Automatic Generation of Cost-Effective Test Oracles

Alberto Goffi
University of Lugano – Faculty of Informatics
via G. Buffi 13, 6904 Lugano, Switzerland
http://www.people.usi.ch/goffia – alberto.goffi@usi.ch

ABSTRACT

Software testing is the primary activity to guarantee some level of quality of software systems. In software testing, the role of test oracles is crucial: The quality of test oracles directly affects the effectiveness of the testing activity and influences the final quality of software systems. So far, research in software testing focused mostly on automating the generation of test inputs and the execution of test suites, paying less attention to the generation of test oracles. Available techniques for generating test oracle are either effective but expensive or inexpensive but ineffective. Our research work focuses on the generation of *cost-effective* test oracles.

Recent research work has shown that modern software systems can provide the same functionality through different execution sequences. In other words, multiple execution sequences perform the same, or almost the same, action. This phenomenon is called intrinsic redundancy of software systems.

We aim to design and develop a completely automated technique to generate test oracles by exploiting the intrinsic redundancy freely available in the software. Test oracles generated by our technique check the equivalence between a given execution sequence and all the redundant and supposedly equivalent execution sequences that are available. The results obtained so far are promising.

Categories and Subject Descriptors

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

General Terms

Verification

Keywords

Software testing, intrinsic redundancy, cross-checking oracles

This paper describes a work at an initial stage.

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

1. RESEARCH PROBLEM

Research in software testing has focused on automating many aspects of the testing process such as creating the scaffolding, generating and executing test cases, maintaining and managing test suites. A neglected, although essential, aspect of software testing is the oracle problem: the issue of deciding whether the output of a test case is correct or not [17]. While there are completely automated tools to generate test inputs, only few techniques are available to generate test oracles. In most of the cases designing and implementing test oracles are still manual and expensive activities [2, 3].

Approaches proposed so far to generate oracles are either inexpensive and ineffective or effective but very costly. For instance, *implicit oracles* [11] are easy to obtain at practically no cost. At the same time, implicit oracles are mostly incomplete, since they are not able to identify semantic and complex failures, but they can only reveal general errors like system crashes, null pointer dereferences or unhandled exceptions. On the other hand, *specified oracles* [11] can be generated from several kind of specifications, such as algebraic specifications or formal models of the system behavior. Specified oracles are effective in identifying failures, but defining and maintaining formal specifications is expensive to the point that such specifications are very rare.

The research problem we investigate is how to generate *cost-effective* test oracles: oracles that are, at the same time, more complete than implicit oracles and less expensive than specified oracles. We aim to address this problem exploiting the redundancy intrinsically present in software systems.

Recent research work has shown that modern software systems are somewhat redundant, in the sense that the same functionality can be achieved through different execution sequences [5, 6]. Informally, software is redundant when it provides the same functionality through, at least slightly, different code. We argue that this kind of redundancy is an intrinsic characteristic of software and can be present because of several reasons such as design for reuse, backward compatibility, and performance optimization. Intrinsic redundancy can be found at different abstractions levels in software systems, from single statements to functions to entire subsystems. In object-oriented systems, redundancy can be found at method call level, when the user can obtain the same functionality from the system through different sequences of method calls called *equivalent sequences*.

The following equivalent sequences for a Java class implementing the Map interface exemplify the concept of redun-

dancy at method call level:

$$\texttt{map.containsValue(value)} \equiv \texttt{map.values().contains(value)} \quad (1)$$

The method `containsValue` checks whether the given `value` is present in the target `map`. The same functionality can be obtained using the method `values`, that returns all the values in the map, followed by the method `contains`. Notice that the left side and the right side of the equivalence may execute different code, like in the implementation of the methods `containsValue(value)` and `values().contains(value)` in the class `ArrayListMultimap` of the Google Guava library.

Intuitively, whenever two sequences of operations intended to be equivalent do not produce the same result, we are facing a faulty behavior. Our intuition is that equivalent sequences can be used as test oracles. Given a test case, such test oracles check the equivalence between the normal execution of the test case and the execution of the test case where some statements are replaced with equivalent ones. Figure 1 shows how equivalent sequences can be used as test oracles. We have a test case exercising the software under test and invoking the method `map.containsValue(value)` that is equivalent to `map.values().contains(value)` (Equivalence 1). Before executing `map.containsValue(value)`, we duplicate the state, we execute both `map.containsValue(value)` and the equivalent statement `map.values().contains(value)`, and we compare the results obtained by executing the two sequences. If the two sequences behaved differently, the oracle signals a failure; otherwise, the test case execution continues normally. To highlight the nature of the mechanism, we call such test oracles *cross-checking oracles*, since we let the system under test cross-check itself.

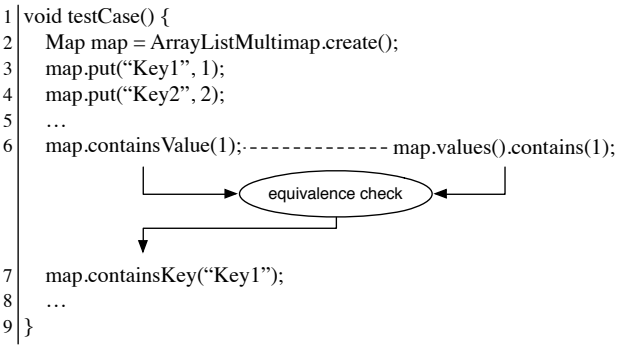

Figure 1: Test oracle from intrinsic redundancy.

Our research hypothesis is that the software is intrinsically redundant, and such redundancy can be encoded as equivalent sequences that can be used to generate cost-effective test oracles.

The major contribution of our research will be the design, the implementation and the validation of a technique to exploit the intrinsic redundancy for generating cost-effective test oracles. This includes the definition of a mechanism to automatically generate and execute alternative and equivalent sequences of operations, and the definition of a mechanism to check the equivalence of the executions.

https://code.google.com/p/guava-libraries/

2. BACKGROUND AND RELATED WORK

The problem of the automatic generation of test oracles is relatively less addressed than other testing problems such as the generation of the test inputs.

In a recent survey, Harman et al. classify test oracles in three categories: specified oracles, implicit oracles, and derived oracles [11]. *Specified oracles* are test oracles generated from formal specification of the system behavior. For example, ASTOOT is a technique to generate test suites along with test oracles from algebraic specifications [7]. Similarly to our approach, ASTOOT generates oracles that check the equivalence between two different execution scenarios. Unlike our approach, ASTOOT requires the algebraic specifications of the system under test. In general, specified oracles are effective in identifying system failures, but a formal specification of the system behavior must be available. Having and maintaining formal specifications is expensive, and thus specifications are rarely created and maintained. The applicability of specified oracles is therefore limited.

As stated in Section 1, *implicit oracles* are inexpensive, since they do not require any domain knowledge or any kind of additional information. At the same time, implicit oracles have a limited effectiveness, since they are incomplete. Implicit oracles are exploited by state of the art tools for generating test suites for object-oriented programs like EvoSuite [9] and Randoop [16].

Derived oracles are derived from properties of the systems under test or artifacts other than specifications. For example, in the context of regression testing oracles can be derived from previous versions of the software under test. In this case, the derived oracles will check that the new version of the system behaves as the previous one [14, 15]. For the test suites they generate, EvoSuite and Randoop derive test oracles from previous versions of the system under test [9, 16]. Oracles can also be derived from invariants automatically inferred from system executions [8]. Additionally, oracles can be derived from properties of the system under test. For instance, symmetric testing [10] and metamorphic testing [20] are two techniques that generate test oracles by exploiting symmetries and metamorphic relations of the system under test. In particular, symmetric and metamorphic testing use symmetries in the behavior of some operations with different inputs or with a permutation of the input. When multiple and independent versions of the system under test are available, pseudo-oracles can be used to check the correctness of the system [19]. Pseudo-oracles are yet another kind of derived oracles that check the consistency of the results of the different versions of the systems, when the same functionality is executed. An inconsistency can reveal a fault in one or more versions of the system. Based on a similar idea, techniques like N-version programming [1] and recovery blocks [18] rely on redundancy *intentionally* added to software systems to guarantee some level of reliability.

Our approach is rooted in the pseudo-oracles idea but, instead of using different and independent implementations of a system, we use the intrinsic redundancy already present in software systems. In this way, our technique does not incur major additional costs, and it is applicable even when only one version of the system is available. This kind of redundancy has been used to automatically recover from runtime failures in JavaScript and Java systems [5, 6], and this work aims to investigate how to use this redundancy to effectively generate test oracles.

3. RESEARCH QUESTIONS

In this thesis, we plan to investigate the problem of generating effective test oracles without incurring high costs. We intend to address this problem by exploiting the intrinsic redundancy of software systems to generate test oracles automatically. In detail, we plan to address the following research questions:

Q1 Can we exploit the intrinsic redundancy of software systems to generate cross-checking oracles automatically?

Q2 Are cross-checking oracles effective in revealing faults?

Q3 Can cross-checking oracles be generated at low cost?

4. APPROACH AND CHALLENGES

In the first part of our research, we studied the feasibility of the approach by designing a technique to generate, deploy, and execute cross-checking oracles. The input of our technique is a list of equivalent sequences that encode the intrinsic redundancy of the system under test, and the output is a test oracle for such system consisting in several inline checks.

The technique works as follow: it examines the code of the system under test and identifies the statements for which we have at least one equivalent sequence. For each identified statement, it instruments the system under test to enable the execution of the original sequence along with the corresponding equivalent sequences. Also, it instruments the system under test to activate the execution of a decision procedure to check the equivalence of the sequences, once they are executed.

To implement the technique we have to face several challenges. A first challenge is finding a suitable way to encode the equivalent sequences. Since we aim to generate oracles automatically from the equivalent sequences, they should be easy to write for developers and, at the same time, easy to use in the context of our technique.

A second challenge consists in executing the equivalent sequences. Ideally, all the sequences equivalent to a given statement should run in parallel and should not influence each other. The readers should notice that all the executions are required to start from the same system state. Furthermore, the executions have to be transparent in the sense that they should not affect the execution of the system under test. In other words, the execution of cross-checking oracles should not modify the behavior of the system under test.

Another challenge stems from the equivalence check. Cross-checking oracles shift the complexity of verifying the correctness of a particular output to verifying the equivalence of different executions. We consider equivalent two executions that produce an equivalent output and lead the system to an equivalent state. In the context of object-oriented software, the output is usually an object and the system state consists of several objects. While comparing primitive values is fairly easy, checking the *semantic* equivalence of two objects is way more complex. The notion of equivalence we use is derived from the *observational equivalence* defined by Hennessey and Milner [12]. We consider equivalent two objects that are not distinguishable through their public interfaces. In particular, two objects are observational equivalent when we can not find a sequence of method calls that produces one, or more, different results when applied to the two objects. A precise evaluation of the observational equivalence would require sequences of calls up to infinite length. We approximate the observational equivalence considering sequences of method invocations of a finite length k.

5. CURRENT STATUS

We tackled the challenges described in Section 4 by implementing a prototype of our technique targeting Java systems [4]. In the prototype, we use aspect-oriented programming to instrument the system under test and execute the equivalent sequences. Currently, equivalent sequences are manually translated into aspects.

To execute the equivalent sequences starting from the same system state, we implemented a deep-clone mechanism. In a nutshell, the original sequence is executed starting from the original system state, while the equivalent sequences are executed starting from copies of the same system state. The current mechanism is not completely safe, since the deep-clone mechanism may not produce completely disjoint objects. For example, static fields are shared between the original object and the deep-cloned objects. The complex structure of objects also affects the deep-clone mechanism. As an example, let us consider a list containing several objects. The deep-cloned list contains objects that are different from the objects contained in the first list. Thus, all the operations based on the hash code of an object work on the original list, but not on the cloned one. For instance, if we try to remove a given element from the lists, the element will be removed from the original list, but not from the cloned list. To limit these problems, we control the result of the deep-clone operation by means of two checks. The first check verifies the equivalence between the original and the cloned object right after its creation. The second check verifies the equivalence between the results of applying the original sequence both on the original and on the cloned object. These checks significantly reduce false positives due to the deep-clone mechanism.

For the equivalence check we rely on an hybrid approach. Each Java object has a method called `equals`. This method should check the equality of two objects, but the default implementation checks the object identity (whether the two references are actually referring the same object). Therefore, we exploit the `equals` method only when developers provided their own implementation overriding the default one. When a useful implementation of the `equals` method is missing, we rely on an implementation of the observational equivalence. We implemented the observational equivalence using sequences of methods calls of length 20, repeating the checking process at most 10 times, each time with a different sequence of method calls.

We evaluated our technique applying the prototype in the context of unit testing of Java systems. We extracted the intrinsic redundancy from a subset of the classes of Guava and JodaTime, and we coded the equivalent sequences into aspects. Then, we generated the oracles for the classes under test. We evaluated the effectiveness of the cross-checking oracles by means of mutation analysis. We seeded mutants in the selected classes with Major [13], and we generated one test suite for each mutated version with Randoop [16]. Then we ran all the test suites with and without our oracles, and we measured the amount of mutants killed by the test suites in both cases. Cross-checking oracles turned out to be effective in revealing seeded faults, when applied to automatically

http://www.joda.org/joda-time/

generated test suites. In fact, cross-checking oracles detected up to the 72% (30% on average) of the seeded faults with very few false positives (0.7%). Oracles generated by Randoop produced lots of false positives (due to uninteresting violations of object-oriented contracts) and detected only a small amount of the faults (11% on average).

We evaluated our technique also with hand-written test suites. Cross-checking oracles resulted to be effective in revealing seeded faults using the developers' tests. Our oracles found up to 53% of the faults, and improved the effectiveness of the oracles written by developers in 4 out of 16 cases.

Currently, of our technique can not be applied to multithreaded programs, included all the programs that interact with the graphical user interface. Also, programs with I/O are problematic since the effects of a sequence can influence the execution of the others.

Although the results are promising, there are several aspects of the approach that can be improved to achieve better results. We aim to find a method to execute the equivalent sequences in a safe way avoiding all the potential side-effects, possibly including the side-effects related with multithreading and I/O. We plan to improve the equivalence check, experimenting different types of heuristics for *efficiently* checking the equivalence of objects. To date, discovering and encoding the intrinsic redundancy have to be done manually. Our research group is currently working on the design of an automated technique to discover the intrinsic redundancy of software systems. This technique could further enhance the automation level of our approach.

6. EVALUATION PLAN

We plan to validate our idea experimentally, by applying the approach to a broad and representative set of software systems. We will evaluate the effectiveness of cross-checking oracles by measuring the soundness and completeness in a series of experiments using both seeded and real faults.

We will compare the effectiveness of our approach with both implicit oracles and hand-written oracles, as well as other kinds of oracles. We expect cross-checking oracles to be considerably more effective than implicit oracles and to be not far from the effectiveness of hand-written oracles.

We plan to evaluate the overall cost of applying our technique. We will consider both the time required for the instrumentation and the overhead on the execution time of the test suites. So far, the main human cost derives from discovering and encoding the intrinsic redundancy. The definition of a mechanism to automatically identify the intrinsic redundancy will consistently reduce the already low cost of the approach.

7. REFERENCES

[1] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 11(12), 1985.

[2] L. Baresi and M. Young. Test Oracles. Technical report, University of Oregon, 2001.

[3] A. Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *2007 Future of Software Engineering*, 2007.

[4] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè. Cross-Checking Oracles From Intrinsic Software Redundancy. In *Proc. of the 36th International Conference on Software Engineering*, 2014.

[5] A. Carzaniga, A. Gorla, A. Mattavelli, M. Pezzè, and N. Perino. Automatic Recovery from Runtime Failures. In *Proc. of the 35th International Conference on Software Engineering*, 2013.

[6] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic Workarounds for Web Applications. In *Proc. of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010.

[7] R.-K. Doong and P. G. Frankl. The ASTOOT Approach to Testing Object-Oriented Programs. *ACM Trans. on Soft. Engineering and Methodology*, 3, 1994.

[8] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly Detecting Relevant Program Invariants. In *Proc. of the 22nd International Conference on Software Engineering*, 2000.

[9] G. Fraser and A. Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[10] A. Gotlieb. Exploiting Symmetries to Test Programs. In *Proc. of the 14th International Symposium on Software Reliability Engineering*, 2003.

[11] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A Comprehensive Survey of Trends in Oracles for Software Testing. Technical report, University of Sheffield, Rep. CS-13-01, 2013.

[12] M. Hennessy and R. Milner. On Observing Nondeterminism and Concurrency. In *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*. Springer Berlin, 1980.

[13] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler. In *Proc. of the 26th International Conference on Automated Software Engineering*, 2011.

[14] L. Mariani, S. Papagiannakis, and M. Pezzè. Compatibility and Regression Testing of COTS component-based Software. In *Proc. of the 29th Intl. Conference on Software Engineering*, 2007.

[15] C. Pacheco and M. D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In *Proc. of the 19th European Conference on Object-Oriented Programming*, 2005.

[16] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *Proc. of the 29th International Conference on Software Engineering*, 2007.

[17] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley and Sons, 2008.

[18] B. Randell. System Structure for Software Fault Tolerance. *SIGPLAN Notes*, 10(6), 1975.

[19] E. J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4), 1982.

[20] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen. Metamorphic Testing and its Applications. In *Proc. of the 8th International Symposium on Future Software Technology*, 2004.