12

Cloud Theory Infected: Using MapReduce To Exhaustively Verify Behaviour

Julian Friedman, Manuel Oriol

Abstract—The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.
The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.The abstract goes here.

Index Terms—Test generation, Cloud Computing, Map/Reduce, unit Testing, Hadoop

1 Introduction

THEORIES are a powerful technique allowing tests to be written about properties which hold against a large series of values whenever an assumptions about these values is true[ref]. They contrast with more traditional scenario-based tests which assert only how code behaves for a particular set of values. The advantage of theories is that they can make more general and useful statements about the code, and often describe more completely what a test aims to prove.

A simple traditional unit test is presented below using JUnit 4.4 syntax:

```
@Test
public void greaterThanThree() {
    assertThat(foo(3),
                    is(greaterThan(3)));
}
```

A theory, by contrast, makes an assertion which is true for all values in a particular range, as below:

```
@Theory
public void alwaysGtThree(int input) {
    assertThat(foo(input),
                    is(greaterThan(3)));
}
```

Theories, of course, are only as good as the sample of the input domain on which they are tested. Various approaches have been used to address this. At the most basic, the programmer or tester can provide a fixed set of input values for the tests. This approach is simple and efficient, but in most cases provides little guarantee that the theory has really been proven against the full input domain. Randomly selecting values from the space of possible values has been used with some success to fill this gap [ref], using an automated tool to generate input values based on introspection of composite objects and random generation of primitive values. Although one may guess that this would leave much of the input space uncovered, experimental results suggest that in many cases this will uncover a good proportion of the bugs in a given program [ref Meyer, Oriol, Satisfying Test Preconditions through Guided Object Selection]. Further, search-based techniques attempt to generate input values intelligently, for example using evolutionary algorithms [ref], white-box analysis [ref] and invariant-finding tools such as Daikon [ref]. The JUnit Factory web service [ref] uses a variety of these techniques and heuristics to generate input values for theories.

For many applications these techniques are sufficient, however there are cases in which they fail. In particular, a stochastic technique relies on being able to quickly generate enough inputs to cover the input domain, and an input domain which is relatively evenly distributed such that bugs are distributed within it such that a sampling method will find most of the bugs. Ideally a stochastic or directed technique is roughly equivalent to an opinion poll; it provides a good approximation of the result of the eventual election at a fraction of the cost of polling the entire population. Such a technique is effective (mostly) for elections, but not for other scenarios. For example if

- *Julian Friedman is an Emerging Technology Specialist working for IBM Cloud Labs and IBM Emerging Technology Services, and a Research Engineer on the Large Scale Complex IT Systems project. E-mail: julz.friedman@uk.ibm.com*
- *Manuel Oriol is with York University [..]*

attempting to find a murderer in a large population, DNA testing every thousandth person will meet limited success. In the absence of knowledge of how a bug is likely to be distributed in the input domain, stochastic testing is likely to be useful only as a tool for finding bugs, but not as a tool which will assure us that bugs do not exist (or that a property truly holds for an entire input domain).

Cloud computing provides IT resources as a service in a pay-per-use manner. Services such as Amazon's EC2 and IBM's Smart Business Cloud provide access to large numbers of "on demand" virtual machines which can be paid for by the hour. The availablility of these resources can make techniques which previously would be prohibitively expensive practical to use.

Up to now the idea of exhaustively testing a code base, by generating all of the inputs for a particular theory has seemed likely to be prohibitively expensive. In this paper we consider using Cloud Computing resources to partition Theory tests across a large number of machines in order to thoroughly test an application. We argue that for certain applications, particularly safety critical systems, by making this technique cost effective and easy to use, this approach may supplement tools such as formal methods and static analysis in providing assurance of the behaviour of the application. Further, if succesful this technique could be extended to general Design By Contract verification.

The paper is structured as follows. Section II describes the architecture of our solution for partitioning and distributing theory tests across a set of machines using Map/Reduce. Section III discusses our experimental results running against a variety of open source systems. Section IV is a discussion of these results and the practical limits of these techniques. Section V concludes with thoughts about related work and future research directions.

2 ARCHITECTURE

The open-source hadoop implementation of the Map/Reduce paradigm is used to distribute the jobs. Using this basic framework to run the tests enables the use of the many existing, relatively cheap map/reduce clusters which are available on a pay-per-hour basis, such as Amazons Elastic Map Reduce service [ref].

Hadoop can be thought of as the combination of a distributed file system and a task management framework which effectively distributes data-parallel tasks against the filesystem. Much of the power of the system is its ability to send the code to the data rather than pulling data to the code as in more traditional systems. Further, the simple nature of Map/Reduce allows all of a large class of problems which can be expressed in it to be parallelized and optimized by a relatively simple framework [ref].

Map/Reduce is an algorithmic skeleton[ref] which encourages the expression of a problem in a parallelizable form by splitting the work in to two phases. The first, 'map', phase runs a series of independent jobs over partitions of the input data. The second, 'reduce' phase combines the results of the map jobs in to a final result.

In our case each test case is fully independent (in fact, this is not only more efficient, but needed to ensure each test is unaffected by the others) but must be run against a large number of input values; this pattern is highly parallelizable using a series of map tasks against the input data. The input to each map task is a subset of the range of values for a particular theory parameter. The reduce jobs receive the passes and failures from the map phase and combine them in to a single summary result.

We have constructed a JUnit theory runner which automatically creates a map/reduce job based on the given parameters and returns passes and failures to JUnit, allowing the results to appear within the IDE. Because it is important that the tool be easy to use within a programmers workflow, we have also created a plugin for the popular Eclipse Java IDE which automatically packages the project under test in to a JAR file and runs the test cases across a remote cloud using Amazon's Elastic Map Reduce service[ref], returning the results in the standard graphical view with the IDE. This allows one click verification of the theories in a project and the remote environment creates virtual machines for the test run and destroys them afterwards.

[Diagram]

2.1 Pluggable Test Data generation

For simple inputs, such as integers, we can generate all values very simply, however for many real world cases we cannot trivially generate inputs. A particularly difficult case is textual input. While we could, of course, generate every possible piece of text under a certain length as input this (very) quickly becomes intractable. Luckily, for many applications with textual inputs, there is a smaller domain of valid inputs which we want to investigate. As an example we may wish to test against all valid HTML pages constructed from a particular set of tags under a given length, or all pages found on the internet in the case of a parsing tool; a much smaller space than all possible texts of a given length.

Our tool supports pluggable test data generation factories which can be used to provide valid candidate inputs for a given input domain. We support two mechanisms for the use of these factories. In many cases the test designer is concerned with a particular input domain and will wish to specify a factory directly in a similar manner to tools such as JUnit [ref] allow for local test data generation, however in others

a class may itself be able to provide a factory which can generate valid instances of itself. For this reason we allow a java annotation on a target class to suggest a factory which can be used to create it. The key advantage of this technique is that these factories can be used to automatically create other classes which take the base class as a constructor argument throough recursion, and can be used to create mock objects [need more detail on these probably] for collaborator classes.

2.2 Limiting the test size safely

[Argue that we can split the program in to a pass known to be safe under certain preconditions, and a pass that we are confident limits input to a particular postcondition. Each can be tested independently without testing the full space of inputs.]

2.3 Generation of Mock Collaborator Objects

[..]

2.4 Pre-distribution of test data

Large data-sets tend to be expensive to create and distribute. Further, in many cases multiple tests require the same test data. Rather than generating the data during the test, as existing solutions tend to do, our approach is to create the test data the first time a test is run if it does not exist. When the test is run we check the distributed filesystem for a directory named uniquely for the test data generation strategy and version. If it does not exist the factory is allowed to produce the test data and write it to the given distributed directory. If a factory changes its generation method significantly, for example to fix a bug or add a new case, it can update a static public integer within its definition to indicate we should consider regenerating the test data.

2.5 Integration with Eclipse

[..]

3 Validation

We have validated our approach by exhaustively testing a number of open source packages.

[.. Fill out all of this section ..]

4 Discussion

[This overlaps too much with the intro I think, need to rewrite to discuss the results in validation once we have them]

Broadly we can consider two ways to cover a large input space. We might decide to fully explore the entire space, or we may take a random sampling of the space. One could say this is the difference between an election and an opinion poll. For many purposes sampling the space, particularly using an intelligent sampling which is weighted towards the most likely errors (for example using invariants analysis[ref] or other white-box techniques [refs]) will discover a large portion of the errors in a program at a fraction of the testing cost. So where is the value in exhaustive testing? This essentially occurs in two cases:

Firstly there are applications for which a sampling method does not provide the necessary assurance, safety-critical systems being the primary example. These systems have often resorted to tools such as formal methods which also introduce a large cost and can fail due to differences between the formal model and the eventual implementation[ref]. We argue that in this case exhaustive testing can be a practical tool, if large scale computing resources is accessible and easily usable on a pay-per-use basis, as it increasingly is today.

Secondly, for some applications it is difficult for an automatic tool to discover the inputs which will adequately cover the input space, in particular for structured textual inputs the full space of potential inputs is extremely large but only a small subset are valid inputs for the program. In this case the majority of randomly generated test data is likely to be quickly rejected without exercising much code (for example consider testing a Java compiler against randomly generated textual strings; the majority will be almost instantly rejected). White box tools may fare better here, being able to adaptively tailor the inputs to attempt to explore all invariants, but again doing this with a large space of textual inputs becomes extremely difficult [ref].

For both of these cases, we assert that exhaustive testing is sometimes a necessary evil. This is particularly valuable in combination with regression testing, for example, testing that a new compiler version produces the same output as a previous version against a large body of existing code. Randomly selecting classes is unlikely to find the one or two edge cases where behaviour subtly varies.

4.1 Other Use Cases

[- Verify that a virus checker does not produce false positive results for any page within WikiPedia unless that page is black listed - Whiley uses theorem prover.. Supplement with CTs - ..]

5 Conclusion

The availability of relatively cheap, pay-per-hour ”utility” computing resources provided by Cloud Computing services brings techniques which would previously have been impractical in to the reach of software engineers. We have demonstrated that exhaustively testing properties of a codebase against many types of inputs is now feasible. Our technique

goes beyond simply finding bugs in a codebase and allows for assurance that a particular property holds for all given input values. We have shown that using Map/Reduce resources we can achieve [measured speed-up] in scaling and distributing this task.

5.1 Related Work

Cloud Theories build heavily on work on Theory Tests and Design by Contract. [JML, JUnit, TestNG, Theory Infected refs]

Stochastic testing tools such as Yeti and JCrasher [..]

Rather than exhaustively testing a codebase we can attempt to find bugs by intelligently adapting our input data towards values likely to cause bugs by analysing code invariants or branch structure. Tools such as [ref DEVONThink/SearchBased papers]

5.2 Future Directions

We have shown that exhaustive and theory-based tests against large input domains can be effectively parallelized using Map/Reduce techniques and demonstrated that for many problems these techniques may now be a cost-effective complement to tools such as Formal Methods and Static Analysis. Further research will look to investigate whether similar techniques can also be adopted to parallelize and scale other testing techniques such as invariant-based test data selection and other search-based techniques, particularly with distributed genetic algorithms. Extending the model here to testing JML-based contracts would allow exhaustive specification of code contracts as well as Theories.

[..]

ACKNOWLEDGMENTS

The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...

REFERENCES

[1] H. Kopka and P.W. Daly, *A Guide to LaTeX*, third ed. Harlow, U.K.: Addison-Wesley, 1999.