Test Driven Development with Oracles and Formal Specifications

Shadi Alawneh and Dennis Peters

Faculty of Engineering and Applied Science
Memorial University, St.John's, NL
Canada A1B 3X5
{shadi.alawneh,dpeters}@mun.ca

Abstract. The current industry trend to using Test Driven Development (TDD) is a recognition of the high value of creating executable tests as part of the development process. In TDD, the test code is a formal documentation of the required behaviour of the component or system being developed, but this documentation is necessarily incomplete and often over-specific. An alternative view to TDD is to develop the specification of the required behaviour in a formal notation as a part of the TDD process and to generate test oracles from that specification. In this paper we present tools in support of this approach that allow formal specifications to be written in a readable manner that is tightly integrated with the code through an integrated development environment, and test oracles to be generated automatically. The generated test code integrates smoothly with test frameworks (e.g., JUnit) and so can be directly used in TDD. This approach has the advantage that the specifications can be complete and appropriately abstract but still support TDD.

Keywords: Test Driven Development, Extreme Programming, Open Mathematical Documents, Test Oracle

1 Introduction

TDD is one of the core practices of Extreme Programming (XP). Two key principles of TDD are 1) that no implementation code is written without first having a test case that fails with the current implementation, and 2) that we stop writing the implementation as soon as all of the existing test cases pass.

In TDD, the test code is a formal documentation of some of the required behaviour of the system being developed. However, tests alone describe the properties of a program only in terms of examples and thus are not sufficient to completely describe the behaviour of a program. So, this documentation is necessarily incomplete and often over-specific. To solve this problem we propose an alternative approach to TDD, which is to develop a formal specification of the required behaviour as part of the TDD process and then generate test oracles from that specification. We thus propose a variation on the key TDD principles

listed above: 1) No implementation code is written without first having a specification for the behaviour that is not satisfied by the current implementation, and 2) we stop writing the implementation as soon as the implementation satisfies the current specification. By generating oracles directly from the specification we are able to quickly and accurately check if the specification is satisfied by the implementation for the selected test cases.

We are using OMDoc (Open Mathematical Documents) [3] as a standardized storage and communications format for our specifications, and so we can take advantage of other tools.

2 Formal Software Specifications

Formal specifications are documentation methods that use a mathematical description of the software or hardware behaviour, which may be used to develop an implementation. With reference to the set of documents described in [6], in this work, we are focused on using module internal design documents [7] or module interface specifications to drive the development [8]. These two types of documents specify the behaviour of the module either in terms of the internal data structure and the effect of each access program on it, or in terms of the externally observable behaviour of the module.

There are several different formal methods for program specifications, e.g. the Java Modeling Language (JML) [4] and Z [9]. In our work, we use relations for the specifications, which characterize the acceptable set of outcomes for a given input. The specifications in our work consists of program specifications, which, in OMDoc terms, are symbol definitions contained within theories. Also, each symbol has a type and possibly other information. A program specification, describes the required behaviour of a program either in terms of the internal data structure and the effect of each access program on it, or in terms of the externally observable behaviour of the module. It consists of these components: constants; variables; auxiliary function and predicate definitions; the program invocation, which gives the name and type of the program and lists all its actual argument program variables; and an expression that gives the semantics of the program.

3 Oracle Generation

As described in [2], an oracle is some method for checking whether the system under test has behaved correctly on a particular execution.

In our work, an oracle is a program which, given a test input and output, will determine if it passes or fails with respect to the specification from which the oracle was derived. The oracle evaluates the characteristic predicate of the specification relation—if it evaluates to **true**, then that test input and output passes, otherwise it fails. Note that such an oracle does not require the existence of a correct version of the program.

Our tool can generate test oracles from both scalar expressions (logical operators, primitive relations, quantifications), and tabular expressions. Moreover, it

can handle auxiliary functions and predicates. It has been shown in the previous work that the tabular representation of relations and functions is a significant factor in making the documentation more readable, and so we have customized our tool to support them.

The oracle in our approach consists of two kinds of code: that generated by the Test Oracle Generator (TOG), and classes (e.g., Integer_Interval.Java, InvertedTable.Java, NormalTable.Java and VectorTable.Java), previously manually implemented that are used by the TOG generated code.

In the example described in Section 4.1, we are using one kind of tabular expression (Vector). Tabular expressions are implemented by instantiating an object of one of several classes of (Java) table objects which implement the various types of tabular expressions (normal, inverted and vector). These table objects contain all knowledge of the semantics of tabular expressions, so there is no need for this knowledge to be in the TOG. The expression in each cell of the table is implemented as Java class that extends CellBase and contains a procedure that evaluates that expression. This approach for implementing tabular expressions has the advantage that the oracle code can be more organized.

Table objects have a method, evaluateTable, which evaluates the tabular expression.

4 Test Driven Development with Oracles

The work reported in this paper is an extension of the work reported in [1]. In [1], the test oracle generator supports generating test oracles from methods but in this work we extended the test oracle generator to allow the users to generate test oracles from module (class) specifications, which are based on the externally observable behaviour of the class. This will allow the use of oracles in class testing. Also, we have introduced an alternative approach to TDD that is to develop the specification of the required behaviour in a formal notation as a part of the TDD process and to generate test oracles from that specification.

The process looks like this:

- Write the specification for some required behaviour.
- Generate the test oracle from the specification and select test inputs.
- Run the program under test in the test framework (e.g., JUnit) using the test oracle to verify if it passes or fails.
- If the test fails, write code until this test passes.
- If the test passes and the specification is not completed yet, add to or refine the specification and redo the process again.
- We keep doing this process until the specification is complete.

4.1 Example

As an illustrative example we use the bounded stack as developed in [5]. The first step in our approach is to specify some required behaviour, in this case for creation of an empty stack:

Data Structure
Integer s[ ]
Integer maxSize
Integer length

Program Functions
Stack stack(**Integer** x)
$\stackrel{\text{df}}{=}$ (result.isEmpty()$\wedge$
result.maxDepth() = x)

Boolean isEmpty()
$\stackrel{\text{df}}{=}$ result = (length = 0)

Integer maxDepth()
$\stackrel{\text{df}}{=}$ result = maxSize

The specification consists of the data structure description, the definition for stack(x) function, which is the program function specifying the behaviour of the constructor and two program functions specifying the behaviour of the methods isEmpty() and maxDepth().

After we write the specification, we generate the test oracle from it and write the test code to call it (e.g., from JUnit). The test case will, of course, fail, so we should implement the constructor and methods so that the test case passes and we have a program that is consistent with the specified behaviour.

We then modify the specification for push to define behaviour for pushing on a non-full stack, and add three more methods:

void push(**Integer** x)
$\stackrel{\text{df}}{=}$

	p_this.size() $\geq 0 \wedge$ p_this.size() $<$ this.maxDepth()
this.size() =	p_this.size() + 1
this|	$\forall i : [0, \text{p\_this.size}() - 1].($ this.elementAt(i) = p_this.elementAt(i))$\wedge$ (this.lastElement() = x)

Integer elementAt(**Integer** i)
$\stackrel{\text{df}}{=}$ result = s[i]

Integer size()
$\stackrel{\text{df}}{=}$ result = length

Integer lastElement()
$\stackrel{\text{df}}{=}$ result = s[length $-$ 1]

Here we use the naming convention of prepending "p_" to a program variable name (e.g., p_this) to represent the value of the program variable (e.g., this) in the state immediately before the function was executed. The new behaviour defined by the specification is to push an object on an a non-full stack. After the push the stack should contain that element and the size for the stack after is increased by one. Again we generate the test oracle and implement a test

case, which will initially fail. The stack code is then developed until the test case passes, and so it implements the specified behaviour.

Again we generate the test oracle and implement test cases, this time to push a few elements onto the stack.

Continuing in this manner, we eventually reach the full specification of the bounded stack, as below, and we have at the same time developed a full implementation and a full suite of test cases.

Data Structure

Integer s[ ]
Integer maxSize
Integer length

Program Functions

Stack stack(**Integer** x)
$\stackrel{\text{df}}{=}$ (result.isEmpty() $\wedge$ result.maxDepth() = x)

void push(**Integer** x)
$\stackrel{\text{df}}{=}$

	p_this.size() $\geq 0 \wedge$ p_this.size() $<$ this.maxDepth()	p_this.size() = this.maxDepth()
this.size() =	p_this.size() + 1	p_this.size()
this|	$\forall i : [0, \text{p\_this.size}() - 1].$ $\binom{\text{this.elementAt}(i) =}{\text{p\_this.elementAt}(i)} \wedge$ (this.lastElement() = x)	this = p_this

Integer pop()
$\stackrel{\text{df}}{=}$

	p_this.size() ≥ 1
this.size() =	p_this.size() $-$ 1
this|	$\forall i : [0, \text{this.size}() - 1]. \binom{\text{this.elementAt}(i) =}{\text{p\_this.elementAt}(i)} \wedge$ (result = p_this.lastElement())

Integer top()
$\stackrel{\text{df}}{=}$ result = this.lastElement()

Boolean isEmpty()
$\stackrel{\text{df}}{=}$ result = (length = 0)

Integer maxDepth()
$\stackrel{\text{df}}{=}$ result = maxSize

Integer size()
$\stackrel{\text{df}}{=}$ result = length

Integer lastElement()
$\stackrel{\text{df}}{=}$ result = s[length $-$ 1]

Integer elementAt(**Integer** i)
$\stackrel{\text{df}}{=}$ result = s[i]

5 Conclusions

In test driven development, tests are used to specify the behaviour of the program, and the tests are additionally used as a documentation of the program. However, tests are not sufficient to completely define the behaviour of a program because they only define the program behaviour by example and do not allow us to state general properties. So, the later can be achieved by using our TDD approach, which uses a formal specification to specify the behaviour of the program and supports testing directly against that specification by generating oracles.

Clearly a next step in this research and tool development will be to support test case generation from the specification as well, which will further reduce the amount of 'manual' test code development effort.

6 ACKNOWLEDGMENTS

This research was supported by the Faculty of Engineering and Applied Science at Memorial University and the Government of Canada through the Natural Sciences and Engineering Research Council (NSERC).

References

1. S. Alawneh and D. Peters. Specification-based test oracles with junit. Proc. IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'10), Calgary, Canada, May. 2010.
2. W. E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill Book Company, 1987.
3. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2).* Number 4180 in Lecture Notes in Artificial Intelligence. Springer Verlag, 2006.
4. G. T. Leavens, A. L. Baker, and C. Ruby. JML: a notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.
5. J. Newkirk and A. Vorontsov. *Test-Driven Development in Microsoft .NET.* Microsoft Press, 2004.
6. D. L. Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, Oct. 1995.
7. D. L. Parnas, J. Madey, and M. Iglewski. Precise documentation of well-structured programs. *IEEE Trans. Software Engineering*, 20(12):948–976, Dec. 1994.
8. C. Quinn, S. Vilkomir, D. Parnas, and S. Kostic. Specification of software component requirements using the trace function method. In *Int'l Conf. on Software Engineering Advances*, page 50, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
9. J. M. Spivey. *The Z Notation: A Reference Manual.* International series in computer science. Prentice Hall, New York, 2nd edition, 1992.