state of practice.

Our second independent variable is the centrality metric used for Dodona. We explore how this metric impacts the effectiveness of Dodona, using the four centrality metrics outlined in Section 2.3.1.

Our third independent variable is the oracle data set size; we vary data set size to understand how the relative effectiveness of the techniques and the centrality metric vary depending on the size of the test oracle.

3.2.2 Dependent Variable

To investigate our research questions, we measure the fault detection effectiveness and the cost of oracle data selection approaches. Let T be a set of test inputs, and let O be an oracle data set for T, created by oracle data selection technique M. To measure the fault detection effectiveness of technique M on object program P, for T and O, we compute the percentage of faults in P that can be detected by T augmented with O. (The faults utilized in our study are mutation faults, and are described further in the following section).

To measure the cost of M on object program P, for T and S, we compute the runtime (as wall clock time) for the entire oracle data selection process. For Dodona this includes running T over P to generate the adjacency matrix, computation of network centrality, mapping data sets to observable points and computation of the ranking. For MAODS this includes running T against all the mutants and computing the ranking.

3.3 Controlled Factors

3.3.1 Test Inputs

[1] We omitted the NanoXML system used by Fraser et al., due to problems encountered applying our prototype to it. These problems are strictly implementation related, and could be surmounted through an improved prototype.

[2] Accurate branch coverage statistics for LOG could not be produced due to an incompatibility with this system and Cobertura

In prior work, we used random test inputs to evaluate the effectiveness of oracle data set specification approaches [18]. This was necessitated by the closed source nature of the projects studied, which prevented us from using the test suites actually developed for the systems. In this work, we wished to compare DODONA not only against MAODS (*RQ2*), but also against test oracles developed by actual testers (*RQ3*). Such test oracles, using oracle data sets carefully selected by the test developers, represent a challenging target for our approach and a good baseline for comparison.

To do this, we needed test suites containing manually constructed test oracles. Constructing such a test suite ourselves for each object would be prohibitively costly for an initial study, but fortunately each of our object programs is part of a mature, open source project, and thus has an associated set of test cases constructed by the developer. We thus used each test suite — with the developer's assertions removed — as the set of test inputs when evaluating both MAODS and DODONA.

Our goal is to support testers via automated oracle data set specification; we expect that the actual choices of expected data values will be manual. In this work, however, we are interested in first developing our approach, with work evaluating the human factor to come later. To allow for evaluation without a user study, we specify expected values for each proposed oracle data set by executing the test suite over the original, unmutated Java program, filling in expected values using the results.

3.3.2 Faults

To measure the fault detection effectiveness of oracle data selection approaches, we embedded mutation faults into our object programs. This process proceeded in two steps. First, we used MAJOR, the mutant generation tool on which MAODS is based, to generate single fault mutants for each object program [12]. The faults seeded by MAJOR model fault classes found in object-oriented programs, and are similar to those used in our previous work [18].

MAJOR generates as many as possible for the operators specified, and for our objects at least 400 mutants were generated for each system, with a larger number being generated for larger programs. These mutants were then divided into an evaluation set (roughly half) which was used for computing all fault finding numbers, and a training set used with MAODS. The evaluation set was then subdivided into subsets of roughly equal size, resulting in 10 or more evaluation mutant sets for each case example, each of at least size 40.

Note that using the same tool for both MAODS and our evaluation represents a risk. MAODS may appear more effective during evaluation than it would be in practice, because the mutants used to select the oracle data are similar to those used in the evaluation.

3.4 Experiment Process

We performed the following process for each object.

1. Remove the test oracles from the original developers test suite.
2. Generate the mutant sets for evaluation.
3. Generate the DODONA-enhanced test suite using the oracle-free test suite, recording the time required.
4. Generate the MAODS-enhanced test suite using the oracle-free test suite, recording the time required.
5. Run both enhanced test suites over the original program, and use the results to fill in the expected values for their respective oracle data sets.
6. Execute each test suite against each mutant set, computing the number of mutants killed.

The foregoing process results in at least 50 fault-detection effectiveness measurements per technique.

3.5 Threats to Validity

External: Our study is limited to nine mid-sized Java libraries. Nevertheless, these objects are common targets in automatic test case generation work, and given that our goal is to help testers use automatic test case generation tools, are representative for our purposes.

We have used manually constructed test suites in our study to allow us to compare our results to manually constructed test oracles. Other methods of generating test suites are possible, notably, approaches using automatically generated test suites. However, the test suites used in this study are comparable in terms of coverage and fault finding to automatically generated tests.

We have generated at least 40 mutants for each mutant set evaluated. This value was chosen to yield a reasonable study run-time, and it is possible that larger sets may yield different results. However, in our experience, larger sets of mutants typically result in similar levels of fault finding.

Internal: It is possible that our implementations of MAODS and DODONA, or the automation used in our experiment, contain faults. The tools underlying our competing approaches (JPF and MAJOR), however, are well tested, and we have extensive experience using both [12, 24].

Construct: We have measured fault detection effectiveness based on seeded faults introduced via mutation analysis. Nevertheless, empirical studies have suggested that for the purpose of testing experimentation, results with mutation faults are comparable to actual faults [4].

When measuring fault detection, we have assumed a "perfect" tester; that is, we have assumed that the tester always specifies the correct value for a proposed set of oracle data. In practice this is unlikely to be true [8]. However, this is a problem affecting all approaches to testing, even the "fully automatic" approaches. In this work we wish only to evaluate whether our approach can quickly find an effective oracle data set. Once we have established whether our approach is technically sound, user studies will be required to determine the effectiveness of the approach in-vitro.

4. RESULTS AND DISCUSSION

In this section, we discuss the conclusions and implications of our results in the context of our four research questions. We begin by visualizing our results (we use the abbreviated program names given in Section 3.1).

In Figure 2 we plot the median fault detection effectiveness for each network centrality metric used (*RQ1*). Note that we highlight the apparent "best" centrality metric, eigenvalue, with a dotted blue line (this is discussed in Section 4.2). This designation of best is assumed in subsequent figures; for these figures, DODONA refers to the approach given in Section 2 used with eigenvalue centrality.

In Figure 3 we plot the median fault detection effectiveness of test suites using oracle data set generated by DODONA, MAODS, and the manually constructed test oracles associated with the original system. We visually represent the statistical analysis (presented below) comparing Dodona and MAODS on the line for Dodona as follows: green rectangles indicate that Dodona outperforms MAODS with statistical significance ($\alpha = 0.05$), yellow triangles indicate that there is no statistically significant difference between the techniques, and red xs indicate that Dodona is outperformed by MAODS with statistical significance.

In Figure 3, we also plot the fault detection effectiveness of the original, manual test oracles as a horizontal red dashed line. For this line, the x-axis does not represent oracle size – in general, computing the size of manually constructed oracle data sets is not feasible. While we control for the size of automatically generated oracle data sets, we naturally cannot for the size of developer constructed test oracles and for a given test suite, the size of the manually constructed expected value test oracles varies. Thus, rather than present a potentially misleading oracle size, we plot the fault detection for test oracles as a horizontal line.

Finally, in Figure 4 we plot the average wallclock runtime required to compute the oracle data sets using Dodona and MAODS.

4.1 Statistical Analysis

As shown in Figures 3 and 4, Dodona appears to be both more effective and more efficient that MAODS in most scenarios. However, in the case of fault detection effectiveness, there is a fair amount of overlap between the approaches. We therefore wished to determine (with statistical significance) at which oracle sizes and for which objects Dodona outperforms MAODS in terms of fault detection. We begin by restating our research questions as statistical hypotheses[3][4]:

H_1. For a given system S and oracle size m, Dodona outperforms MAODS.

H_2. For a given system S, Dodona requires less time to generate an oracle data set than MAODS.

We have a large number of observations (30+), and thus the t-test is appropriate (even in the absence of normality). We apply this test at for each case example and oracle size for H_1, and for each case example for H_2. This produces a large number of p-values in the case of H_1. Rather than report p-values, we visually indicate the statistical significance of each comparison at the level of $\alpha = 0.05$ in Figure 3 as described above.

In considering H_2, we can reject each null hypothesis in each case and thus accept H_2. We therefore conclude that for each object, Dodona is more efficient than MAODS with statistical significance at $\alpha = 0.05$.

Note that no statistical hypothesis testing is presented for *RQ1* or *RQ3*. In the former case, this is due to the very large number of observations – one per combination of oracle size and metric, resulting in an unwieldy amount of data. In the latter case, there exists only a single manually constructed test oracle for each object, and thus we have insufficient data on which to perform statistical hypothesis testing.

4.2 RQ1: Impact of Centrality Metric

Our first task is to select a single centrality metric for use with Dodona or, failing that, to develop a set of guidelines for when each metric should be used. From Figure 2, we can see that the clear winner here is the eigenvalue centrality metric. While other centrality metrics outperform eigenvalue centrality for some oracle sizes (typically oracles of size one or two) on most systems, the differences in fault detection effectiveness are usually slight – under 5%.

In contrast, on all but one of the nine systems (*LOG*), eigenvalue becomes the most effective metric for oracles of size four or greater, often by a wide margin. For example, on *CDC*, eigenvalue centrality outperforms the next best metric (degree centrality) by 32%, for an increase in fault detection effectiveness of 74%. Even on *LOG*, eigenvalue centrality overtakes the next best metric (betweenness centrality) for oracles of size seven and eight.

Initially, we had expected that the betweenness metric would be the best fit for oracle data selection. Intuitively, betweenness in our context captures the likelihood that a variable is frequently in the path of data propagating through the system. We expected that such variables would be good candidates for a test oracle, being somewhat in the middle of computations, a point which balances the likelihood of errors being masked with the need for enough computation to have occurred to make detecting a fault likely. In contrast, we believed eigenvalue centrality—while also a reasonable choice—would often select many highly related variables, defeating the purpose of selecting an oracle data *set*.

We hypothesize that, in practice, the variables selected by betweenness are difficult to actually observe. These variables, being in the middle of computations, may be too far from a public method return to be observed with accuracy, and are thus either filtered out of the ranking or observed very indirectly (e.g., not using accessor methods). In contrast, variables selected by eigenvalue centrality, being connected to other highly connected variables, typically are found near the end of long computations. Consequently, these variables are usually easy to observe, and the mapping process is more accurate. Furthermore, in practice, several fairly unrelated variables are selected by this process, as evidenced by rapid increases in fault finding for small oracles.

In summary, on the object programs that we consider, the eigenvalue centrality metric is clearly the best choice for use with Dodona. Consequently, in the remainder of this discussion we focus on the use of this metric.

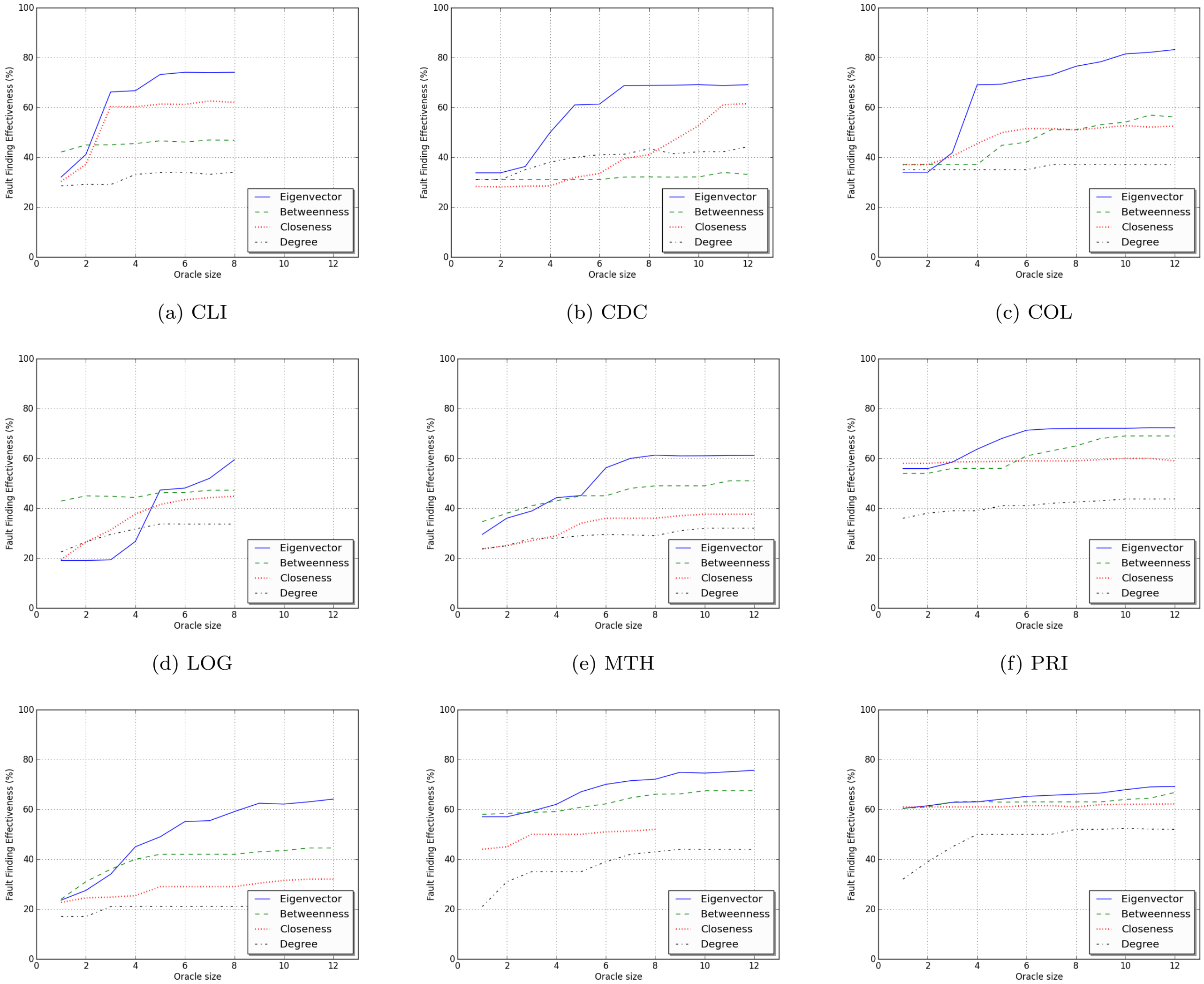
4.3 RQ2: Effectiveness Relative to MAODS

As noted in Section 1, our goal in developing Dodona was to improve on the efficiency of automated approaches selecting oracle data sets. Nevertheless, we still wish to produce effective oracle data sets, and therefore we seek to determine how Dodona compares, in terms of fault-detection effectiveness, with MAODS.

As Figure 3 shows, Dodona typically produces oracle data sets that are at least as effective as those produced by MAODS, albeit with some variation between case examples. On four of the nine object programs — *CLI*, *CDC*, *JGT*, and *JOT* — Dodona is clearly more effective. In these cases, Dodona outperforms MAODS with statistical

[3] Null hypotheses are omitted for space reasons.

[4] Note that we do not generalize across objects as the appropriate statistical assumption—random selection from the population of Java programs—is not met. Furthermore we do no generalize across oracle sizes as our approach's effectiveness may vary depending on size. The tests are used to determine if observed differences are likely due to chance.



(a) CLI

(b) CDC

(c) COL

(d) LOG

(e) MTH

(f) PRI

(g) JGT

(h) JOT

(i) GUA

Figure 2: Median effectiveness of each network centrality metric when used with Dodona.

significance for nearly all oracle sizes, with improvements of up to 115% (for *CDC*when using an oracle size of six), and for no oracle sizes does MAODS outperform Dodona.

Additionally, on three of the nine object programs — *PRI*, *MTH*, *GUA* — Dodona and MAODS produce comparable levels of fault detection effectiveness across oracle sizes. For *PRI* we see that Dodona outperforms MAODS by up to 16% (for oracle sizes larger than 4). For *MTH*, oracle data sets produced by Dodona typically achieve higher detection effectiveness with statistical significance, but an exception exists for oracles of size 12. For *GUA*, Dodona is more effective than MAODS for lower oracle sizes.

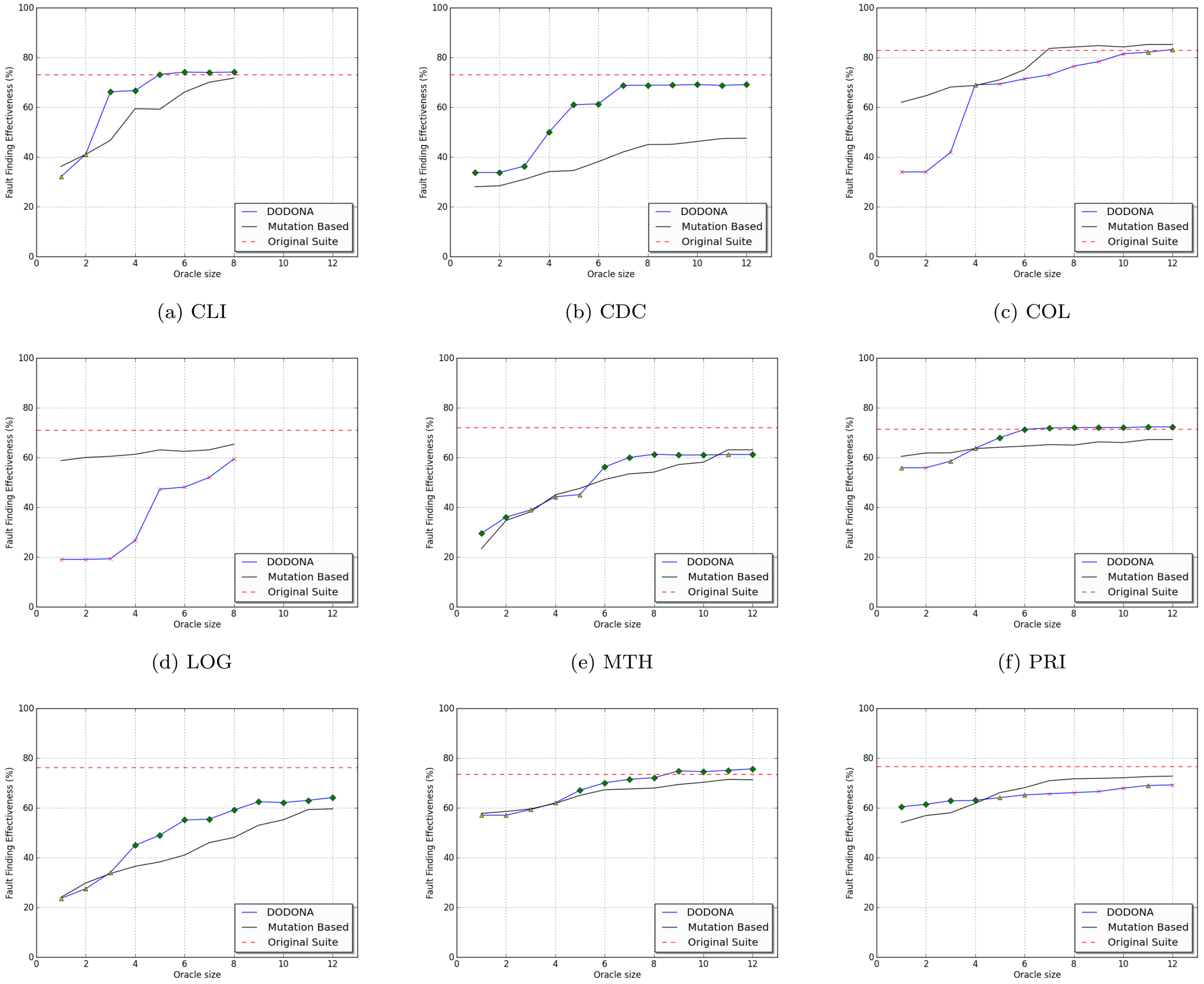
Only for two object programs, *LOG* and *COL*, does MAODS consistently outperform Dodona. This is particularly pronounced for *LOG*, where Dodona finds less than 50% of the faults for oracle sizes of six or less, while MAODS achieves no less than 58% fault detection effectiveness for any oracle size. Only at oracles of size eight are the fault detection effectiveness results for both approaches comparable.

Overall, Dodona appears to be a better choice than MAODS when selecting oracle data, despite some variation across object programs. While in some cases Dodona can result in the selection of a less effective oracle data set, these cases are in the minority. In fact, in many cases Dodona produces more effective oracle data sets than MAODS— sometimes much more, as in the case of *CDC* and *CLI*. This is despite the fact that, per Section 3.3.2, the implementation and evaluation of MAODS both use MAJOR and thus our evaluation somewhat favors MAODS.

4.4 RQ3: Effectiveness Relative to Manually Constructed Oracles

Software developers, being familiar with their systems, are well suited to construct test inputs and oracles, and in practice they must routinely do so. Thus while our goal is create an effective automated approach for reducing effort—not surpassing human intelligence—developers' test oracles (and by implication, the selected oracle data sets) are an interesting datapoint to consider when assessing effectiveness.

We expected that in practice both Dodona and MAODS



(a) CLI

(b) CDC

(c) COL

(d) LOG

(e) MTH

(f) PRI

(g) JGT

(h) JOT

(i) GUA

Figure 3: Median effectiveness of each oracle data selection approach.

would be less effective than manually constructed test oracles. However, as shown in Figure 3, we found that Dodona and manually constructed oracles were often comparable.

For four systems — *CLI*, *CDC*, *JGT*, and *PRI* — Dodona provides fault detection effectiveness within 5% of that of manually constructed test oracles for oracles of moderate size (size four or larger). For several other systems, Dodona provides reasonable effectiveness – within 20% of that of manually constructed test oracles. Only for *LOG* does Dodona provide fault detection effectiveness considerably worse that that of manually constructed test oracles (42%+ less for oracles of size five or less), and as noted above, this system represents the outlier in terms of effectiveness for Dodona.

We find these results to be very encouraging. We expect oracle data manually selected by developers to be very effective; our goal is find reasonably effective oracle data with a level of automation that is capable of reducing programmer effort. The fact that Dodona can select oracle data not only better than or comparable to that selected by MAODS, but also often comparable to the data selected by developers themselves demonstrates the promise of the approach.

4.5 RQ4: Efficiency Comparisons

While the results for *RQ2* demonstrate that Dodona is relatively effective in terms of fault detection effectiveness, the original motivation behind this work was to correct perceived technical shortcomings in MAODS; notably, the reliance on potentially expensive mutation testing to select oracle data. Thus, one of our primary concerns is the relative efficiency of Dodona relative to MAODS.

As shown in Figure 4, Dodona required less time to generate oracle data sets than MAODS, with decreases in the time required ranging from of 17.3%-89.8%. (Per Section 4.1, all differences were statistically significant at $\alpha = 0.05$). We thus conclude that Dodona does indeed reduce the time required to produce oracle data sets.

While our results concerning efficiency were positive, we were surprised at how competitive MAODS was relative to Dodona. On paper, Dodona should clearly be the faster approach. Instead of running a test suite multiple times,

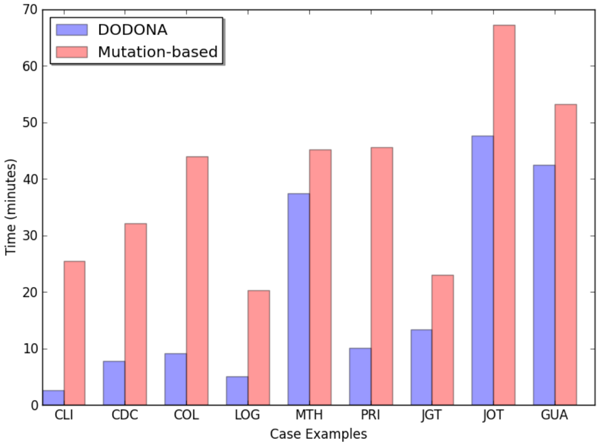


Figure 4: Execution time for each approach.

once for each generated mutant (for MAODS), Dodona runs each test suite once, tracks the flow of data during execution, and applies a network centrality metric. While there is some overhead for dataflow analysis, after which we must compute the network centrality (always a runtime linear to the number of vertices), we expected Dodona to be at least twice as fast as MAODS for all objects. Instead, for four of the nine study objects, MAODS required only 15.3%-42.5% more time to compute an oracle data set.

We also expected that for larger Java programs, the improvement in speed achieved by Dodona would be more pronounced. In practice, however, we observed no relationship between the number of statements in object programs, and the time required to generate oracle data sets for those programs. From this we infer that the relative scalability of the approaches is not a simple function of program size.

Concerning the relative speed of the approaches, we note that the dataflow computations done by Dodona rely on Java Pathfinder (JPF). JPF was selected because it is easily extensible, but this comes with a cost: it is a custom-built research JVM, making it a heavyweight tool for tracking dataflow relationships. In contrast, MAODS uses the standard (highly optimized) JVM for execution, and the mutation analysis tool MAJOR is a product of an extensive body of research on mutation testing. Thus, while our implementation of Dodona could likely easily be made more efficient by using more lightweight, dataflow-specific tools based on a standard optimized JVM, improving the speed of our MAODS implementation would be more challenging.

To better understand the limitations on Dodona's scalability, we analyzed the runtime for Dodona for each system. For most systems, we found that the computation of the eigenvalue network centrality metric was very fast – less than one minute. However, for our problem systems — *MTH*, *JGT*, *JOT*, *GUA* — we found that runtimes were higher, ranging from 2.2 to 4.3 minutes. While this is a small percentage of the overall runtime, the runtime for eigenvalue centrality is linear in the number of nodes (variables). Thus we can infer that the increase in cost is linked to capturing large dataflow networks: as the number of intermediate computations to be tracked grows, the workload for Dodona alone increases. This suggests that future versions of Dodona could be made more efficient by preemptively dropping uninteresting/useless aspects of the dataflow network, or again by simple performance increases in dataflow tracking by using a more lightweight dataflow engine.

5. RELATED WORK

While significant work on automatic test generation exists, active work specific to test oracles is a recent phenomenon [22]. For example, several authors have recently discussed the need to focus on test oracles when evaluating the quality of the testing process [22, 3], and Harman et al. have recently conducted a comprehensive survey of test oracle research [11].

However, there still exist little work specific to constructing, or supporting the construction, of test oracles. Xie and Memon explore methods of constructing test oracles specifically for GUI systems, yielding several recommendations [27, 13]. Several tools exist for automatically generating invariant-based test oracles for use in regression testing, including Eclat [15] and DiffGen [23], though such work assumes the program is currently correct.

Work on generating oracles for non-regression testing also exists. Several authors have proposed methods of inferring invariants from programs for use in testing [26, 7]. Fraser et al. [9] propose $\mu TEST$, which generates complete JUnit test cases for object oriented programs. Both bodies of work assume the tester will later manually correct generated test oracles, and are part of the "generate-and-fix" paradigm for test oracle construction. Work evaluating this paradigm with users is mixed, but on the whole discouraging [8, 19].

In contrast, we are trying to support creation of a test oracle, rather than completely automate it. Towards this, Staats et al. proposed a mutation-based approach for selecting oracle data based on how often a variable reveals a fault in a mutant [18]. This work's limitations are scalability and the need to estimate the number of required mutants to select effective oracle data. Pastore et al. [17] have proposed CrowdOracles, an approach to use crowdsourcing for checking assertions. The main limitation here is the need for qualified crowd to produce the test oracle.

To the best of our knowledge, this work is the first to leverage network centrality metrics to produce oracle data sets. Work by Zimmerman et al. leveraging network centrality metrics in the context of software engineering work does exist, though the context is very different (defect prediction) [28]. Voas and Miller also note that errors typically propagate through a system, but provide no method of selecting oracle data based on this observation [25].

6. CONCLUSION

Test oracles, like test inputs, are a key aspect in achieving effective test results, but research on oracle generation is relatively scarce. In this work we have presented an approach for automatically specifying oracle data sets, with the goal of helping harness engineers' understanding of systems to create effective oracles. Our system, Dodona, in most case outperforms the state-of-the-art MAODS system in terms of effectiveness and efficiency, resulting in improvement in fault finding of up to 115% and reduction in generation time by up to 89.8%. Furthermore, Dodona performs surprisingly well in comparison to oracles created fully manually by system developers, resulting in very similar fault finding for four of the nine objects studied.

7. REFERENCES

[1] Cobertura framework. Available at `http://cobertura.github.io/cobertura/`.

[2] Jung framework. Available at `http://jung.sourceforge.net/`.

[3] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, et al. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 2013.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, pages 402–411, 2005.

[5] L. Baresi and M. Young. Test oracles. *Technical Report CISTR-01*, 2, 2001.

[6] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14, New York, NY, USA, 2011. ACM.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[8] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 188–198. ACM, 2013.

[9] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[10] L. C. Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1979.

[11] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical report, Technical Report Research Memoranda CS-13-01, Department of Computer Science, University of Sheffield, 2013.

[12] R. Just, F. Schweiggert, and G. M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 612–615. IEEE Computer Society, 2011.

[13] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should i use for effective gui testing? In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003.*, pages 164–173. IEEE, 2003.

[14] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. *SIGSOFT Software Engineering Notes*, 25(6):30–39, Nov. 2000.

[15] C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. *ECOOP 2005-Object-Oriented Programming*, pages 504–527, 2005.

[16] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 199–209, 2011.

[17] F. Pastore, L. Mariani, and G. Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *2013 IEEE Sixth International Conference on Software Testing*, 2013.

[18] M. Staats, G. Gay, and M. P. Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the International Conference on Software Engineering*, pages 870–880, 2012.

[19] M. Staats, S. Hong, M. Kim, and G. Rothermel. Understanding user understanding: determining correctness of generated program invariants. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 188–198. ACM, 2012.

[20] M. Staats, P. Loyola, and G. Rothermel. Oracle-centric test case prioritization. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 311–320, 2012.

[21] M. Staats, M. W. Whalen, and M. P. Heimdahl. Better testing through oracle selection. In *Proceedings of the International Conference on Software Engineering (NIER Track)*, pages 892–895, 2011.

[22] M. Staats, M. W. Whalen, and M. P. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 391–400. IEEE, 2011.

[23] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. ASE 2008.*, pages 407–410. IEEE, 2008.

[24] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[25] J. M. Voas. Pie: A dynamic failure-based technique. *IEEE Trans. Software Eng.*, 18(8):717–727, 1992.

[26] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 191–200, 2011.

[27] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1):4, 2007.

[28] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM.