Cross-Checking Oracles from Intrinsic Software Redundancy

Antonio Carzaniga
University of Lugano
Switzerland
antonio.carzaniga@usi.ch

Alberto Goffi
University of Lugano
Switzerland
alberto.goffi@usi.ch

Alessandra Gorla
Saarland University
Germany
gorla@st.cs.uni-saarland.de

Andrea Mattavelli
University of Lugano
Switzerland
andrea.mattavelli@usi.ch

Mauro Pezzè
University of Lugano
Switzerland
University of Milano-Bicocca
Italy
mauro.pezze@usi.ch

ABSTRACT

Despite the recent advances in automatic test generation, testers must still write test oracles manually. If formal specifications are available, it might be possible to use decision procedures derived from those specifications. We present a technique that is based on a form of specification but also leverages more information from the system under test. We assume that the system under test is somewhat redundant, in the sense that some operations are designed to behave like others but their executions are different. Our experience in this and previous work indicates that this redundancy exists and is easily documented. We then generate oracles by cross-checking the execution of a test with the same test in which we replace some operations with redundant ones. We develop this notion of cross-checking oracles into a generic technique to automatically insert oracles into unit tests. An experimental evaluation shows that cross-checking oracles, used in combination with automatic test generation techniques, can be very effective in revealing faults, and that they can even improve good hand-written test suites.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Redundancy, test oracles, oracle generation

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568287

1. INTRODUCTION

Test oracles discriminate successful from failing executions of test cases. Good oracles combine simplicity, generality, and accuracy. Oracles should be simple to write and straightforward to check, otherwise we would transform the problem of testing the software system into the problem of testing the oracles. They should also be generally applicable to the widest possible range of test cases, in particular so that they can be used within automatically generated test suites. And crucially, they should be accurate in revealing all the faulty behaviors (completeness, no false negatives) and only the faulty ones (soundness, no false positives).

Test oracles are often written manually on a case-by-case basis, commonly in the form of assertions, for example JUnit assertions.[1] Such input-specific oracles are usually simple and effective but they lack generality. Writing such oracles for large test suites and maintaining them through the evolution of the system can be expensive. Writing and maintaining such oracles for large *automatically generated* test suites may be practically impossible.

It is possible to also generate oracles automatically, even though research on test automation has focused mostly on supporting the testing process, creating scaffolding, managing regression test suites, and generating and executing test cases, but much less on generating oracles [7, 27]. Most of the work on the automatic generation of oracles is based on some form of specification or model. Such oracles are very generic, since they simply check that the behavior of the system is consistent with the prescribed model. However, their applicability and quality depend on the availability and completeness of the models. For example, specification-based oracles are extremely effective in the presence of precise specifications, such as protocol specifications [21], but they are not easily applicable to many other systems that come with informal and often incomplete specifications.

Another classic approach to obtain generic oracles is to use what Weyuker calls a pseudo-oracle [46, 17], that is, another program intended to behave exactly as the original. The actual oracle requires the execution of the two programs with the same input, followed by a comparison between the results of the two executions. The production of an

[1] http://junit.org

alternative version of the program makes this technique completely generic, but it also makes it quite expensive.

There are also interesting solutions somewhere in between model-based oracles and pseudo-oracles. For example, ASTOOT [19], symmetric testing [25] and metamorphic testing [12] do not require complete behavioral models, and instead exploit symmetries and equivalences in the specification. ASTOOT derives tests and corresponding pseudo-oracles from algebraic specifications in the form of pairs of equivalent and non-equivalent sequences of operations. In a similar way, metamorphic and symmetric testing use the commutativity of some operations, or the symmetric behavior of an operation with different inputs, to identify different sets of inputs that should produce the same result.

The testing methods proposed with these techniques are interesting alternatives to both input-specific and completely generic oracles. Differently from input-specific oracles, they are not limited to a fixed set of inputs. And differently from generic oracles, they do not attempt to compute or verify a result using an alternative program but instead use the system as both the original and the alternative program, effectively making the system test itself.

We propose a similar method that is also rooted in the idea of a pseudo-oracle. Specifically, we exploit the intrinsic redundancy of software systems to generate what we call *cross-checking oracles*. Unlike other approaches, we use specifications that are closer to the code and therefore arguably simpler to obtain. We also decouple the generation of the oracle from the test input, obtaining a more versatile method to generate oracles and therefore a more effective testing process.

In very simple terms, we build oracles by having the system under test cross-check itself. These cross-checks exploit the intrinsic redundancy of the system under test. This is a property of various systems that we have studied and characterized [10] and that we have used to implement a self-healing mechanism for Web applications [9] and for general purpose systems [8]. In this paper we describe a method to exploit this intrinsic redundancy to generate oracles and enhance the efficacy of test suites.

As summarized in the next section, the intrinsic redundancy of software systems has various sources, from design for adaptability and generality to reusability and backward compatibility. Also, such redundancy manifests itself at different levels, from single statements, to method calls, subsystems, and entire libraries. The technique we propose uses redundancy at the method level, and in particular generates oracles localized around the invocations of methods (in the test driver) that admit to redundant alternative code, which then leads to potentially fault-revealing cross-checks.

The technique is opportunistic in nature: a cross-check may exercise the right functions, and such functions may be redundant enough to reveal a fault. However, cross-checking oracles are not intended to be complete. On the other hand, cross-checking oracles can be generated and embedded in a test suite *completely automatically,* at practically no cost to the developers and testers.

We develop a technique to generate and activate oracles through an aspect-oriented programming mechanism. Notice that such oracles lead to intermediate checks, as well as checks on the final result, and such intermediate checks may reveal faults whose effects may be masked during the complete execution of the test. We then report the results of an initial experimental evaluation that demonstrates that this method can indeed lead to efficient and effective oracles. In summary, we make the following contributions:

- We introduce the notion of cross-checking oracles obtained by exploiting the intrinsic redundancy of software systems and realized through the embedding and coordinated execution of fragments of cross-checking code.
- We develop a specific concrete technique that makes cross-checking oracles practical and accurate.
- We report the results of an experimental evaluation that shows that cross-checking oracles can be effective especially in conjunction with automatic test generation.

The paper is organized as follows. Section 2 introduces the concept of intrinsic redundancy for software, summarizes the origins and scope of the phenomenon, and shows some cases of intrinsic redundancy in libraries at the method level. Section 3 details a technique to generate and deploy cross-checking oracles obtained from specifications of redundancies in the system under test. Section 4 presents the experimental evaluation of this technique, and in particular shows that cross-checking oracles complement state-of-the-art test case generation tools such as Randoop. Section 5 surveys the main approaches to the generation of oracles with a special focus on those that are most related to this work. Section 6 summarizes the contributions of the paper and describes ongoing research work.

2. SOFTWARE REDUNDANCY

We obtain cross-checking oracles by exploiting the intrinsic redundancy of software systems. In this section we introduce the concept of intrinsic redundancy informally, and discuss some manifestations of this redundancy. The interested reader can find more details about software redundancy and some of its uses elsewhere [10, 11, 24].

Informally, a system is redundant when it can perform the same actions through different executions, either with different code or with the same code but with different input parameters or in different contexts. This happens for example in systems that use different algorithms for the same functionality. For instance, the GNU Standard C++ Library implements its basic stable sorting function using insertion-sort for small sequences, and merge-sort for the general case. The same redundancy arises in libraries that provide specialized implementations of functionalities already available in other components of the system. For instance, the *Trove4j* library implements collections specialized for primitive types that overlap with the standard Java library.

Redundancy can occur in various forms and at different levels of abstractions in a system, from statements to services to entire subsystems. The defining elements of this software redundancy are what is considered *different* in two executions and what is considered *equivalent* in their outcome. For the purpose of this paper, we consider differences in the code of a test that amount to different sequences of method calls of the system under test that then induce different executions of the same test. And for the outcome of those different executions, we consider functional properties and therefore we build our technique on a notion of observational equivalence [28].

```
 1 public boolean put(K key, V value) {
 2   Collection<V> collection = map.get(key);
 3   if (collection== null) {
 4     collection = createCollection(key);
 5     if (collection.add(value)) {
 6       totalSize++;
 7       map.put(key, collection);
 8       return true; // mutation: return false;
 9     } else {
10       throw new AssertionError("Spec␣violated");
11     }
12   } else if (collection.add(value)) {
13     totalSize++;
14     return true;
15   } else {
16     return false;
17   }
18 }
```

```
 1 public boolean putAll(K key, Iterable<? extends V> values) {
 2   if (!values.iterator().hasNext()) {
 3     return false;
 4   }
 5   Collection<V> collection = getOrCreateCollection(key);
 6   int oldSize = collection.size();
 7   boolean changed = false;
 8   if (values instanceof Collection) {
 9     Collection<? extends V> c = Collections2.cast(values);
10     changed = collection.addAll(c);
11   } else {
12     for (V value : values) {
13       changed |= collection.add(value);
14     }
15   }
16   totalSize += (collection.size() − oldSize);
17   return changed;
18 }
```

Figure 1: Methods put and putAll of the AbstractMultimap<K,V> Class from the Guava Library

We say that two executions of a system are equivalent (functionally) if their outcome can not be distinguished by probing the system through its public interfaces. In particular, two different sequences of method calls on an object may leave the object in two different internal states. However, if those two states cannot be distinguished through any sequence of methods of the public interface of the object, then we say that the two sequences are equivalent. Notice that this is an ideal notion of equivalence. In Section 3 we discuss the necessary assumptions and limitations we pose for the actual implementation of the equivalence checks.

Having defined at least informally the notion of redundancy that we plan to exploit to obtain oracles, we now argue, again informally, that such redundancy is indeed present in modern software systems. We start from an example and then provide more evidence based on prior analyses and other observations.

Consider the methods put(K key, V value) and putAll(K key, Iterable<? extends V> values) of the AbstractMultimap<K,V> class of the popular Google Guava library.[2] The put and putAll methods associate a given key with a given value and collection of values, respectively. This suggests that put(k,v) would be equivalent to putAll(k,c) with the same key k and a singleton collection c containing value v.

The put and putAll methods of the Guava library are implemented with different code. Figure 1 reproduces the code of the two methods with only minor formatting changes. The executions of the two methods may in turn invoke the same code (for example, line 12 of put and line 13 of putAll, depending on dynamic binding). Still, one fault in the put method could be detected by a test that runs that method, and by an oracle that compares the result with an equivalent execution of the putAll method. For example, consider a simple mutation of line 8 of method put that causes put to return false instead of true. Such a faulty mutation would be detected by a simple test that calls put to insert one key–value mapping, and by an oracle that does the same through putAll. In this case, the equivalence check is trivial, since the two executions themselves return different values, and therefore are immediately deemed observationally non-equivalent without the need for any additional probing calls.

The effectiveness of test oracles built from related methods such as put and putAll depends on the independence of their code. When related methods share some or most of their code, they may produce the same wrong result and thus may not be able to detect a failure. And indeed related methods often share code and sometimes are almost completely identical, as is the case when a method simply wraps a call to the other with a few minor initialization or termination steps. Still, our past experience indicates that software systems contain many equivalent methods or sequences of methods that are independent enough and therefore usable for their redundancy [8, 9]. The experiments presented in Section 4 also offer further evidence that software is intrinsically redundant and that such redundancy is usable.

Having observed that intrinsic redundancy exists, it is natural to ask how and where this redundancy might arise. Without wanting to explore these questions in great depth, we simply mention some potential sources of useful intrinsic redundancy with some examples.

There are plausible and general reasons to assume that modern software systems, especially modular components, are intrinsically redundant. This is redundancy that is not deliberately introduced by the designers as in N-version programming [5], and therefore that does not incur additional development costs. Design for reusability is a source of intrinsic redundancy. Reusable libraries typically offer different ways of executing the same functionalities. For example, Google's Guava library offers several methods to create immutable collections: *copyOf()*, *builder()*, *of()*, etc. These methods differ in the interface usage, but they are essentially equivalent in terms of the final results. Performance optimization and context-specific optimization are another source of redundancy. As we already mention above, a sorting function may be implemented through multiple different and therefore redundant algorithms, and similarly, one library such as the *Trove4j* library may provide an optimized version of the functionality already provided by another library (in this case, the standard Java library). Backward compatibility is another source of redundancy. For example, the Java 7 standard library contains 45 classes and 365 methods that are deprecated and that overlap with the functionality of newer classes and methods.

[2] http://code.google.com/p/guava-libraries/

There are also various studies that demonstrate that a considerable amount of intrinsic redundancy exists in software systems at various abstraction levels, in different application domains and in various forms. Gabel et al. reported more than 3000 semantic clones in the Linux kernel [22] and Jiang and Su studied the Linux kernel at a different granularity level and found more than 600,000 semantically equivalent code fragments [29], where semantic clones and semantically equivalent code refer to code fragments that produce the same results without sharing the whole code.

In previous work, some of these authors identified more than 150 semantically equivalent sequences of method calls in popular Javascript Web APIs, such as YouTube, Google Maps, and JQuery, and more than 4,000 semantic equivalences in Java applications and libraries of non-trivial size and complexity, including Apache Ant, Google Guava, Joda-Time, and Eclipse SWT. The same previous work also shows that this redundancy can be put to a good use, in particular to implement a self-healing mechanism based on "automatic workarounds" [8, 9].

For the experiments reported in this paper, we identified a total of 529 equivalences over 873 methods. We did that manually, starting from the Javadoc specifications of the considered libraries, with an effort of about 40 person-hours by PhD students with some familiarity with the libraries. Both our group and the group of Martin Monperrus at the University of Lille are investigating the possibility of identifying redundant methods automatically with dynamic analysis. The results obtained so far indicate that we can identify redundant methods automatically with negligible effort.[3]

3. CROSS-CHECKING ORACLES

We now describe in detail the technique we have developed to generate test oracles that exploit the intrinsic redundancy of software systems.

The technique exploits the redundancy present at the level of method calls. More specifically, we consider systems in which a method call (possibly a static method) can be replaced by one or more other calls plus possibly some glue code. This redundancy can be expressed through equivalences of the form:

$$\begin{aligned}&\mathit{class}.\mathit{method}(\mathit{type}_1\ \mathit{param}_1, \ldots)\\&\equiv \{\text{code using } \mathit{target} \text{ of type } \mathit{class} \text{ and}\\&\quad \mathit{param}_1 \text{ of type } \mathit{type}_1, \ldots\}\end{aligned}$$

$$\begin{aligned}&\mathit{class}.\mathit{static\_method}(\mathit{type}_1\ \mathit{param}_1, \ldots)\\&\equiv \{\text{code using } \mathit{param}_1 \text{ of type } \mathit{type}_1, \ldots\}\end{aligned}$$

The right-hand side of the equivalences is in principle arbitrary code but in practice amounts to little more than one or a few alternative calls. Below are two examples:

```
AbstractMultimap.put(String key, Object value)
≡ { List list = new ArrayList();
    list.add(value);
    target.putAll(key,list); }

AbstractMultimap.clear()
≡ { for (String key : target.keySet()) {
      target.removeAll(key);
    } }
```

The first example expresses the equivalence between the put and putAll methods of the AbstractMultimap class of the Guava library. The second example expresses an analogous equivalence for two other methods of the same class.

The technique we propose takes equivalence specifications and produces oracles that can be automatically deployed within any existing test suite. The technique is based on an instrumentation of the code of the test that, for every call corresponding to the left-hand side of an equivalence specification, performs a cross-check that executes both the original call and the equivalent code specified in the right-hand side, and then checks that the two executions were indeed equivalent. Thus the main ingredients of the technique are:

- *Automatic deployment:* a mechanism to deploy an oracle for every call corresponding to the left-hand side of an equivalence.
- *Cross-check execution:* a mechanism to correctly execute both the original call and the corresponding equivalent code, and then to compare their outcomes.
- *Equivalence check:* a decision procedure to verify that the outcomes of the two executions are indeed equivalent.

Notice that for the technique to be truly automatic in general, all the above mechanisms must be independent of the test and of the system under test, which is one of the essential technical difficulties of implementing our technique. Another technical difficulty is to avoid spurious differences between the executions of the original call and the equivalent code. This is where we see the most significant limitations of our technique. For example, such differences may be caused by interference between the two executions. One form of interference that our current implementation does not handle well is through the use of multiple threads in the original or in the equivalent code. Other problematic forms of interference are through files and other input/output operations. Yet other spurious differences might arise from non-determinism such as the simple use of the system clock, which may return different results if the two executions are shifted in time.

We now describe our approach to implement each technical aspect of cross-checking oracles.

3.1 Automatic Deployment

We translate each equivalence into an *advice* that we then implement and deploy into test programs using the AspectJ system.[4] See Figure 2 for an example. An equivalence E defines an advice class A_E. The left-hand side of the equivalence defines the join points for the advice, which is where the body of the advice is executed and that corresponds to the specified method invocations. Then, the left-hand side and the right-hand side of the equivalence translate directly into an originalCall and an equivalentCode method of the advice class A_E, respectively.

The execution of an oracle derived from an equivalence E proceeds as follows: (1) the advice implemented in A_E.advice is invoked right before a call corresponding to the left-hand side of E; (2) A_E.advice saves the target object and the parameters of the call in specifically declared member variables of the A_E class; (3) A_E.advice calls the crossCheck method of its base class that implements a generic cross-check procedure (described in the next section); (4) at some point the

[3] Martin Monperrus, private communication, 2013.

[4] http://eclipse.org/aspectj/

```
class Oracle1 extends BasicOracle {
  private String key;
  private Object value;

  @Around("call(put(String,Object))&&target(map)&&args(key,value)")
  public void advice(AbstractMultimap m, String k, Object v) {
    target = m;
    key = k;
    value = v;
    return crossCheck(); // calls BasicOracle.crossCheck()
  }

  Boolean originalCall() {
    return target.put(key,value);
  }

  Boolean equivalentCode() {
    List list = new ArrayList();
    list.add(value);
    return target.putAll(key,list);
  }
}
```

```
abstract class BasicOracle {
  protected Object target;

  abstract Object originalCall();
  abstract Object equivalentCode();

  protected boolean crossCheck() {
    Object target_orig = target;
    Object target_copy = copy(target);

    Object orig_res = originalCall();
    target = target_copy;
    Object copy_res = equivalentCode();

    assert(equivalence(orig_res,copy_res));
    assert(equivalence(target_orig,target_copy));
  }
}
```

Figure 2: Oracle Implementation and Deployment Through Advice Classes

crossCheck method calls the originalCall and equivalentCode methods that invoke the implementations defined in A_E, which can then refer to the target and parameters of the original call saved by A_E.advice; (5) the crossCheck method calls a generic equivalence check to compare the results and the state of the target object after the executions of originalCall and equivalentCode; (6) if the comparison detects an inconsistency then the oracle signals a failure, otherwise the execution of the test continues with the result of the originalCall.

3.2 Cross-check Execution

Cross-checking oracles compare the outcome of the execution of an original method call and an equivalent code on a target object. We now describe the mechanism that supports these two executions on the same object within a test. A fundamental requirement is that the execution of the equivalent code should not affect the execution of the original call, and vice-versa, and since a test may contain multiple oracles, it should also not affect the remainder of the test. In other words, the execution of the equivalent code should be invisible, and yet its outcome must be well visible to the comparison procedure to obtain an accurate cross-check.

Thus, the challenge in implementing cross-checks is to obtain two independent executions whose outcomes can be later compared accurately, and with only one of them continuing beyond the cross-check, as if the other one never existed—and to do all that efficiently. To solve this challenge we tried a number of cross-check patterns, striving to make the cross-check as accurate as possible.

We first explored a solution in which the cross-check forks the execution of the test into a completely separate virtual machine [31]. However, this solution proved problematic, because it is complex and inefficient, but more importantly because it requires an equivalence check based on the serialization of the state of the target object, which incurs a high rate of false positives.

We therefore turned to a solution in which the original call and the equivalent code, as well as the comparison procedure, execute in the same virtual machine. In this case, without the isolation guaranteed by the platform, the main problem is to maintain the state of the execution consistent with an execution of the original test without cross-checks. We tried two approaches: one using a checkpoint mechanism and one using a copy of the target object.

With the checkpoint mechanism we obtain separation by undoing the effects of the equivalent code by reverting to the prior state of the test. First we create a checkpoint of the state of the test; then execute the equivalent code; save the state of the target object; restore the state of the test to the checkpoint (but retaining the saved state of the target object); and then execute the original call. At this point the equivalence check compares the state of the target object (after the execution of the original call) with the state of an object created from the previously saved state of the target object (after the execution of the equivalent code). Unfortunately, this solution also incurs false positives because, similar to forking a separate virtual machine, it must perform an equivalence check on a de-serialized object. And again this yields too many false positives.

We then focused on a solution in which the cross-check would execute the equivalent code on a copy of the target object: the cross-check starts by creating a copy of the target object, then it executes the original call on the original target and the equivalent code on the copy, and then it checks the equivalence of the two executions by comparing the original target with the copy, as well as the results of the execution.

This solution is perhaps simple, but with a number of refinements it also proved to be the most effective one. We used a generic deep-copy library[5] that we had to tweak in order to reduce the occurrence of false positives. In particular, we changed the copy procedure to avoid making copies of singleton objects (for example, those declared as static final) whose references are often used as special values in various data structures (for example, compared with the == operator). We also changed the way the library handles some collections, by copying their exact structure as opposed to using the collection's specialized putAll or addAll methods.

[5] http://code.google.com/p/cloning/

No matter what copy mechanism one might use, the execution of the equivalent code on a copy may still interfere through static variables or other shared data. To mitigate this problem, we also instrumented the oracles to detect interference whenever one execution writes a value that is then read by the other. However, ultimately we discarded this detection mechanism because it was itself a source of false positives, which in the end proved to be much more problematic than the interference between executions.

Finally, to further reduce the rate of false positives, we augmented the basic copy pattern with a cascade of preliminary checks. We first check that the copy alone would not introduce false differences. Further we check that the execution of the same original sequence on two copies would still result in an equivalent state, and only then we proceed with the execution of the equivalent sequence, and compare its outcome (on a copy) with that of the original sequence (on the original target). These additional checks are quite effective and indeed absolutely necessary to discard spurious differences and therefore to obtain accurate oracles.

3.3 Checking Equivalence

We compare the effects of different executions that are expected to be semantically equivalent. This requires comparing the immediate results of the two executions, as well as the state of the system after the two executions. In particular, in the context of object oriented systems, comparing the state of an execution requires comparing the objects used or produced during the execution.

We developed a hybrid technique to compare two objects. On the one hand, we have the standard equals method. However, we observed that equals tests are sometimes too conservative and lead to false positives. This is because the default implementation simply compares the two object references, which is an overly strict equality condition for our purposes especially since we start with two different objects obtained through a deep copy. Notice also that we are interested in determining the *semantic* equivalence of two objects, and the default equals method fails to do that. Some applications override the default method with their specific and more semantically meaningful equivalence checks. However, those checks would often directly or indirectly delegate the check to the equals method of other classes (typically for referenced objects) that in turn may use the default reference comparison.

To address the shortcomings of the equals method, we implemented a generic procedure to check the observational equivalence of two objects. In essence, this procedure tries to prove that two objects (states) are *not* equivalent. Thus this check tries to find a counter-example, meaning a sequence of calls that, when applied identically to the two objects, would produce one or more different results. This difference is decided, for each call, by comparing the output of the call on the two objects. When a call returns a primitive value, the comparison is immediate. When the call returns an object, the comparison may in turn require a recursive check for the observational equivalence of the returned objects. The search for a counter-example uses a simple random selection and is currently limited to methods without parameters. If the search succeeds, then the equivalence check returns false and the oracle reports a failure.

We then combine the use of the generic observational equivalence check with the original equals to improve efficiency, since the observational equivalence is more accurate but also typically more expensive than the equals method. In the current implementation we limit the generic search to sequences of up to twenty calls, including direct and recursive calls, and we rely on the application-specific checks whenever they are available. To do that, we invoke the equals method in the cross-check procedure, but we also instrument the default equals check (implemented in the Object class) to run our observational equivalence check. This way, the cross-check uses the observational equivalence check unless an application-specific equals is available, and it does so even when the application-specific equals invokes the default check directly or indirectly. For brevity we omit other details of the implementation of the equivalence check but we make the implementation available for further analysis and use.[6]

4. EXPERIMENTAL EVALUATION

We conducted an experimental evaluation of the idea of cross-checking oracles embodied in the technique described above in Section 3. The purpose of this evaluation is to demonstrate that we can exploit the intrinsic redundancy of reusable components to generate cross-checking test oracles, and that such oracles are effective in revealing the presence of faults in real systems.

We already know that systems are intrinsically redundant, and our experiments readily demonstrate that the technique indeed works, meaning that cross-checking oracles *can* be deployed and used with minimal runtime overhead. In this evaluation we do not discuss runtime performance, but we can say that for our experiments we executed millions of tests with tens of millions of cross-checking oracles, and in all cases the tests terminate within a few seconds and most often in fractions of a second (except some that result in infinite-loop failures). We conclude that cross-checking oracles are a practical technique, and the most important remaining question is whether they can also be *effective.*

Generally speaking, we measure effectiveness in terms of the ability of the testing process as a whole to reveal faults. This includes, and depends crucially on, the choice of test cases. We consider both hand-written and automatically generated test suites with which we explore two specific research questions:

RQ1 How do cross-checking oracles perform within hand-written test cases? In particular, are they better or worse than hand-written oracles on the same test inputs? And if and when they are worse, can they still improve the effectiveness of hand-written test cases if used in conjunction with hand-written oracles?

RQ2 How do cross-checking oracles perform within automatically generated test suites? Do they improve the fault-revealing ability of those tests in comparison with generated implicit oracles, such as exceptions and contract violations? Do they achieve a good fault revealing rate in absolute terms and in comparison with hand-written test cases?

We attempt to answer these questions experimentally using seeded faults.

[6] http://star.inf.usi.ch/star/cross-check

4.1 Evaluation Setup

The subjects of the evaluation are classes taken from the following three non-trivial open-source Java libraries.

Guava is a large utility library developed by Google that contains several classes to support collections, caching, concurrency, string processing, etc.[7]

Joda-Time is a library to process dates and times that supports several calendars and is compatible with the Java date and time classes.[8]

GraphStream is a library to model and analyze dynamic graphs.[9]

Table 1: Subjects Considered in the Evaluation and Number of Available Equivalences

Subject Class		Methods	Equiv.
Guava	ArrayListMultimap	25	29
	ConcurrentHashMultiset	27	32
	HashBiMap	20	16
	HashMultimap	24	29
	HashMultiset	26	30
	ImmutableBiMap	25	19
	ImmutableListMultimap	30	34
	ImmutableMultiset	32	50
	LinkedHashMultimap	24	30
	LinkedHashMultiset	26	31
	LinkedListMultimap	24	29
	TreeMultimap	26	28
	TreeMultiset	35	37
Joda-Time	DateMidnight	118	20
	DateTime	153	27
	Duration	44	6
GraphStream	SingleGraph	107	41
	MultiGraph	107	41

From these libraries we consider concrete classes for which we have a set of equivalences and also a set of test cases written by the developers. We first select a subset of the classes of the Guava collections and Joda-Time that we already analyzed and for which we identified a list of equivalences in previous studies [8]. We then consider other classes from the GraphStream library for which we identify additional equivalences. In total, we select 18 subject classes: 13 from Guava, 3 from Joda-Time, and 2 from GraphStream, with a total of 529 equivalences. We then translate each equivalence into an AspectJ aspect as described in Section 3. Table 1 lists and briefly characterizes the subject classes selected for our experiments.

We assemble two types of test suites for each subject class. We first take the tests written by the developers, which we refer to as the *hand-written* test suites. We then generate test suites using Randoop, one of the most popular tools for generating unit tests for Java [39]. We refer to these as the *generated* test suites. The hand-written tests come with corresponding oracles that are also hand-written by the developers. The generated tests also come with oracles. In particular, Randoop generates oracles based on previously seen values, which amount to regression oracles, and oracles based on implicit language contracts, such as the fact that the equals method must be symmetric. We refer to these latter ones as *implicit* oracles. Since we do not examine regression testing in this evaluation, we only consider the implicit oracles in the generated test suite.

We evaluate the effectiveness of cross-checking oracles by means of a systematic mutation analysis. We generate mutants of the subject libraries (Guava, Joda-Time, and GraphStream) using the Major mutation analysis framework [30], activating all the mutation operators that the tool supports. To simulate the normal use of the technique by a developer, we generate a test suite for each mutant. We then test each mutant with both the corresponding generated test suite and the hand-written test suite, each with various combinations of oracles. We then record which mutant is revealed (or "killed") by which oracle and in which test.

In summary, we proceed with the following experimentation process for each subject class:

1. We seed faults in the class under test and in its direct and indirect super-classes with Major. We then inject those mutations in the source code. We do that because Major instruments the bytecode directly, but we must have the mutation in the source code to use Randoop to generate a specific test suite.

2. We create a set of *generated* tests and we select a set of relevant mutants for those tests as follows:

   (a) For each mutant of the class under test, we use Randoop to generate a test suite. We specify the class under test as the only target of the test and we use a timeout of 90 seconds for Randoop to generate each test suite. We discard the few mutants for which Randoop fails to generate a test suite.

   (b) We run each generated test suite on its corresponding mutant.

   (c) When running a generated test suite on the corresponding mutant, we check whether the test suite covers the mutation. We then discard all the mutations that could not be covered by the corresponding test suite. In other words, since we are interested in evaluating the oracles rather than the tests, we disregard those tests within which an oracle would not even have a chance to detect the fault.

3. We select a set of the generated mutants for the *hand-written* tests in a similar way, by retaining only those mutants that are covered by at least one test.

4. We then activate the cross-checking oracles and run both the generated and hand-written test suites on the respective selected mutants. We activate one equivalence at a time to identify precisely which equivalence would kill which mutant in which test. We also record the kill scores of the hand-written oracles in the hand-written tests, and the kill scores of the implicit oracles in the generated tests. In any case, since our focus is on the effectiveness of the oracles, we do not count runtime exceptions in the kill scores, unless the exceptions are raised within a test oracle.

[7] https://code.google.com/p/guava-libraries

[8] http://joda-time.sourceforge.net

[9] http://graphstream-project.org

Table 2: Results of the Experiments on the Classes Under Test

	Generated Tests				Hand-Written Tests			
	selected mutants	mutants killed by which oracle			selected mutants	mutants killed by which oracle		
Subject Class		implicit	cross-checking	both		hand-written	cross-checking	both
ArrayListMultimap	30	7	11	0	80	30	1	41
ConcurrentHashMultiset	89	0	64	0	150	54	2	60
HashBiMap	12	4	1	0	14	5	0	6
HashMultimap	40	7	12	0	90	21	0	47
HashMultiset	66	0	35	0	83	27	1	38
ImmutableBiMap	13	2	1	1	30	6	0	3
ImmutableListMultimap	22	1	8	0	43	19	0	11
ImmutableMultiset	49	6	5	0	72	44	0	4
LinkedHashMultimap	26	11	10	0	95	29	0	46
LinkedHashMultiset	53	0	33	0	89	31	0	39
LinkedListMultimap	24	2	4	0	61	45	0	12
TreeMultimap	35	1	10	0	81	24	0	43
TreeMultiset	113	0	37	0	121	27	2	55
DateMidnight	81	2	11	0	134	72	0	24
DateTime	148	0	8	0	181	78	0	34
Duration	146	2	26	0	152	119	0	26
SingleGraph	243	0	10	0	248	71	12	14
MultiGraph	254	0	11	0	253	74	14	13

4.2 Evaluation Results

Table 2 summarizes the high-level results of our experiments. For each subject class we report the number of mutants selected for the experiments, which are those whose mutation is executed by at least one test case. We then count the numbers of mutants "killed" by the generated and hand-written test suites. For the generated test suites we report how many mutants were killed only by the implicit oracles, only by the cross-checking oracles, and by both. Similarly, for the hand-written tests we count how many mutants were killed only by the hand-written oracles, only by the cross-checking oracles, and by both.

Notice that all the kill scores reported in this paper are *true* positives, as determined by an analysis of the individual tests and the corresponding failure reports. We discuss this analysis in Section 4.3.

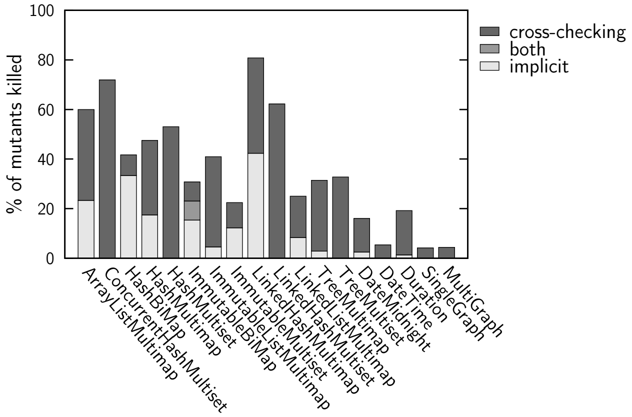

Figure 3: Kill Ratios for Generated Tests

Figures 3 and 4 illustrate the relevant data from Table 2 in terms of kill ratio, meaning the portion of covered mutants killed by each type of oracle. Consider first the case of generated test cases shown in Figure 3. In general, the graph shows that cross-checking oracles are quite effective in revealing faults for many of the subject classes (**RQ2**). Notice that the reported kill ratios are *conservative* measures, first because some selected mutants may be ineffective, but also because, in order to focus on the effectiveness of the oracles, we do not consider failures resulting from runtime exceptions. Such failures typically abort the execution of a test and therefore may shortcut the execution of our oracles, which may also be able to reveal the fault.

Figure 3 also shows that the implicit oracles can also be effective. However, a further analysis of the successful uses of the implicit oracles reveals that *all* those cases correspond to mutations of the equals method, which happens to be checked by implicit contracts (symmetry and reflexivity of the equals relation).

Figure 3 shows that cross-checking oracles are most effective with classes of the Guava library and least effective with GraphStream. A plausible explanation is that the Guava subjects are collection classes that presumably possess more redundancy than the GraphStream subjects. A review of the GraphStream code supports this hypothesis. Another potential issue with GraphStream is that many of its core methods are private and therefore not visible from the user documentation. This means that we did not even consider those methods in writing equivalences. A knowledgeable developer could write more direct and therefore more effective equivalences. However, in order to take advantage of equivalences on private methods, we would have to extend our oracle deployment technique, perhaps using reflection.

The case of GraphStream highlights some limitations of cross-checking oracles, but it is nevertheless a positive case. In fact, our cross-checking oracles evidenced a true and previously unknown fault confirmed by a developer.[10] Also, as we see below in Figure 4, GraphStream is the subject with which our cross-checking oracles achieve the highest improvement over hand-written oracles.

[10] https://github.com/graphstream/gs-core/issues/109

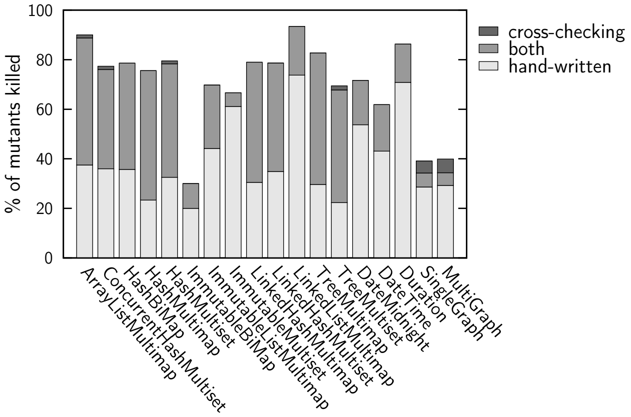

Figure 4: Kill Ratios for Hand-Written Tests

The experiments with generated test suites are representative of the primary usage we envision for cross-checking oracles. However, our evaluation also demonstrates that cross-checking oracles are useful also in conjunction with hand-written test suites (**RQ1**). Considering that the subject cases we used are part of mature software libraries that come with many carefully written tests, it is not surprising to see that specific hand-written oracles are superior at revealing faults. However, our generic cross-checking oracles still achieve significant kill ratios (up to 53% of all covered mutants) and in some cases even *improve* the effectiveness of the test suite by revealing additional faults (up to +6% overall and +16% relative to the hand-written oracles).

4.3 Accuracy

We validate the accuracy of all oracles by analyzing all test failures. Since different cross-checking oracles can be deployed within the same test case, possibly multiple times, we analyze all the individual failures signaled by a cross-checking oracle within a failed test. For each oracle (cross-checking or other) we examine the test case, the oracle invocation, and the mutation, to see if the failure is real or spurious. As soon as we find one real individual failure, we record that the test accurately killed the mutant thanks to the oracle. Conversely, if none of the individual oracles signals a true failure, we classify that test failure as a false positive.

As it turns out, this analysis was not very time consuming for our cross-checking oracles. In all those cases, for all the true positives we found a true individual failure almost immediately, and for all the false positives (two) we had to analyze a single individual failure for each test failure. The same analysis was instead quite laborious for the implicit oracles. One reason is that the implicit oracles reported a large number of failures. Another reason is that most of those failures were violations of contracts on the equals relation that required an in-depth analysis. In particular, for each failure we had to determine whether the violation of the contract (for example, the symmetry of the equals relation) was attributable to a mutation within the equals method, or to an indirect effect of a mutation elsewhere, or to a deliberately non-conforming (non-symmetric) implementation of the equals method provided by the developers.

Table 3: Accuracy of Cross-Checking Oracles

	Reported Test Failures			
	Generated		Hand-Written	
Subject Class	total	false	total	false
ArrayListMultimap	11	0	42	0
ConcurrentHashMultiset	64	0	62	0
HashBiMap	1	0	6	0
HashMultimap	12	0	47	0
HashMultiset	35	0	39	0
ImmutableBiMap	2	0	3	0
ImmutableListMultimap	8	0	11	0
ImmutableMultiset	7	2	4	0
LinkedHashMultimap	10	0	46	0
LinkedHashMultiset	33	0	39	0
LinkedListMultimap	4	0	12	0
TreeMultimap	10	0	43	0
TreeMultiset	37	0	57	0
DateMidnight	11	0	24	0
DateTime	8	0	34	0
Duration	26	0	26	0
SingleGraph	10	0	26	0
MultiGraph	11	0	27	0

Table 3 shows, for both the generated and hand-written tests, the number of false positives incurred by our cross-checking oracles compared to the total number of failures signaled by the same oracles. The results are excellent in all cases. The only two false positives (for ImmutableMultiset) are due to a limitation in the deep-clone used within the cross-checking oracles. In particular, a clone and the original object may have a different hash, which may lead to a different observable behavior whenever the object is used as a key in a hash table.

4.4 Threats to Validity

The most crucial threat to the validity of our claims is the limited number of classes used to evaluate the effectiveness of our technique. The number of subjects was limited primarily by the cost of the analysis we conducted on each subject. In particular, since we wanted to report accurate results for our cross-checking oracles (true positives) we had to analyze each individual failure manually.

Another threat to validity is the nature of the faults we considered in our study. We resorted to seeded faults produced by Major, and these faults may not be representative of real faults in the classes we analyzed. However, according to the study by Andrews et al. [2], mutation analysis can be used as a valuable way to assess the effectiveness of a test suite. Yet another threat to the validity of the evaluation is in the analysis of false positives. We performed this analysis manually and we may have misjudged some cases. In order to mitigate the effects of this threat, we will publish all the data collected in the experimentation.[11]

5. RELATED WORK

Baresi and Young surveyed the research literature on testing oracles in 2001 [7] and back then they found very little on how to generate proper oracles. Most of what they found were techniques to generate oracles from formal specifications such as algebraic specifications [3, 23], assertions [33, 42, 44], Z specifications [32, 35], context free grammars [18], and finite state machines for protocol conformance [21].

[11] http://star.inf.usi.ch/star/cross-check

A recent survey by Harman et al. [27], which also covers advances in the last decade, finds again techniques to generate oracles from various forms of specifications, in particular from models [45], contracts [16, 34], assertions [4, 14], and annotations [15], and also from models and assertions derived from dynamic traces [37, 47] (including tests) that amount to regression oracles. Harman et al. also find a few other specific techniques that exploit symmetries in the input [13, 25, 48] or human knowledge [40], but still conclude that the problem of automatically generating test oracles is "less well explored and comparatively less well solved" than other important problems in software testing.

One such problem that has received much attention in recent years is the generation of test cases. Test cases can be generated automatically at the unit [4, 6, 14, 20, 39], integration [41] or system level [26]. Input generation techniques do not compete with but rather complement the oracle generation technique we propose. However, some of these techniques also generate oracles. These oracles are either assertions of generic contracts (e.g., x.equals(y)==y.equals(x)) or they are assertions of previously seen values, and therefore amount to regression oracles.

Recently, Staats et al. have proposed a technique that can suggest oracles by observing the actual behavior of the unit under test, and by selecting among different candidates the ones that have higher mutation scores [43]. This technique is more akin to a discovery of invariants than a generation of algorithmic oracles.

The technique we propose is most related to the notion of an implicit oracle, whereby the system under test contains within itself the ability to replicate and therefore check some of its functionalities. Symmetric testing embodies this notion by exploiting the commutativity of the function under test to generate different but supposedly equivalent inputs [25]. Thus symmetric testing compares the results of the same function called with two permutations of the inputs (for example, mult(a,b)==mult(b,a)). Metamorphic testing also exploits equivalences as implicit oracles, but relies on different rather than symmetric inputs (for example, sin(x)==sin(PI-x)) [13, 48]. Murphy et al. refine the idea of metamorphic and symmetric testing in a framework that, given a set of metamorphic properties, can automatically generate and execute test cases at the system level [36]. Very similar ideas were developed earlier in the context of fault tolerant systems by Amman and Knight, who propose data diversity as a technique to execute the same program on different but equivalent inputs and then select the output with a voting mechanism [1].

All these techniques share the same basic idea of testing a function by exploiting symmetries and equivalences of that function. Our technique is more general, since we can also exploit symmetries and equivalences across different functions, as well as other forms of redundancy that cannot be expressed as simple commutative relations.

Similarly to our technique, ASTOOT exploits symmetries across different functions to automatically generate test cases with implicit oracles [19]. However ASTOOT assumes that the component under test comes with a full set of algebraic specifications from which it derives sequences of operations that should or should not be equivalent. Therefore, ASTOOT generates test oracles that are bound to the generated test cases, which are themselves limited to operations that are provably equivalent according to the specifications. By contrast, our technique produces oracles that are more general (although partial) and therefore that are more likely to reveal faults whose effect propagates to any of the equivalent operations. Also, crucially, these more general oracles can be easily integrated within any test case and therefore can be coupled with any test case generation technique.

6. CONCLUSIONS

In recent years we observed and studied the intrinsic ability of software systems to perform the same actions in different ways. Our intent was (and still is) to exploit this cheap form of redundancy to make software systems more reliable. With this paper we propose a technique to use this same redundancy for testing. In particular, we describe a technique to create and automatically deploy test oracles.

The technique we propose is valuable also because oracles have been somewhat neglected in the efforts to assist and automate software testing. Moreover, the technique is also significant because it is almost completely independent of the test input, and therefore can be combined with various automated test input generation techniques for which the only available oracles are very basic forms of assertions.

The main use of our technique is in conjunction with the many tools that automatically generate test inputs. These tools, like Randoop and Evosuite, generate large amounts of test inputs with very basic oracles, like checks of runtime exceptions, simple checks on the symmetry of equals, and verification of non-regression problems [20, 38]. The lack of good oracles reduces the effectiveness of automatically generated test suites. Enhancing test inputs with manually designed test oracles has prohibitive costs that nullify the advantages of generating test inputs automatically. Our oracles can be generated automatically with almost no overhead and can reveal semantic failures that are likely impossible to identify with simple checks, as shown in the experimental data reported in the paper. The comparison with carefully hand-written oracles indicates that our technique not only generates oracles that can reveal important failures that would otherwise escape automatically-generated tests, but that also identifies a surprisingly high percentage of the failures revealed by ad-hoc test oracles, and also improve those oracles by revealing additional failures.

The only manual step of the technique is the identification of redundant methods. We are currently working on a formalization of equivalence and redundancy, and on techniques to automatically identify redundant methods. These techniques use interface specifications and common patterns to discover potential equivalences, and then dynamic analysis to validate that those equivalences indeed hold. The preliminary results are promising and we expect to be able to infer redundant code with good precision and with no additional effort by developers, thus eliminating the only human cost that, although proven not too high in our experiments, still impacts on the overall cost of the approach.

7. ACKNOWLEDGMENTS

This work was supported in part by the Swiss National Science Foundation with projects *SHADE* (grant n. 200021-138006) and *ReSpec* (grant n. 200021-146607) and by the European Research Council Advanced Grant *SPECMATE* (n. 290914). Some experiments were conducted on machines of the Swiss National Supercomputing Center (CSCS).

8. REFERENCES

[1] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*, pages 402–411, 2005.

[3] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transaction on Software Engineering*, 26(1):55–69, 2000.

[4] W. Araujo, L. C. Briand, and Y. Labiche. Enabling the runtime assertion checking of concurrent contracts for the Java modeling language. In *ICSE'11: Proceedings of the 33rd International Conference on Software Engineering*, pages 786–795, 2011.

[5] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[6] L. Baresi, P. L. Lanzi, and M. Miraz. TestFul: an evolutionary test approach for Java. In *ICST'10: Proceedings of the 2010 International Conference on Software Testing Verification and Validation*, pages 185–194, 2010.

[7] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Department of Computer and Information Science, 2001.

[8] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *ICSE'13: Proceedings of the 35th International Conference on Software Engineering*, pages 782–791, 2013.

[9] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for Web applications. In *FSE'10: Proceedings of the 2010 Foundations of Software Engineering conference*, pages 237–246, 2010.

[10] A. Carzaniga, A. Gorla, and M. Pezzè. Healing Web applications through automatic workarounds. *International Journal on Software Tools for Technology Transfer*, 10(6):492–502, 2008.

[11] A. Carzaniga, A. Gorla, and M. Pezzè. Handling software faults with redundancy. In R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M. H. ter Beek, editors, *Architecting Dependable Systems VI*, LNCS, pages 148–171. Springer, 2009.

[12] T. Y. Chen, S. C. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[13] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Z. Q. Zhou. Metamorphic testing and beyond. In *STEP'03: Proceedings of the 11th International Workshop on Software Technology and Engineering Practice*, pages 94–100, 2003.

[14] Y. Cheon. Abstraction in assertion-based test oracles. In *QSIC'07: Proceedings of the 7th International Conference on Quality Software*, pages 410–414, 2007.

[15] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP'02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 231–255, 2002.

[16] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: Adaptive random testing for object-oriented software. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 71–80, 2008.

[17] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 Conference*, pages 254–257, 1981.

[18] J. D. Day and J. D. Gannon. A test oracle based on formal specifications. In *SOFTAIR'85: Proceedings of the 2nd Conference on Software development tools, techniques, and alternatives*, pages 126–130, 1985.

[19] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3:101–130, 1994.

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

[21] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transaction on Software Engineering*, 17(6):591–603, 1991.

[22] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE'08: Proceedings of the 30th international conference on Software engineering*, pages 321–330, 2008.

[23] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, 1981.

[24] A. Gorla. *Automatic Workarounds: Exploiting the Intrinsic Redundancy of Software Systems*. PhD thesis, University of Lugano, Switzerland, 2011.

[25] A. Gotlieb. Exploiting symmetries to test programs. In *ISSRE'03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 365–375, 2003.

[26] F. Gross, G. Fraser, and A. Zeller. EXSYST: search-based GUI testing. In *ICSE'12:Proceedings of the 34th International Conference on Software Engineering*, pages 1423–1426, 2012.

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

[28] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, 1980.

[29] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA'09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, 2009.

[30] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *ASE'11: Proceedings of the 26th International Conference on Automated Software Engineering*, pages 612–615, 2011.

[31] K. Kawachiya, K. Ogata, D. Silva, T. Onodera, H. Komatsu, and T. Nakatani. Cloneable JVM: a new approach to start isolated Java applications faster. In *VEE'07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 1–11, 2007.

[32] J. Mcdonald. Translating object-z specifications to passive test oracles. In *ICFEM'98: Proceedings of the 1998 International Conference on Formal Engineering Methods*, pages 165–174, 1998.

[33] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1st edition, 1988.

[34] B. Meyer, I. Ciupa, A. Leitner, and L. L. Liu. Automatic testing of object-oriented software. In *SOFSEM'07: Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science*, pages 114–129, 2007.

[35] E. Mikk. Compilation of Z specifications into C for automatic test result evaluation. In *ZUM'95: Proceedings of the 9th International Conference of Z Users*, pages 167–180, 1995.

[36] C. Murphy, K. Shen, and G. Kaiser. Automatic system testing of programs without test oracles. In *ISSTA'09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 189–200, 2009.

[37] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP'05: Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 504–527, 2005.

[38] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA'07: Proceedings of the 22nd Conference on Object-Oriented Programming Systems and Applications*, pages 815–816, 2007.

[39] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, 2007.

[40] F. Pastore, L. Mariani, and G. Fraser. CrowdOracles: Can the crowd solve the oracle problem? In *ICST'13: Proceedings of the 2013 International Conference on Software Testing Verification and Validation*, pages 342–351, 2013.

[41] M. Pezzè, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *ICST'03: Proceedings of the 2013 International Conference on Software Testing, Verification and Validation*, 2013.

[42] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transaction on Software Engineering*, 21(1):19–31, 1995.

[43] M. Staats, G. Gay, and M. P. E. Heimdahl. Automated oracle creation support, or: how i learned to stop worrying about fault propagation and love mutation testing. In *ICSE'12: Proceedings of the 34th International Conference on Software Engineering*, pages 870–880, 2012.

[44] R. N. Taylor. An integrated verification and testing environment. *Software: Practice and Experience*, 13(8):697–713, 1983.

[45] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[46] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[47] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA'11: Proceedings of the 21th International Symposium on Software Testing and Analysis*, pages 353–363, 2011.

[48] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. H. Tse, F.-C. Kuo, and T. Y. Chen. Automated functional testing of online search services. *Software Testing, Verification and Reliability*, 22(4):221–243, 2012.