International Journal of Engineering Sciences Paradigms and Researches (IJESPR)
(Vol. 23, Issue 01) and (Publishing Month: August 2015)
(An Indexed, Referred and Impact Factor Journal)
ISSN (Online): 2319-6564
www.ijesonline.com

Testing Abstract Data Types: A Formal Goal Oriented Approach

Amal Awad Mirghani Yassin

Faculty of Computer Science and Information Technology,
Sudan University of Science and Technology, khartoum, Sudan
aamy_22@hotmail.com

Publishing Date: 5th August 2015

Abstract

Whereas software testing is known to be able to prove the presence of faults, but not their absence, it can also be used to build a statistical argument regarding the likelihood of failure free operation. In this paper, we elaborate on this argument to present a testing method that is part of an integrated software validation process, whereby testing and proving methods play complementary roles in advancing product quality.

Keywords: *Software testing, program proving, program specification, test oracle, software tools.*

1. Introduction

Software testing is the activity where a software product is executed on sample data and its behavior is judged with respect to the specification that the program is intended to satisfy. A complementary technique to ensure or verify the correctness of software products is program verification, a static method that attempts to prove by logical reasoning that the program is correct, assuming a given semantic definition of the source language. These two approaches to program quality assurance are often seen as alternatives, and offer contrasting attributes:

- Whereas program testing is a dynamic technique that uses the executable code of the program, program verification is a static technique, which operates on the source code.
- Whereas program testing is based on the assumption that the testing environment is a faithful imitation of the user's operating environment, program verification is based on the assumption that the compiler and the run time environment of the program are compatible with the semantic definition of the programming language.
- Whereas program testing can be used to find faults (if a test fails) but cannot be used to prove the absence of faults (it is virtually impossible in general to test the program on all possible test data), program verification can be used to prove the absence of faults (under the assumptions cited above) but cannot necessarily be used to prove their presence (if a proof fails, we have no easy way to tell whether it is because the program is incorrect or because the proof was not well planned).
- Whereas testing can be applied to programs of any size and complexity, program proving can only be applied to very small programs, which use only simple constructs.
- Whereas program proving concludes with a logical claim about the correctness of the program, program testing concludes either with a report of a failure or some statistical claim about the reliability of the program.

In practice, program verification has been the subject of much research, but has made little inroads into industrial practice; by contrast, software testing has been common practice in industry, but in circumstances where it is difficult to make any credible claims of software quality. In this paper, we argue in favor of a hybrid approach, whereby testing and proving are used in concert, each being deployed where it works best. Specifically, we envision employing our hybrid proving/ testing approach in the context of verifying source code implementing abstract data types (or other systems that maintain an internal state) specified axiomatically, by means of axioms and rules. Whereas verification techniques are used to verify the source code against the axioms, testing is used to check the source code against oracles defined by the rules; also, the testing phase follows the discipline of cleanroom software engineering [1], and concludes with an empirical estimate of the reliability of the software product, measured as a mean time to failure.

2. Testing Abstract Data Types against rules

We represent the specification of abstract data types by means of three attributes:

- An input space, say X that represents the symbols that may be fed into the ADT as inputs. For a stack ADT, for example, these would be:
  X = {init, pop, top, size, empty} ∪ {push} × itemtype.
  From this set, we build set H of sequences of elements of X, and we refer to H as the set of input histories, or input sequences.
- An output space, which represents the set of symbols that the ADT returns on output. For the stack ADT, this would be:
  Y = itemtype ∪ integer ∪ Boolean ∪ {error}.
- A relation (often a function) from H to Y, which associates an output for each input sequence. For the stack ADT, this relation would include pairs such as:
  Stack (init.push(a).push(b).top.push(c).pop. top) = b.

This specification model resembles trace specification, described in [2], though they were developed independently. In order to represent specifications such as these in closed form, we use an axiomatic notation that includes:

- Axioms, which represent the behavior of the ADT for elementary input sequences. As an example, consider the following axiom for the stack specification:
  1. Top axioms.
     a. stack(init.top) = error.
     b. stack(init.h.push(a).top) = a.
  2. Size axiom.
     a. stack(init.size) = 0.
  3. Empty axioms.
     a. stack(init.empty)=true.
     b. stack(init.push(a).empty)=false.
- Rules, which define the behavior of the ADT for complex input sequences as a function of their behavior for simpler input sequences. As an example, consider the following rule for the stack specification:
  1. Init rule:
     stack(h.init.h')=stack(init.h').
     Init reinitializes the stack state, even if there a history h prior to init or not.
  2. Init Pop rule:
     stack(init.pop.h) = stack(init.h).
     stack(init.h.push(a).pop.h+)= stack(init.h.h+).

  A pop operation cancels the push before it.
  3. Size rule:

  stack(init.h.push(a).size)=1+stack(init.h.size)
  Push operation raises the size of the stack by 1 because the stack size is not restricted.
  4. Empty rules
     a. stack(init.h.push(a).h'.empty)═> stack(init.h.h'.empty).
     If, despite having operation push(a) in its history, the stack is empty, then a fortiori it would empty without push(a).
     b. stack(init.h.empty)═> stack(init.h.pop.empty).
     If the stack is empty, then a fortiori it would be empty if an extra pop operation was performed in its past history.
  5. V-operation rules
     a. stack(init.h.top.h+)=stack(init.h.h+).
     b. stack(init.h.size.h+)=stack(init.h.h+).
     c. stack(init.h.empty.h+)= stack(init.h.h+).

V-operations have no impact on the future behavior of the stack.

Where h is an arbitrary input history and h+ is an arbitrary non null input history.

We suppose that we have a stack implementation in the form of a class in an object oriented language, such as C++ or Java, and we suppose that the class implementation has a method for each input symbol of the specification. In order to check that our implementation satisfies the rule indicated above, we proceed as follows [3]:

- In order to raise our level of confidence in the correctness of the implementation, we resolve to test the implementation a large number of times, using a random data generator for the test data.
- The main goal of the test data generator is to generate input histories h and h+ at random.
- The way to check that the stack is in the same state after executing init.h.push(a).pop.h+ and after executing init.h.h+ is to check that the two input histories lead to the same internal state; since we do not want to have to review what is the internal state, we equate that condition with the condition that all the function that report on

the state (namely top, size, and empty) return the same value, in other words:

```
init.h.push(a).pop.h+.top = init.h.h+.top
init.h.push(a).pop.h+.size = init.h.h+.size
init.h.push(a).pop.h+.empty = init.h.h+.empty
```

To check this, we first check the test driver which looks as follow:

```
{
int nbf=0;
for (int i=0; i<testsize; i++)
   {
switch (i%9)
case 0: initrule();           case 1: initpoprule();
case 2: pushpoprule();        case 3: sizerule();
case 4: emptyrulea();         case 6: vopruletop();
case 5: emptyruleb();         case 8: vopruleempty();
case 7: voprulesize();
   }
cout<<"failure rate:" <<nbf<<"out of
"<<testsize<<endl; }
```

3. Goal Oriented Testing

Once we are confident that our test driver is working properly, i.e. generates the right random data and tests the right condition for correctness, we can use it to estimate the mean time to failure of the software product by following the cleanroom process for reliability estimation [4]. This process advocates to executing the test driver until a test fails; when that happens, we stop the experiment, diagnose the fault, remove it, and then we continue with the test until the next failure. At the conclusion of this process, we are able to deliver the software product with a guaranteed MTTF estimate, based on empirical observations of how the product behaved under test.

As an illustration, consider the following table of test runs

Table1. Test runs

Number of faults removed	Number of executions without failure
0	12
1	40
2	38
3	90
4	500
5	220
6	2300

Our estimate of the MTTF at the completion of this test is:

$$MTTF_N = MTTF_0 \times R^N \tag{1}$$

Where

MTTF: Mean Time to Failure, is the mean of the random variable that measures the number of executions before the next system failure,

$MTTF_0$: is the mean time to failure of the software product at the beginning of the testing phase,

$MTTF_N$: is the mean time to failure after N faults have been removed,

R: is the reliability growth factor, which reflects by what multiplicative factor the MTTF grows, on average, after each fault is removed.

By applying linear regression to the logarithmic to the equation below

$$Log(MTTF_N) = Log(MTTF_0) + N \times Log(R) \tag{2}$$

Where

$Log(MTTF_N)$: Dependent variable

N: independent variable.

The calculations in the table blew include the logarithm of the Number of executions without failure in the third column, $\bar{N}$ Is the average of Number of faults removed and $\bar{y}$ is average of logarithm of the Number of executions without failure.

Table2. The Calculations of MTTF

Number of faults removed (N)	Number of executions without failure	log (y)	$Ni - \bar{N}$	$(Ni - \bar{N})^2$	$(Yi - \bar{Y})^2$
0	12	1.08	-3	9	3.03
1	40	1.6	-2	4	0.98
2	38	1.58	-1	1	0.51
3	90	1.95	0	0	0.00
4	500	2.7	1	1	0.61
5	220	2.34	2	4	0.50
6	2300	3.36	3	9	3.81
$\bar{N}$ = 3		$\bar{y}$= 2.09		∑= 28	∑ = 9.44

4. Related Work

An approach for testing Java implementations of abstract data types against property-driven algebraic specifications presented in [5], by determining at run-time if the analyzed class behaves as required by the specification. This done by reducing the compatibility checking problem to the run-time monitoring of contract annotated classes, which is supported by several runtime assertions-checking tools. The automatic generation of monitorable contracts and a refinement language supported by using a simple language to convert specifications into Java types and technique to generate monitorable classes, allowing for a simple and effective runtime monitoring and supports developers to use formal specifications. However, this approach is dependent on using the clone method quality (method is used to create a copy of an object of a class) which is supplied by the user.

[6] Presents a testing approach for modules using an algebraic specification as a set of executable rewrite rules. The user provides the specification, an implementation class, and an explicit mapping from concrete data structures of the implementation instance variables to abstract values of the specification. The union of the automatically generated direct implementation; and the implementation given by the implementer, with additional code to check their agreement; together are used to build a self-checking implementation. The direct implementation is generated from the specification by representing instances of abstract data types as terms, and manipulating them according to the rewrite rules defined in the ADT specification. However, the user must write the representation mapping, or abstraction mapping in the same language as the implementation class, and asks user knowledge about internal representation details. And there are some axioms that are not accepted by this approach; for example, equations like insert(X,insert(Y,Z))=insert(Y,insert(X,Z)) cannot be accepted as rewrite rules because they can be applied infinitely often.

AutomatedQA Testcomplete9 from SmartBear [7], is a functional and unit testing tool that used to handle a wide range of application and technologies such as Visual Basic, Delphi, C++Builder, .NET, WPF, Visual C++, Java and Web applications. Further, it can start the test with different dataset for specific item to be tested that can be generated by using TestComplete table variables which can be created with any number of columns or stored data excel file format. After recording the user steps, the test is executed against these steps, Testcomplete9 analyses the test that displayed in Test Log and posts messages about results of each test operation, with successful or failed according to Test Log with image associated to each result.

Test Studio by Telerik [8], is a functional, load, performance and mobile applications testing tool for ASP.NET AJAX, PHP, Silverlight, and MVC technologies. Test data can be created locally and associated with a given test, or add external data source in format of Excel spread sheet, XML file, Database source and CSV file , further more Test Studio can run test inside visual studio application.

The first step in test is recording the user steps (in web test or WPF test) and then run the test according to these steps. After that, Test Studio view test results with pass or fail and also can report the result in graph form.

HP Unified Functional Testing (UFT) [9], is a functional and regression testing tool for GUI and API testing. The UFT record the user action on specific application or web site and can use data table for adding data for test. After running the test, UFT provides test steps view and executive summary report associated with each step, further it gives the statistics about the previous run and current run in the form of pie charts with pass or fail notification.

Ranorex is a graphical user interface (GUI) testing tools for web, desktop, and mobile technologies [10]. It supports Win32, WPF, and Desktop applications, in addition to Flex, HTML, Flash, AJAX, and Silverlight web applications. The first step in Ranorex test is creating new test suite, and then start record user steps in specific application or browser or start mobile recording. The recorder constructs step for each action or step performed in the application or browser. After running the test, Ranorex simulate the user actions or steps which were recorded. The last step in the test is executing it with report that shows the test is successful or not with mechanism to face the errors which make the test fail. Ranorex provides test with internal or external data source as Excel files, CSV files and SQL databases.

5. Automation Plans

We envision developing an automated tool that supports the test of class implementations with respect to abstract data type specifications using the pattern discussed in this paper. This tool proceeds in four steps:

- Oracle specification. We use rules to generate oracles, so that the test checks that the class implementation satisfies the proposed rule.
- Test Data Generation. Test data is generated randomly, by instantiating the various occurrences of arbitrary input histories (represented by h, h', h'', h+, etc) in the rules.
- Test Driver Design. The test driver that we adopt is an adaptation of the basic design shown in section II; we merely change the number of rules and how they are called.
- Test Execution. The test driver queries the user about the number of random tests she/he wants to run, as well as the type of error report she/he wants to get, and it produces a test report accordingly.

The main source of difficulty in this project is the wide diversity in the structure of rules, their wide variety of parameters, and their different forms. This matter is currently under investigation.

6. Conclusions

Our main focus in this paper is employing our testing approach in the context of verifying source code implementing abstract data types specified axiomatically, by means of axioms and rules, concludes with measuring of a mean time to failure as an empirical estimate of the reliability of the software product.

Acknowledgements

I would like to express my deep gratitude to my supervisor Prof. Ali Mili For his persistent constant guidance and wise counsel.

References

[1] S. A. Becker and J. A. Whittaker, Cleanroom software engineering practices. Igi Global, 1997.

[2] D. Hoffman and R. Snodgrass, “Trace specifications: Methodology and models,” Software Engineering, IEEE Transactions on, vol. 14, no. 9, pp. 1243–1252, 1988.

[3] F. Tchier and A. Mili, “On the verification and validation of software modules: Applications in teaching and practice,” Tech. Rep. of NJIT, 2013.

[4] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, Cleanroom software engineering: technology and process. Addison-Wesley Professional, 1999.

[5] I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. Reis, “Testing implementations of algebraic specifications with design-by-contract tools,” Department of Informatics, University of Lisbon, 2005.

[6] S. Antoy and D. Hamlet, “Automatically Checking an Implementation against Its Formal Specification Self-checking ADTs,” Software Engineering, IEEE Transactions on, vol. 26, no. 1, pp. 55–69, 2000

[7] Getting Started With TestComplete9, SmartBear Software, 2013.

[8] Test Studio Standalone & Visual Studio Plug-In, Quick-Start Guide, Telerik Corp, 2012.

[9] HP Unified Functional Testing, Tutorial, HP, 2012.

[10] Ranorex Tutorial, Test Automation Guide, Ranorex, 2013.