2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

CrowdOracles: Can the Crowd Solve the Oracle Problem?

Fabrizio Pastore and Leonardo Mariani
University of Milano Bicocca
Milan, Italy
{*pastore,mariani*}*@disco.unimib.it*

Gordon Fraser
University of Sheffield
Sheffield, UK
Gordon.Fraser@sheffield.ac.uk

***Abstract*—Despite the recent advances in test generation, fully automatic software testing remains a dream: Ultimately, any generated test input depends on a *test oracle* that determines correctness, and, except for generic properties such as "the program shall not crash", such oracles require human input in one form or another. CrowdSourcing is a recently popular technique to automate computations that cannot be performed by machines, but only by humans. A problem is split into small chunks, that are then solved by a crowd of users on the Internet. In this paper we investigate whether it is possible to exploit CrowdSourcing to solve the oracle problem: We produce tasks asking users to evaluate CrowdOracles – assertions that reflect the current behavior of the program. If the crowd determines that an assertion does not match the behavior described in the code documentation, then a bug has been found. Our experiments demonstrate that CrowdOracles are a viable solution to automate the oracle problem, yet taming the crowd to get useful results is a difficult task.**

***Keywords*-test oracles, crowd sourcing, test case generation**

I. Introduction

Automatic test generation is an important and intensively investigated research area, in which many solutions have been proposed. For example, PEX [1] or DART [2] use dynamic symbolic execution, EvoSuite [3] uses evolutionary search, and Randoop [4] uses random sampling to generate test cases. Many of these techniques generate test cases that stimulate the software under test without explicitly checking the correctness of the results returned by the tested program. Unless there is a formal specification, these techniques can only reveal faults that cause general problems that can be recognized without requiring any knowledge about the system under test, such as crashes or uncaught exceptions. The lack of a means of verifying the correctness of an execution beyond the presence of general problems is known as the *oracle problem* [5]. This problem is extremely relevant – solving it would enable full automation of software testing.

Automatic synthesis of test assertions is an initial step towards automatic generation of oracles [3], [4]. Unfortunately, the synthesized assertions are not oracles because they encode the behavior observed by executing the test case, rather than the *intended* behavior. For instance, if a calculator component implements a faulty `sum` operation that always returns 0, a test case that executes `sum` with inputs 5 and 3 will include the (incorrect) assertion

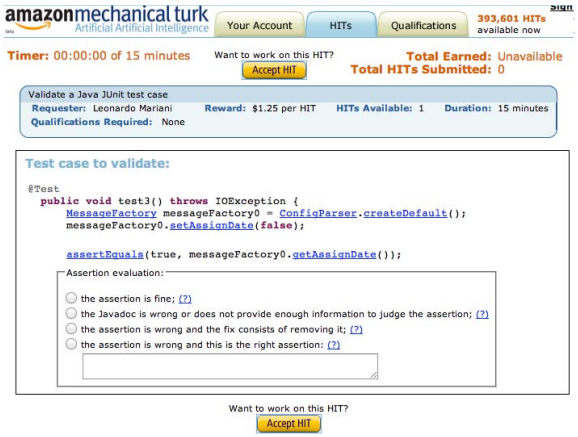

Figure 1. An example CrowdOracle task, demonstrating how the oracle problem can be represented as a Human Intelligence Task (HIT) on commercial CrowdSourcing platforms: Crowd workers decide on the correctness of the presented assertions based on the code documentation.

`assertEquals(0, sum(5,3))` rather than the (correct) assertion `assertEquals(8, sum(5,3))`. Even if the automatically synthesized assertions cannot be considered oracles, these assertions represent a relevant contribution towards the generation of oracles. In fact, the generation of these assertions implies the identification of both the relevant points where the assertions should be placed and the right variables that should be checked in the assertion; and state of the art tools do a good job in both cases [3].

To turn synthesized assertions into oracles it is necessary to identify and fix the incorrect assertions. Fixing assertions provides two important benefits: (1) each fixed assertion usually corresponds to an observed behavior that differs from the intended behavior, and thus it leads to the immediate identification of a fault; and (2) the test suite with the fixed assertions can be effectively and efficiently used for regression testing because every violation produced in the future by the assertions will be relevant for developers.

Unfortunately, verifying synthesized assertions can hardly be automated because it requires human intelligence – correcting an assertion requires understanding what the system is supposed to do. Humans can usually fix assertions relatively easily because unit tests tend to be short and simple. In practice, however, developers do not revise synthesized assertions because the total effort necessary to revise an

978-0-7695-4968-2/13 $26.00 © 2013 IEEE
DOI 10.1109/ICST.2013.13

entire test suite with many tests and assertions is significant.

Recently, the idea of CrowdSourcing software engineering problems [6] has gained popularity. CrowdSourcing a problem consists of specifying it in the form of a Human Intelligence Task (HIT) and making the problem available in a CrowdSourcing platform, where registered workers can choose to complete HITs for a small remuneration. CrowdSourcing offers a unique opportunity for the oracle problem: In this paper, we introduce the idea of *CrowdOracles*, where test cases with synthesized assertions are verified with respect to the documentation and fixed by the crowd, thus offering a solution to the oracle problem that, from the viewpoint of the developer, is fully automated. In detail, the contributions of this paper are:

- We describe the idea of using CrowdSourcing to overcome the oracle problem.
- We empirically investigate the feasibility of CrowdOracles by studying different models of interactions with the crowd and different ways of collecting and combining the results produced by the crowd.
- We empirically investigate how well the crowd behaves, not only in identifying the wrong assertions, but also in providing the correct fixes.

We do not systematically look into the aspects that are affected by rapidly changing factors, such as the economic convenience of CrowdOracles compared to other solutions, which is influenced by the relative incomes of countries, and the time that the crowd requires to complete the tasks, which is influenced by the number of workers subscribed to a platform. However, we provide the empirical data that reflect our experience with the platform that we used, to give some hints about the current state of the practice.

The main findings of our study are that the crowd can perform well at verifying CrowdOracles, but the performance strongly depends on the qualification of the crowd and the clarity and complexity of the underlying tests and documentation. Interestingly, our experiments also showed that once the crowd has identified a wrong assertion, it can also fix it. The results of our study can be used to setup a process for CrowdSourcing the oracle problem.

This paper is organized as follows: Sections II describes CrowdSourcing and how we CrowdSourced the oracle problem. Section III elaborates the research questions that are investigated in this paper. Section IV describes the empirical process that we followed to answer the research questions, and Section V reports and discusses the empirical results. Finally, Section VI discusses related work, and Section VII provides final remarks.

II. CrowdSourcing Software Engineering Tasks

CrowdSourcing a task can be seen as a problem solving strategy [7]. To solve a problem, a requester specifies a Human Intelligence Task (HIT), which encodes the problem, and assigns it to an undefined (and possibly large) group of workers (the crowd) who provide their solutions back to the requester; the requester can then derive the final solution from the solutions collected from the workers. The workers are usually rewarded for their work with cash or prizes. A CrowdSourcing platform mediates the interaction between the requesters and the workers. Even if the identity of the workers is unknown, the platforms usually provide a certain level of control over the profile of the workers who can accept a HIT.

Amazon Mechanical Turk

There exist a number of platforms that support CrowdSourcing, and each platform has its own crowd and its own set of features. Well known platforms are Amazon Mechanical Turk[1], CrowdFl$\hat{o}$wer[2], Gun.io [3],and Topcoder[4]. The HITs that encode the oracle problem are quite simple and most of the platforms are suitable for our purposes. We selected the Amazon Mechanical Turk (AMT) platform for our experiments because it is well known, well documented and it implements a web service API that lets us automate part of the work.

The typical way of interacting with AMT is the following: The requestor accesses AMT and creates a new HIT by providing a title and a description. The description is usually provided in HTML and includes fields to let workers enter the results. When entering a HIT into the platform, the requestor can specify the number of assignments associated with the HIT, that is the number of (different) workers who will have to complete the same task. Finally, the requestor specifies the monetary reward that is provided to a worker that completes an assignment.

A HIT can be associated with a qualification that must be owned by a worker to be able to accept it. A requestor can define new qualifications and can define the test that a worker has to complete to obtain the qualification. Usually the test is another HIT that is available in the platform and that a worker has to successfully complete in order to get the qualification.

Workers can fully view HITs before accepting them, but can provide an answer only upon accepting the assignment. As soon as a worker submits a result for an assignment, the result is available to the requestor. The requestor can decide whether to accept or reject a result submitted by a worker. The worker gets the monetary reward associated with the HIT only if the requestor accepts the result.

CrowdOracles

In our study each HIT requires checking and fixing the assertions included in a test case. We implemented an Eclipse plug-in that automatically creates a HIT, in HTML

[1]https://www.mturk.com/mturk/welcome
[2]http://crowdflower.com
[3]http://gun.io
[4]http://www.topcoder.com

format, for every JUnit test case in an Eclipse project (or for a selection of test cases). Every HIT has an initial description that explains what the worker is supposed to do[5]. After the description, the HIT includes the code of the test case, and next to each assertion there is a form with the following four options (see Figure 1):

1) the assertion is fine;
2) the Javadoc does not provide enough information to judge the assertion;
3) the assertion is wrong and the fix consists of removing it;
4) the assertion is wrong and this is the right assertion:

The worker has to select the proper option. If option 4 is selected, the worker has to specify the correct assertion in a textfield.

A test case may include a `fail` statement rather than an `assert` statement, which typically occurs in a try-catch block to signify an expected exception. In these cases, a HIT includes slightly different options.

1) the fail assertion is fine (the method before fail should throw the expected exception);
2) the Javadoc does not provide enough information to judge the fail assertion;
3) the fail assertion is wrong, an exception should not be thrown;
4) the fail assertion is wrong, an exception of the following type should be thrown:

If option 4 is selected, the worker has to specify the correct exception that should be caught. In the rest of the paper we will use the term assertion to generally refer to both, the case of an assert statement and a fail statement.

Each method name or class name that occurs in the HIT can be clicked to open a pop-up window. If a method name is clicked, the pop-up window includes the method signature and the description of the method extracted from the Javadoc code documentation. If a class name is clicked, the pop-up window includes the class description extracted from the Javadoc. This information is useful to help the worker understanding and analyzing the test case.

In our study we also considered HITs that only workers with a qualification can accept. To this end, we introduced the CrowdOracle qualification and a qualification test to obtain it. The qualification test consists of revising two assertions and a fail statement included in three test cases. One assertion is wrong and requires to be fixed, while the other is correct. The exception caught in the test case with the fail statement is also wrong and should be changed. When evaluating the result of the qualification test submitted by a worker, we assign 20 points for every correct answer, 20 points if the wrong assertion is fixed in the right way, and 20 points if the exception is fixed in the right way.

[5]For space reason we do not report the description here, but the tool, the objects of the study, and the results are available at http://www.lta.disco.unimib.it/tools/crowd/

Listing 1. Qualification Test

```
1  public void test1 () {
2    ArrayList list0 = new ArrayList()
3    assertEquals( false, list0.isEmpty() );
4  }
5
6  public void test2 () {
7    ArrayList list0 = new ArrayList()
8    Integer int0 = new Integer(3);
9    list0.add(int0);
10   assertEquals( 1, list0.size() );
11 }
12
13 public void test3 () {
14   ArrayList list0 = new ArrayList()
15   Integer int0 = new Integer(3);
16   list0.add(int0);
17   Integer int1 = new Integer(-2);
18
19   try {
20     list0.add(3,int1);
21     fail("Expecting exception:
22         IllegalArgumentException");
23
24   }catch(IllegalArgumentException e) {
25   }
26 }
```

A worker gains the qualification if at least 60 out of 100 possible points are obtained. Intuitively 60 points correspond to the capability of recognizing what is wrong and what is right, even if the worker is not able to provide any fix. The objective of the qualification test is to select the workers that have a reasonable confidence with JUnit tests and assertions, and prevent that unskilled workers complete our assignments. Listing 1 shows the code in our qualification test (to save space we removed the four options following each assertion).

III. Research Questions

In our empirical investigation we consider the following three key research questions:

- RQ1: Is it possible to use an unqualified crowd to fix automatically generated assertions?
- RQ2: What is the quality of the results returned by a qualified crowd?
- RQ3: Is it possible to distill correct CrowdOracles from the results returned by the crowd?

Research question RQ1 investigates if crowd platforms can be used without worrying about the profile and expertise of the workers. The question is whether the workers in a crowd behave fairly enough to exclude themselves from accomplishing tasks that are out of their expertise, or on the contrary workers tend to accept tasks regardless of their real capability of correctly completing the tasks.

Research question RQ2 investigates what is the quality of the results returned by the workers in the crowd. If the crowd generates a high number of wrong results, fixing automatically generated assertions through the crowd would be infeasible. On the contrary, a well behaving crowd could be the basis for generating CrowdOracles. This research question looks at the results returned by the crowd from

two perspectives. First, we investigate if the crowd is able to distinguish wrong from correct assertions. Second, we investigate if the crowd is able to fix the assertions that are recognized to be wrong.

Research question RQ3 investigates the effectiveness of the possible processes for taking decisions about the correctness of synthesized assertions, starting from the results returned by the crowd. This research question provides insights about the feasibility of the CrowdOracle approach.

IV. Empirical Setup

In this section we describe the objects of the study and the empirical process that we followed to investigate research questions RQ1-3.

A. Objects of the Study

To study the capability of the crowd at fixing assertions generated by tools, we need both a test case generation technique that produces test cases that include assertions and applications that include faults. We need applications with faults otherwise the generated oracles would be always correct, and the crowd should only confirm the assertions in the test cases, and we would be unable to measure the capability of the crowd in fixing wrong assertions.

We used EvoSuite [3] for generating test cases because the tool targets Java, a well known language that will likely be known by a number of workers, and represents the state of the art in assertion generation.

As applications with faults we considered two sets of cases: simple cases and hard cases. Simple cases correspond to the case of assertions embedded into test cases designed for classes with a clear and well documented interface. The hard cases correspond to the case of assertions embedded into test cases designed from classes with a complex and relatively well documented interface.

We generated the simple cases by injecting faults in the `java.util.Stack` class, because it has a clear and well documented API. Test cases to be submitted to AMT were produced as follows: First, we used MuJava [8] and its set of sufficient mutation operators defined in [9] to inject faults in the `Stack` class; this resulted in 59 mutants. Then, we used EvoSuite to generate, for each mutant, a full test suite targeting branch coverage. The test cases that cover the injected faults lead to wrong oracles, which reflect the incorrect behavior that has been revealed. For instance, one of the test cases automatically produced by EvoSuite is shown in Listing 2. The test case creates an empty Stack, checks if it is empty and retrieves the object at the top of the stack causing an exception of type `EmptyStackException`. The presence of a fault in the implementation of class `Stack` leads EvoSuite to generate an erroneous assertion in line 3: the assertion expects that the invocation of method `empty()` on the empty stack returns `false` instead of `true`.

Listing 2. Example test generated by EvoSuite for a `Stack` class mutant

```
1  public void test2()  {
2    Stack stack0 = new Stack();
3    assertEquals(false, stack0.empty());
4
5    try {
6      stack0.peek();
7
8      fail("Expecting exception:" +
9         "EmptyStackException");
10   } catch(EmptyStackException e) { }
11 }
```

To automatically detect the wrong assertions we executed each test case generated by EvoSuite on the original `Stack` class. If a test case fails, it means that the test case covers the mutation, and thus it likely includes an incorrect oracle. Using this approach we determined that EvoSuite produced 67 test cases that reveal the injected faults for 37 of the 59 mutants. For these test cases we manually double checked the presence of incorrect oracles. To avoid selecting trivial or redundant test cases, we manually iterated over the generated test cases and selected, for each mutant, the test case with the highest number of assertions that has not been already selected. This led to 21 test cases, including a total of 50 assertions. Among these 50 assertions, 27 are correct and 23 are erroneous assertions that need to be fixed.

The simple cases include no case of incomplete documentation because the documentation for the `Stack` class is fairly complete. Therefore, we complemented the simple cases with hard cases. The hard cases were obtained by browsing public bug repositories looking for classes that (a) have a non-trivial interface, (b) include faults, (c) are documented with a Javadoc. For these classes we generated test cases with EvoSuite, re-executed the test cases on the version of the class without the fault, to automatically check if the fault has been revealed; finally, we manually inspected the test cases to confirm the presence of wrong oracles. We thus selected four hard cases (summarized in Table I), originating from the Java libraries Trove4J[6], which provides high performance collection classes, and j8583[7], a Java implementation of the ISO 8583 protocol.

Table II summarizes the contents of the test cases for the hard cases: Rows indicate the individual test cases, whereas columns discriminate the answers that the workers should return for each assertion in the test cases. The crowd has to evaluate 3 correct assertions, 1 assertion that refers to an incomplete Javadoc, 1 meaningless assertion that should be removed and 2 wrong assertions that need to be fixed, for a total of 7 assertions to be evaluated.

B. Empirical Setup for RQ1

To evaluate if workers without a qualification can be used to produce CrowdOracles, we initially submitted 4 HITs from the simple cases to AMT. The 4 HITs include a total of

[6]http://trove4j.sourceforge.net
[7]http://j8583.sourceforge.net

Table I
DESCRIPTION OF HARD CASES

Case Study	Software	Bug Id	Description
TC1	Trove4J	846286	The documentation for method `TObjectDoubleHashMap.get(Object key)` erroneously indicates that it returns true when no value is associated to the given key. This method should return a primitive value, a double, thus it cannot return null.
TC2	Trove4J	3196242	Method `TIntObjectHashMap.getNoEntryKey()` erroneously returns the default value, because the constructor of the class does not properly set field `noEntryKey`.
TC3	Trove4J	3448111	Method `TIntArrayList.retainAll` erroneously throws an ArrayIndexOutOfBoundsException.
TC4	ISO8583	2941743	Method `setAssignDate(boolean)` does not use the passed in argument but erroneously sets field `assignDate` to true.

Table II
DETAILS OF HARD CASES

	Opt1: correct assertion	Opt2: incomplete Javadoc	Opt3: remove the assertion	Opt4: wrong assertion	Tot
TC1	-	1	-	-	1
TC2	1	-	-	1	2
TC3	2	-	1	-	3
TC4	-	-	-	1	1
Tot	3	1	1	2	7

3 correct assertions, 1 assertion that should be removed and 5 wrong assertions that need to be fixed. For each HIT we submitted 50 assignments. To evaluate the performance of the crowd, we measure metrics about both the general ability of the crowd in evaluating the assertions and the ability of the crowd in identifying and correcting the wrong assertions.

The considered metrics are:

- **Correct**: percentage of correct results returned by the crowd
- **Wrong**: percentage of wrong results returned by the crowd
- **WrongAssert**: percentage of wrong assertions identified by the crowd
- **FixAssert**: percentage of wrong assertions correctly fixed by the crowd

C. Empirical Setup for RQ2

To investigate the quality of the results returned by the crowd, we submitted the 21 HITs corresponding to the simple cases and the 4 HITs corresponding to the hard cases. For each HIT we submitted 20 assignments, for a total of 500 assignments returned by the crowd. Since the correct answer (i.e., the option that must be selected among the four available options) to each question is known, the results returned by the crowd have been checked automatically. Every time a field contains a new assertion entered by a worker, we automatically compared the text with the expected assertion (manually specified by us). The answers that did not match our assertions were inspected manually to distinguish the case in which the worker provided a different but still correct assertion, from the case of a worker providing the wrong assertion.

Based on the results returned by the crowd (note that here we used qualified workers only), we compute three groups of metrics. The first group of metrics measures the ability of the crowd in recognizing wrong and correct assertions. The second group of metrics investigates the effectiveness of the crowd when different answers are expected. The third group of metrics measures the ability of the crowd in fixing the incorrect assertions.

As part of the first group we compute the following metrics:

- **TP**: True Positives - the number of wrong oracles correctly classified as wrong
- **FP**: False Positives - the number of correct oracles incorrectly classified as wrong
- **TN**: True Negatives - the number of correct oracles correctly classified as correct
- **FN**: False Negatives - the number of wrong oracles incorrectly classified as correct oracles
- **Precision**: $\frac{TP}{TP+FP}$ - the rate of incorrect oracles in the set of oracles classified as wrong by the crowd
- **Recall**: $\frac{TP}{TP+FN}$ - the rate of incorrect oracles discovered by the crowd

As part of the second group, we aggregate the results according to the option that the worker is expected to select, and we compute the following metrics:

- **Correct**: $\frac{TP+TN}{TP+FP+TN+FN}$ - the rate of correct answers
- **Wrong**: $\frac{FP+FN}{TP+FP+TN+FN}$ - the rate of wrong answers

As part of the third group we compute the following metrics:

- **AsIs**: percentage of fixes returned by the workers that exactly match our expectation
- **Equivalent**: percentage of fixes returned by the workers that are equivalent but not equal to our expectation
- **Wrong**: percentage of wrong fixes returned by the workers

D. Empirical Setup for RQ3

In this research question we investigate if a voting mechanism can be used to automatically distill a correct CrowdOracle from the results returned by the crowd. The idea is that when at least **T**% of the workers agree on selecting the same option between the four available options, the selected option represents the right answer. If there is no agreement on one option, the crowd has been inconclusive about the correctness of the evaluated assertion. We evaluate the quality of the decisions taken by the voting schema for T=60%, 70%, 80% and 90%. We exclude 50% because two options may get enough votes to be selected both at the same time, and we exclude 100% because we assume it is unrealistic to have a crowd perfectly agreeing on a result.

We investigate the voting schema in the same cases considered from RQ2. Thus we submit 20 assignments for each HIT, and we consider both the 21 simple cases and

the 4 hard cases. Since the effectiveness of the voting schema might be affected by the number of assignments submitted per HIT, we evaluate the schema with considering different numbers of **NUM_ASSIG** submitted assignments. In particular, we consider NUM_ASSIG = 5, 10, 15 and 20.

We obtain the results for a number of assignments smaller than the ones that we really submitted by randomly selecting NUM_ASSIG answers from the results returned by the crowd and applying the voting schema to the data pool. To mitigate the effect of randomness we repeat the process 100 times and we report average results.

To evaluate the results, for each combination of values T and NUM_ASSIG, we compute the following metrics:

- **CORRECT**: Percentage of correct results returned by the voting schema
- **INCONCLUSIVE**: Percentage of inconclusive results returned by the voting schema
- **WRONG**: Percentage of wrong results returned by the voting schema
- **DISCOVERED**: Percentage of wrong assertions discovered by the voting schema

Finally, to investigate if, in addition to identifying the wrong and correct assertions, it is possible to automatically fix assertions, we evaluate the following mechanism. When an assertion is classified as wrong, we collect all the assertions provided by the workers. If at least T% of the returned assertions match, we automatically replace the assertion in the test with the one identified by the crowd. We evaluate this strategy by computing for every combination T and NUM_ASSIG the following metrics:

- **FIXED**: percentage of wrong assertions replaced with a correct assertion
- **WRONG FIX**: percentage of wrong assertions replaced with an incorrect assertion
- **MAYBE FIXED**: percentage of wrong assertions with no decision, with the majority of workers agreeing on the new correct assertion (but the number of workers who agree are less than T%)
- **SUGGESTED**: percentage of wrong assertions with no decision, with a minority of workers agreeing on the new correct assertion
- **NO FIX**: percentage of wrong assertions with no decision and no worker suggesting the right fix

V. Results

In this section we present the results obtained for RQ1-3, report additional information about our experience, and discuss the threats to validity.

RQ1: Is it possible to use an unqualified crowd to fix automatically generated assertions?

Table III reports the results obtained by using unqualified workers. We can notice that the unqualified workers selected a wrong option in 62% of the cases, denoting a poor ability of evaluating assertions. The already poor results are even worse when the assertion that is evaluated must be classified as a wrong assertion. In fact in only the 20% of the cases the workers recognized a wrong assertion as wrong. Note that this result is even worse than a random choice, which on average provides the right answer in 25% of the cases. Finally, only a few times (11% of the cases) workers succeeded in specifying the right fix for an assertion that has been qualified as wrong.

Table III
Quality of the results returned by the unqualified crowd

	Unqualified Crowd
Correct	38%
Wrong	62%
WrongAssert	20%
FixAssert	11%

Table IV
Quality of the results returned by the crowd

	Simple Cases	Hard Cases
TP	371	43
FP	46	32
TN	526	53
FN	57	12
Precision	0.89	0.57
Recall	0.86	0.78
Correct	0.90	0.69
Wrong	0.10	0.31

In summary, the results with four simple cases for the `Stack` class clearly indicate that CrowdOracles cannot be produced using an unqualified crowd. We believe that this behavior of the crowd is not specific to the oracle problem, but is general and would likely be observed whenever HITs that require technical knowledge need to be evaluated.

The unqualified crowd is not suitable for performing technical tasks that require specific knowledge.

This, however, is a problem that can be overcome by using only *qualified* workers. A qualification test may reduce the pool of workers to choose from, which may result in longer response times or may require higher renumeration, but it avoids receiving unnecessary wrong responses. For the two remaining empirical questions we used a qualified crowd, obtained according to the process described in Section II.

RQ2: What is the quality of the results returned by a qualified crowd?

Table IV reports data about the performance of the qualified crowd in recognizing the wrong oracles among the assertions that occur in the submitted HITs. As one would expect, we can notice that the qualified crowd behaves definitely better than the unqualified crowd. In fact for the simple cases the rate of wrong answers drops from 62% to 10%. Moreover, the crowd performed a good job in indicating as wrong only the incorrect oracles (precision

Table V
QUALITY OF THE RESULTS PER OPTION

	Simple Cases		Hard Cases	
	Correct	Wrong	Correct	Wrong
Opt 1 (correct assertion)	90%	10%	46%	54%
Opt 2 (wrong Javadoc)	-	-	35%	65%
Opt 3 (remove the assertion)	89%	11%	80%	20%
Opt 4 (wrong assertion)	79%	21%	50%	50%

0.89) and detecting most of the incorrect oracles (recall 0.86).

These good results are essentially confirmed in the hard cases. In fact, the rate of correct answers, even though slightly decreased, is still high (69% of the answers are correct). In the hard cases, the crowd seems to be more conservative in judging oracles, reflected by a relatively high recall (0.78 compared to 0.86 for simple cases), which indicates that still many of the wrong oracles are discovered. However, the precision (0.57 compared to 0.89 for the simple cases) decreases more compared to the simple cases, which suggests that when in doubt the crowd prefers to classify an assertion as wrong. This conservative behavior of the crowd is beneficial because it reduces the risk of overlooking at wrong assertions and increases the number of faults that can be discovered through the CrowdOracle approach.

A qualified crowd can support the CrowdOracle approach.

Table V shows the performance of the crowd per option. We can notice that workers handled assertions that must be removed and wrong assertions in both simple and hard cases well. Surprisingly, the workers address the correct assertions in the simple cases well, while they perform poorly with the correct assertions in the hard cases. This is due to the complexity of the interfaces and lack of clarity of the Javadoc that describes the methods used in the tests. An example is the Javadoc for the constructor of `TIntObjectHashMap`, a `Map` tailored for using keys of integer type, that indicates that the first argument, initialCapacity, is *used to find a prime capacity for the table*. This Javadoc led workers to errors when evaluating the assertion in line 2 of test TC3 (Listing 3): They expected the return value of `capacity()` to be 174 instead of 3, while it is correct that `capacity()` returns 3 [8]. This is reflected by only 35% of the workers correctly answering that the Javadoc does not provide enough information to judge this assertion. We can conclude that the clarity of the documentation is of huge relevance for the performance of the crowd.

Poor documentation strongly degrades the performance of the crowd.

Listing 3. Portion of the test case TC3 for class `TIntObjectHashMap`

```
1 TIntObjectHashMap tIntObjectHashMap0 =
2   new TIntObjectHashMap(174, -1305.8732F, 174);
3 assertEquals(3, tIntObjectHashMap0.capacity());
```

Table VI
ABILITY OF FIXING ASSERTIONS

	Simple Cases	Hard Cases
AsIs	249 (83%)	17 (85%)
Equivalent	13 (4%)	2 (10%)
Wrong	38 (13%)	1 (5%)

Table VI summarizes data about the quality of the fixes suggested by the workers when a wrong assertion is identified. When workers identify wrong oracles, the workers perform exceptionally well in providing the right assertion that should replace the existing one. Our data show that workers tend to perform better with hard cases, which led to 95% of correct assertions vs. the 87% of the simple cases. A possible explanation is that the wrong assertions of the hard cases are detected only by good Java programmers who are also able to properly fix the assertion. The workers provided exactly the assertion that we expected in 83% and 85%of the simple and hard cases, and equivalent assertions in 4% and 10% of the simple and hard cases, respectively. Equivalent assertions are correct assertions that do not syntactically match the one expected. This difference on the percentage of equivalent assertions seems to depend mostly on the number of assignments which tend to mask the outliers in the simple cases. In most of the cases equivalent answers just include additional Java style comments that explain the worker choice.

The crowd performs well at fixing assertions detected as wrong.

RQ3: Is it possible to distill correct CrowdOracles from the results returned by the crowd?

The previous section showed that a qualified crowd can support oracle generation in principle. The aim of this research question is to determine if we can make use of the information gathered from the crowd in making decisions on the correctness of assertions.

Tables VII and VIII summarize the results of the decisions taken as described in Section IV. For the simple cases we can notice that the number of wrong decisions is extremely small: 4.18% in one configuration, less than 2% in seven configurations and 0% for the rest. In addition, the number of correct decisions is extremely high, especially for T=60% (the percentage of correct answers ranges from 92.72% to 94%) and T=70% (the percentage of correct answers ranges

[8]JavaDoc is misleading because the HashMap implementation differs in Trove and Java JDK: the former uses the *initialCapacity* parameter to estimate the initial capacity of the Map, the latter uses the value of the parameter as the actual initial capacity of the Map. For details see http://sourceforge.net/p/trove4j/bugs/137/

Table VII
RESULTS FOR RESEARCH QUESTION RQ3 (SIMPLE CASES)

NUM_ASSIG	T	Correct	Inconclusive	Wrong	Discovered	Fixed	Maybe Fixed	Suggested	No Fix	Wrong Fix
5	60%	93.28	2.54	4.18	89.783	79.421	6.105	0.684	12.421	1.368
	70%	81.28	17.78	0.94	74.261	61.737	7.895	0.842	29.421	0.105
	80%	81.28	17.78	0.94	74.261	50.474	19.158	0.895	29.474	0
	90%	56.08	43.9	0.02	40.522	21.789	16.421	0.211	61.579	0
10	60%	92.72	5.64	1.64	87.957	78.368	6.316	0.684	14.579	0.053
	70%	86.64	13.04	0.32	81.261	67.579	9.316	0.421	22.684	0
	80%	77.58	22.38	0.04	70.522	50.053	15.316	0.158	34.474	0
	90%	63.88	36.12	0	51.348	26.211	21.947	0	51.842	0
15	60%	93.96	5.1	0.94	88.652	79.368	6.316	0.579	13.737	0
	70%	82.2	17.8	0	76.957	63	9.105	0	27.895	0
	80%	75.18	24.82	0	69.348	50.895	12	0	37.105	0
	90%	53.46	46.54	0	36.174	27.421	7.368	0	65.211	0
20	60%	94	6	0	86.957	84.211	0	0	15.789	0
	70%	86	14	0	78.261	63.158	10.526	0	26.316	0
	80%	74	26	0	69.565	52.632	10.526	0	36.842	0
	90%	60	40	0	43.478	31.579	10.526	0	57.895	0

Table VIII
RESULTS FOR RESEARCH QUESTION RQ3 (HARD CASES)

NUM_ASSIG	T	Correct	Inconclusive	Wrong	Discovered	Fixed	Maybe Fixed	Suggested	No Fix	Wrong Fix
5	60%	52.286	15	32.714	64.667	49.5	0.5	0	50	0
	70%	38.143	40.571	21.286	58.667	45.5	4.5	0	50	0
	80%	38.143	40.571	21.286	58.667	45.5	4.5	0	50	0
	90%	21.857	65.143	13	42	23	27	0	50	0
10	60%	53.143	19	27.857	66.667	50	0	0	50	0
	70%	43.571	32.571	23.857	64.667	50	0	0	50	0
	80%	33.143	47.714	19.143	56.667	46	4	0	50	0
	90%	21.143	62.857	16	44.667	30.5	19.5	0	50	0
15	60%	52.857	18.571	28.571	66.667	50	0	0	50	0
	70%	41	38.286	20.714	66.667	50	0	0	50	0
	80%	30	53.429	16.571	58.333	50	0	0	50	0
	90%	15	70.571	14.429	34.667	7.5	42.5	0	50	0
20	60%	57.143	14.286	28.571	66.667	50	0	0	50	0
	70%	42.857	28.571	28.571	66.667	50	0	0	50	0
	80%	28.571	57.143	14.286	66.667	50	0	0	50	0
	90%	14.286	71.429	14.286	33.333	0	50	0	50	0

from 81.28% to 86.64%). Higher values for T and number of assignments minimize the risk of wrong decisions. The choice of the best configuration largely depends on the possibility to tolerate the presence of few wrong oracles in the final test suite.

The best configuration depends on whether to minimize wrong oracles or maximize detection of wrong assertions.

Results indicate that even few assignments per HIT could be effective *if a few wrong oracles are acceptable*. In fact T=60% and 5 assignments per HIT are sufficient to correctly identify 93.28% of the correct assertions, discover 89.78% of the wrong ones, and fix 79.42% of the wrong ones.

If the presence of wrong oracles must be minimized, the best configuration consists of T=70% with 15-20 submitted assignments. Here, the number of correct decisions decreases to about 82%-86%, and the number of inconclusive decisions rises to 14%-18%. To be conservative about the assertions that should occur in a test case, those without decision could be removed, and only those with a decision could be preserved (if evaluated as correct by the crowd) or replaced with the correct one (if evaluated as wrong by the crowd). This configuration reveals 76.95%-78.26% of the wrong assertions, while the rest are evaluated as inconclusive by the crowd, and thus would be eliminated according to the suggested process. When a wrong assertion is discovered, the right assertion is automatically obtained in most of the cases (around 86%), thus allowing to fix wrong assertions in 63% of the cases. When the right assertion is not obtained automatically, the right assertion is always the one with the highest number of preferences, thus it seems reasonable to always suggest the majority choice as a replacement.

For the hard cases the number of wrong answers can be minimized by using T=90%. Even with a few number of submitted assignments, 5-10, the crowd provides wrong decisions only in 13%-16% of the cases, identifies 42%-44.67% of wrong assertions, but provides 65.14%-62.85% of inconclusive results. To lower the number of inconclusive results without maximizing the number of wrong answers a configuration with T=70% and 10-15 submitted assignments can be adopted. This configuration allows to discover 64.67%-66.67% of wrong assertions, provide 41%-43.57% of correct answers, and repair 50% of the wrong assertions.

Other Notes From Our Experience

Our main objective was to evaluate the CrowdOracle approach, with special emphasis on the correctness of the results returned by the crowd. However, there are secondary aspects that are worth to be mentioned.

One interesting aspect is the time passing from the submission of a HIT to the reception of the results. This information does not represent a kind of durable knowledge because it depends on the number of workers registered on a CrowdSourcing platform. However, we want to report data about the state of the practice at the time we conducted the experiment.

According to our empirical investigation the best configurations require submitting a number of assignments per HIT between 5 and 15. When interacting with the crowd without using a qualification, it took only 4 hours (median value) to receive 20 assignments per HIT. However, the quick responses produced by the unqualified crowd resulted of little use for the CrowdOracle approach.

The median time taken by the qualified workers to complete 5, 10 and 15 assignments consisted of 4, 17 and 26 days, respectively. The time required to complete from 5 to 10 assignments per HIT, which covers most of the optimal configurations that we identified, could be already acceptable in a real scenario. If the CrowdSourcing approach gains popularity in the future, the timing aspect would naturally improve, thus increasing the number of configurations that can produce results in few days. The use of professional crowd based testing services, such as uTest[9], instead of free platforms is another option for reducing the response time.

We also want to report that the cost for generating CrowdOracles is extremely small, definitely smaller than hiring someone just for fixing assertions in test cases. In fact, for each completed assignment we paid $0.05\$ + 0.05\$ \times$ *number of assertions in the assignment*, for a total cost per assignment ranging from 0.15$ to 0.2$. The amount paid per HIT is likely to affect the speed and quality of results.

Threats to Validity

Our results might be affected by the crowd registered at Amazon Mechanical Turk, and might be different using other CrowdSourcing platforms. This threat is mitigated by the qualification test. In fact, our qualification test would likely select a similar subset of workers in any platform. We do not expect the results about the quality of the responses would change much by changing platform (on the contrary the timing aspect could change significantly).

The results cannot be generalized, as we used a relatively small combination of simple and hard cases. Even so, the reported data include relevant insights about the quality of the results that can be returned by the crowd, and the performance of different decision processes that can be defined to automatically fix assertions. The reported results can thus be used to setup a baseline for future similar studies.

In principle, different mechanisms can be used to derive conclusions from results returned by the crowd. In this initial study we used what we believe is an obvious voting schema; results might change with different decision processes.

The results could be affected by the techniques that we used to generate the tests and the assertions. To mitigate this issue, we used a state of the art technique to generate tests with assertions; we do not expect results could improve considering other techniques.

[9]http://www.utest.com

VI. Related Work

Many techniques to produce inputs have been proposed over the years. Yet, the problem of the expected outcome persists, and has become known as the *oracle problem* [5]. In general, the term *test oracle* refers to both, the information about the expected behavior, and the procedure to verify this behavior on a concrete execution.

In the case of unit testing, oracles are usually specified in terms of test assertions. Such assertions can be synthesized for automatically generated tests. For example, Randoop [4] can include assertions based on annotated observer methods. Orstra [10] generates assertions based on observed return values and object states and adds assertions to check future runs against these observations. As the number of assertions can be large, and many of the assertions are actually irrelevant for a given test, Fraser and Zeller proposed mutation analysis as a technique to identify fault revealing assertions [11]. A related approach to synthesize assertions aims to distinguish two versions of the same program [12], [13]. In all of these cases, assertions reflect the current behavior of the program, and not the *intended* behavior. A noteworthy exception is Eclat [14], which generates assertions based on a model learned from a set of tests that are assumed to be correct. However, ultimately these assertions also need to be manually verified. Consequently, CrowdOracles can be seen as the logical next step after applying such approaches.

As software evolves, assertions that passed previously might fail on future versions. The idea of test repair, as for example done by the ReAssert tool [15], is to suggest possible assertion fixes to the developer. It is conceivable to apply the CrowdOracle approach also in a scenario of test repair, in order to let the crowd decide whether the program is now faulty, or whether the assertion needs to be fixed.

The use of CrowdSourcing in software engineering experimentation is recently becoming popular. For example, Stolee and Elbaum [6] performed a study on the effects of code smells on user's perference and understandability of web mashups. Their experiences are along the line of our own: Qualification tests are necessary in order to guarantee that resulting data is useful, yet it has a negative impact on the response rate. Applications of CrowdSourcing related to software engineering range from database queries [16] to

graphical perception experiments [17]. However, to the best of our knowledge our work is the first to apply CrowdSourcing to software testing and the oracle problem. In general, although CrowdSourcing is very popular, many questions are still open, and current research is trying to better understand and use CrowdSourcing (e.g., [18]).

VII. Conclusion

The oracle problem is one of the main challenges in software testing, as it makes full automation impossible. To overcome this issue, we have proposed the use of CrowdSourcing in this paper in terms of *CrowdOracles*. From the point of view of the developer, CrowdOracles make it possible to fully automate software testing – on the click of a button, the developer could produce tests, and the crowd would decide which of the tests failed, based on the documentation. Our empirical investigation showed that this approach is indeed possible.

However, our experiments also revealed that the CrowdOracles approach highly depends on the ability to represent the oracle problem in a simple and understandable way. An essential prerequisite for this are clear and well documented interfaces – which is desirable in any case, yet not always done in practice. Similarly, the test cases need to be easy to understand, in order to keep the human oracle costs to a minimum. Consequently, our results point out important directions of future work.

Our current approach relies on the developer providing sufficient API documentation to make it possible for crowd workers to determine the correctness of assertions. In principle, this process could be supported by natural language processing [19], for example by filtering out obvious bugs. A fault detected by the crowd may of course also be a fault in the documentation and not the code – this is also a bug and is just as valuable to know for the developer. In fact, even the information that an assertion cannot be conclusively verified by the crowd points out a problem in the documentation, so it is conceivable to use our approach to test documentations.

CrowdSourcing has opened up new possibilities for software engineering, as demonstrated by the CrowdOracles approach. We believe that CrowdOracles can currently provide practical benefits especially for open-source software. In fact even in presence of long Crowd response time, open source systems with frequent maintenance releases will benefit from test cases generated automatically and validated with CrowdOracles. Furthermore the availability of source code will limit the drawbacks of incomplete documentation.

Our future work will face the issues identified in our study. We are planning an evaluation of CrowdOracles on a broader set of case studies, and the comparison of results obtained with generic and professional crowds. We are also considering the possibility to adopt the CrowdOracle approach as a mean to measure the quality of a program's documentation.

References

[1] N. Tillmann and J. D. Halleux, "Pex: white box test generation for .NET," in *Proc. International Conference on Tests and Proofs*. Springer-Verlag, 2008, pp. 134–253.

[2] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proc. Conference on Programming Language Design and Implementation*, 2005, pp. 213 – 223.

[3] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proc. ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*. ACM, 2011, pp. 416–419.

[4] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *Proc. Object-oriented Programming Systems and Applications*, 2007, pp. 815–816.

[5] E. Miller and W. E. Howden, *Software Testing and Validation Techniques*, 2nd ed. IEEE Computer Society, 1981.

[6] K. T. Stolee and S. Elbaum, "Exploring the use of crowdsourcing to support empirical studies in software engineering," in *Proc. International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 35:1–35:4.

[7] J. Howe, "The rise of crowdsourcing," *Wired*, vol. 14, no. 6, June 2006.

[8] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: a mutation system for Java," in *Proc. International Conference on Software Engineering, tool demo*, 2006, pp. 827 – 830.

[9] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[10] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *Proc. European Conference on Object-Oriented Programming*, 2006, pp. 380–403.

[11] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2011.

[12] R. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *Proc. European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007, pp. 549–552.

[13] K. Taneja and T. Xie, "DiffGen: Automated regression unit-test generation," in *Proc. IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 407–410.

[14] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *Proc. European Conference on Object-Oriented Programming*, 2005, pp. 504–527.

[15] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting repairs for broken unit tests," in *Proc. Int. Conference on Automated Software Engineering*, 2009, pp. 433–444.

[16] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin, "CrowdDB: answering queries with crowdsourcing," in *Proc. Int. Conf. on Management of Data*, 2011, pp. 61–72.

[17] J. Heer and M. Bostock, "Crowdsourcing graphical perception: using mechanical turk to assess visualization design," in *Proc. International Conference on Human Factors in Computing Systems*, 2010, pp. 203–212.

[18] J. J. Horton and L. B. Chilton, "The labor economics of paid crowdsourcing," in *Proc. ACM conference on Electronic commerce*. ACM, 2010, pp. 209–218.

[19] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language API descriptions," in *Proc. International Conference on Software Engineering*, June 2012, pp. 815–825.