SPECIFICATION-BASED TEST ORACLES WITH JUNIT

Shadi G.Alawneh, Dennis K. Peters

Faculty of Engineering and Applied Science
Memorial University of Newfoundland
St. Johns NL A1B 3X5

ABSTRACT

Software testing is an important step to help ensure that the software is behaving correctly. An important component of the test process is a test oracle, which determines if the software behavior is correct or not. In this paper, we present tools that enhance an integrated development environment to give the user the ability to write the formal specifications in a readable manner and generate test oracles automatically. The generated test oracles integrate smoothly with test frameworks (e.g., JUnit) and hence they can be directly used to test the behavior of the program. This approach for testing has the advantage that the quality of testing can be high and very efficient.

Index Terms— Open Mathematical Documents, Test-Driven Development, Test Oracle, Automated testing.

1. INTRODUCTION

Testing is a costly process that, if not automated, requires high concentration from the tester in order to assess the system behavior. Unfortunately, non-automated testing usually results in less than accurate information about the correctness of the software system. One reason for this is the unavailability of an effective means for deciding whether the system has behaved correctly on test executions. The system is executed for some test data, but usually the result is left for someone to visually check and predict whether the system has behaved correctly. Thus, the actual knowledge about the system behavior may be missed if this is done heedlessly or rashly.

This problem can be solved by using automated test oracles in the testing process. As discussed in [1], an oracle is some method for checking whether the system under test has behaved correctly on a particular execution. We are developing an approach to derive test oracles from formal specifications and associating them into the testing process. Our approach to deriving and using specification-based test oracles provides an integrated development environment which gives the user the ability to write formal specifications in a readable manner and to generate test oracles automatically. The generated test oracles integrate smoothly with test frameworks (e.g., JUnit), and hence can be directly used to test the behavior of the program.

2. FORMAL SOFTWARE SPECIFICATIONS

Formal specifications are documentation methods that use precisely defined notations, which are usually mathematically based, to define the software or hardware behavior. These specifications may be used to develop an implementation and to drive automated testing, as is discussed in this paper. The emphasis in the specification is on *what* the system should do, not necessarily *how* the system should do it. Examples of such languages (or notations) that are used to define formal specifications are VDM, Z, and B.

Formal specifications have several advantages over more traditional (informal) techniques:

- Since they are precisely defined, there is little room for misinterpretation of the intended meaning. This is in stark contrast to natural language and other informal techniques, which leave lots of room for (mis)interpretation.
- Formal specifications are a kind of mathematical entity, so they may be analyzed and studied using mathematical tools and methods.
- They can be processed automatically, so they can be used as an exchange medium between software tools.

For automated testing some form of formal specification of the required behaviour is essential. In a traditional automated testing process, this specification is in the form of the testing code, which will implement comparisons or tests to determine if the actual behaviour is acceptable. In this work we propose that the specification be expressed in a mathematical notation and that specification can be used to automatically generate testing code.

With reference to the set of documents described in [2], in this work, we are focused on deriving test oracles from the module internal design document [3]. This type of document describes the module's data structure, states the intended interpretation of that data structure (in terms of the external interface), and specifies the effect of each access-program on the module's data structure.

Computer system behavior is often such that the system must react to changes in its environment and behave differently under different circumstances. The result is that the mathematics describing this behavior consists of a large number of conditions and cases. It has been long established that tables can be used to help in the effective presentation of such mathematics [4, 5, 6]. 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 tools to support them.

A complete discussion of tabular expressions is beyond the scope of this paper, hence interested readers are referred to the cited publications [5, 6]. In their most basic form, tabular expressions represent conditional expressions. For example, the function definition in equation (1), could be represented by the tabular expression in equation (2).

The tabular form of the expressions is not only easier to read, but also easier to write correctly. Tabular expressions make it very clear what the cases are, and all that cases are considered.

$$f(x,y) \stackrel{\text{df}}{=} \begin{cases} x+y & \text{if } x>1 \wedge y<0 \\ x-y & \text{if } x\leq 1 \wedge y<0 \\ x & \text{if } x>1 \wedge y=0 \\ xy & \text{if } x\leq 1 \wedge y=0 \\ y & \text{if } x>1 \wedge y>0 \\ x/y & \text{if } x\leq 1 \wedge y>0 \end{cases} \quad (1)$$

$f(x,y) \stackrel{\text{df}}{=}$ (2)

	$x>1$	$x\leq 1$
$y<0$	$x+y$	$x-y$
$y=0$	x	xy
$y>0$	y	x/y

2.1. Program Specification

A program specification in our work, consists of these components: the program invocation gives the name and type of the program and lists all its actual argument program variables; an expression that gives the semantics of the program; constants, variables, auxiliary function and predicate definitions. The following explains these in more detail.

2.1.1. Constants

A constant is a special kind of variable whose value cannot be altered during program execution. Many programming languages make an explicit syntactic distinction between constant and variable symbols. For example, in Java the following are constants: 10 and "Any Text".

2.1.2. Variables

A *program variable* is a way of referring to a memory location used in a computer program. This memory location holds a value—perhaps a number or text or more complicated data type—and this value will change as the program executes.

In a specification, as in mathematics, variables represent values: the value of program variables in either the initial state or final state of an execution, the value of expressions passed as arguments in auxiliary definitions, or the value of quantification indices. Variables which represent quantification indices are considered to represent a value only where they are bound.

All variables must have a type and should be defined in the documentation.

2.1.3. Auxiliary Function And Predicate Definitions

The definition of an auxiliary function consists of a name, a type, a list of argument variables and an expression that defines the semantics of the auxiliary function. Also, the definition of the auxiliary predicate is the same but the expression is a predicate expression which is described in Section 2.1.4.

2.1.4. Predicate Expressions

A predicate expression is an expression that evaluates to **true** or **false** and consists of either quantified expression as described below, or a string of the form $G \wedge H$, $G \vee H$, $H \Rightarrow G$ or $\neg G$, where G and H represents predicate expressions.

2.1.5. Quantified Expressions

In our test oracle generator, quantification ($\forall$ — for all, and $\exists$ — there exists) must be restricted to a finite set, which can be implemented as a java collection so that it can be automatically generated. This is done by permitting only the following forms of quantified expressions $(\forall i : G(i).H(i))$ and $(\exists i : G(i).H(i))$, where i is a variable, known as the index variable of the quantification, $G(i)$ is a collection and $H(i)$ is any predicate expression of a permitted form.

2.2. Sample Program Specification

Figure 1, specifies a program '`ggcd`' which compares an integer value '`i`' with another integer value '`j`', returns the greatest common divisor of them if '$\texttt{i} > 0 \wedge \texttt{j} > 0$', otherwise returns 0. Additionally, it indicates if the two integers are positive by using the returned value, which is represented by a boolean variable '`result`'. Note that the auxiliary function '$\gcd$' is a recursive function and so it will be used repeatedly until the greatest common divisor is found.

3. TOOL SUPPORT

3.1. OMDoc Document Model

As described in [7], the OMDoc (Open Mathematical Documents) format is a content markup scheme for (collections of)

Program Specification

Boolean
ggcd(**Integer** i, **Integer** j, **Integer** gcdvalue)

	$i > 0 \wedge j > 0$	$i \leq 0 \vee j \leq 0$
gcdvalue =	$\gcd(i, j)$	0
result =	TRUE	FALSE

Auxiliary Function Definitions

Integer $\gcd(\textbf{Integer}\ a, \textbf{Integer}\ b)$
$\stackrel{df}{=}$

$b \neq 0$	$\gcd(b, a\%b)$
$b = 0$	a

Fig. 1. Ggcd Program Specification

mathematical documents including articles, textbooks, interactive books, and courses. OMDoc also serves as the content language for the communication of mathematical software. 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. Consequently, this leads us to propose our specification model which consists of these OMDoc elements:

Theory : a theory is a self-contained part of a specification. It could, for example, represent a requirements specification, a module interface specification, a module internal design document or a single program function. A theory contains zero or more sections of each of the following kind.

Symbol : a symbol is a basic component of a specification: a variable, function, relation or constant. All symbols that are used in a specification must be defined somewhere, either by being declared to be a bound variable, defined in the specification itself, defined (globally) in an imported theory, or from a standard set (e.g., standard OpenMath content dictionary). A symbol has the following attributes:

- **Name** : for referring to the symbol (required).
- **TTS Role** : indicates how this symbol is used as part of a specification (optional).

Type : all symbols should have a type supplied.

Definition : a definition contains an expression that gives the semantics of a symbol.

Presentation : a presentation contains the format for mathematical symbol. A presentation element has **for** attribute which identifies the symbol represented. Each presentation contains one or more **use** elements.

Use : indicates how the symbol represented in a specific language. A use element has the following attributes:

- **Format** : specify the name of the language that the symbol represented in.
- **Fixity** : determines the placement of the symbol. This attribute can be one of the keywords **prefix**, **infix**, and **postfix**. For **prefix** it is placed in front of the arguments. For **infix** it is placed between the arguments. Finally, for **postfix** it is placed behind the arguments.
- **Separator** : this specifies the separator in the argument list.
- **lbrack/rbrack** : these two attributes handle the brackets to be used in presentation.

Code : is unparsed formal text and it is not needed in our documents but in some documents it is needed.

Text : is unparsed informal text and it is important for readability of the document.

3.2. The Eclipse Framework

Eclipse is a software platform comprising extensible application frameworks, tools and a runtime library for software development and management. It is written primarily in Java to provide software developers and administrators an integrated development environment (IDE).“Eclipse employs plug-ins in order to provide all of its functionality on top of (and including) the runtime system, in contrast to some other applications where functionality is typically hard coded”.[8] Using this framework to develop our tool provides significant advantages over developing a stand-alone tool including its widespread use in the user community, its facilities for tight integration of documents with other software artifacts, and provision of support for software development tasks.

3.3. Specification Editor

As a part of our tools, we are developing a specification editor to support production of software documents, which is illustrated in Figure 2. This Editor provides a “multi-page editor” (which provides different views of the same source file) for “.tts” files, which are OMDoc files. One page of the editor is a structured view of the document, another one shows the raw XML representation, and another gives a detailed view of the document giving the user the ability to view and edit the mathematical expressions. The support libraries in Eclipse provide techniques to ensure that the views of the document are consistent. This editor is built using several open source libraries in including the RIACA OpenMath Library.

This editor is seen as a primary means for the human users to interact with specification documents.

4. TEST ORACLES WITH JUNIT

Our approach for testing the behavior of the program consists of these steps:

- Write a complete specification of the required behavior for the program in a formal notation.
- Generate test oracle from the specification.
- Run the program under test in the test framework (e.g., JUnit) using the test oracle to verify if it passes or fails.

A good example to illustrate our testing approach is provided in [9, 10]—converting integers into their roman number equivalent. We selected this example because it shows the TDD process which is the next step for our work.

The following uses this example to show the whole process for the testing. According to our approach, the first step is to write a complete specifications of the required behavior for the program. So, we have written a complete specifications as follows:

String dToR(**Integer** i)
$\stackrel{df}{=}$

	$i \geq 1 \wedge i \leq 3999$	$i < 1 \vee i > 3999$
result =	subDToR(i)	"NA"

String subDToR(**Integer** i)
$\stackrel{df}{=}$

$i \geq 1000$	"M"+ subDToR(i − 1000)
$i \geq 900 \wedge i < 1000$	"CM"+ subDToR(i − 900)
$i \geq 500 \wedge i < 900$	"D"+ subDToR(i − 500)
$i \geq 400 \wedge i < 500$	"CD"+ subDToR(i − 400)
$i \geq 100 \wedge i < 400$	"C"+ subDToR(i − 100)
$i \geq 90 \wedge i < 100$	"XC"+ subDToR(i − 90)
$i \geq 50 \wedge i < 90$	"L"+ subDToR(i − 50)
$i \geq 40 \wedge i < 50$	"XL"+ subDToR(i − 40)
$i \geq 10 \wedge i < 40$	"X"+ subDToR(i − 10)
$i = 9$	"IX"
$i \geq 5 \wedge i < 9$	"V"+ subDToR(i − 5)
$i = 4$	"IV"
$i > 0 \wedge i < 4$	"I"+ subDToR(i − 1)
$i = 0$	""

The previous specifications consist of two parts: the first part is the definition for the function dToR(i) which represents the program function, the second part is the definition for subDToR(i) function which represents an auxiliary function. The required behavior that is represented by that specification is to support the conversion of numbers (1-3999) into their corresponding roman numerals.

After writing the specifications, then we generate the test oracle from it as described in Section 5.

Using the oracle involves implementing test code, as laid out by the test framework, that calls the program under test and then calls the oracle procedures. In our work, we use JUnit because using a framework like JUnit to develop test cases has a number of advantages, not the least of which is that others will be able to understand the test cases and easily write new ones. In addition, it provides a graphical user interface (GUI) which makes it possible to write and test source code quickly and easily. JUnit shows test progress in a bar that is green if testing is going fine and it turns red when a test fails. As a consequence, this is makes it possible for the software developer to easily correct bugs as they are found. The code below shows how to run the oracle generated in the above example with JUnit:

```
package oracles;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;

public class OracleTest extends
junit.framework.TestCase{

  OracleOut o;
  @Before
  public void setUp()
  throws Exception {
    o=new OracleOut();

  }

  @Test
  public void testCon(){
    o.assertdToRTOracle(34,Romans.dToR(34));

  }

}
```

The previous code contains one test case that test the conversion of 34 into it is roman letter (XXXIV). The roman letter is computed by the static method Romans.dToR(int) meant to implement the specification. The user can add any number of test cases. The result for the previous code is shown in Figure 3.

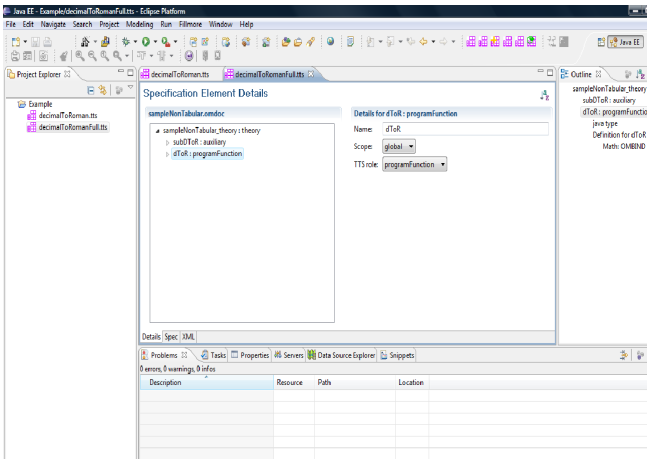

Fig. 2. Screenshot of Editor

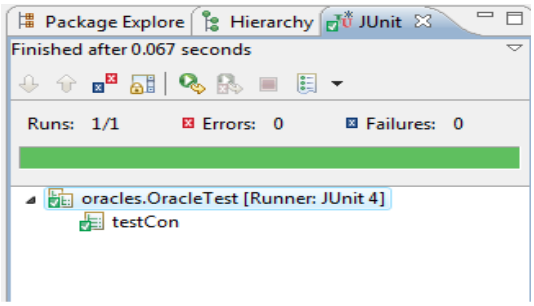

Fig. 3. TestResult

5. ORACLE GENERATION

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 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.

In [11] Peters and Parnas discuss the use of test oracles generated from program documentation. They describe an algorithm that can be used to generate a test oracle from program documentation, and present the results of using a tool based on it to help test parts of a commercial network management application. The results demonstrate that these methods can be effective at detecting errors and greatly increase the speed and accuracy of test evaluation when compared with manual evaluation. The design of test oracle generator they used allows using only C programming language in this prototype. If we need to choose among several programming languages we need to add several additional sub-modules, one for each language.

The work reported in this paper is similar to the work in [11] but our approach for generating test oracles has the following characteristics that make it unique:

- We are using OMDoc as a standardized storage and communications format for our specifications, and so we can take advantage of other tools.
- The semantics of tabular expressions have been generalized to allow more precise definition of a broader range of tabular expression types.
- The test oracle generator is implemented using Java. This makes it easy to integrate with the Eclipse platform.
- The oracle generator has a 'graphical user interface' which is shown in Figure 2. This interface gives the user the ability to select any program specification and generate the oracle from it. This has the advantage of enabling the user to interact easly with the specifications.
- The generated test code integrates smoothly with test frameworks (e.g., JUnit) and hence, it can be directly used to test the behavior 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.

The oracle in our approach consists of two kinds of code: that generated by the Test Oracle Generator (TOG), in OracleOut.java (this represents the root class for the oracle and contains the method which can be used to access the oracle), and the other kind of object classes, in Integer_Interval.Java, InvertedTable.Java, NormalTable.Java and VectorTable.Java, which are not generated by the TOG but are used by the TOG generated code.

The code below shows the implementation of the root class for the oracle (OracleOut.java) for the example that described in Section 4.

```
package oracles;
import ca.fillmoresoftware.plugin.
OracleUtilities.*;
import static org.junit.Assert.*;

public class OracleOut{

  private VarMap vars;
  private OutdToR1 t0;

  public OracleOut(){

    vars=new VarMap();
    t0=new OutdToR1(vars);
  }
```

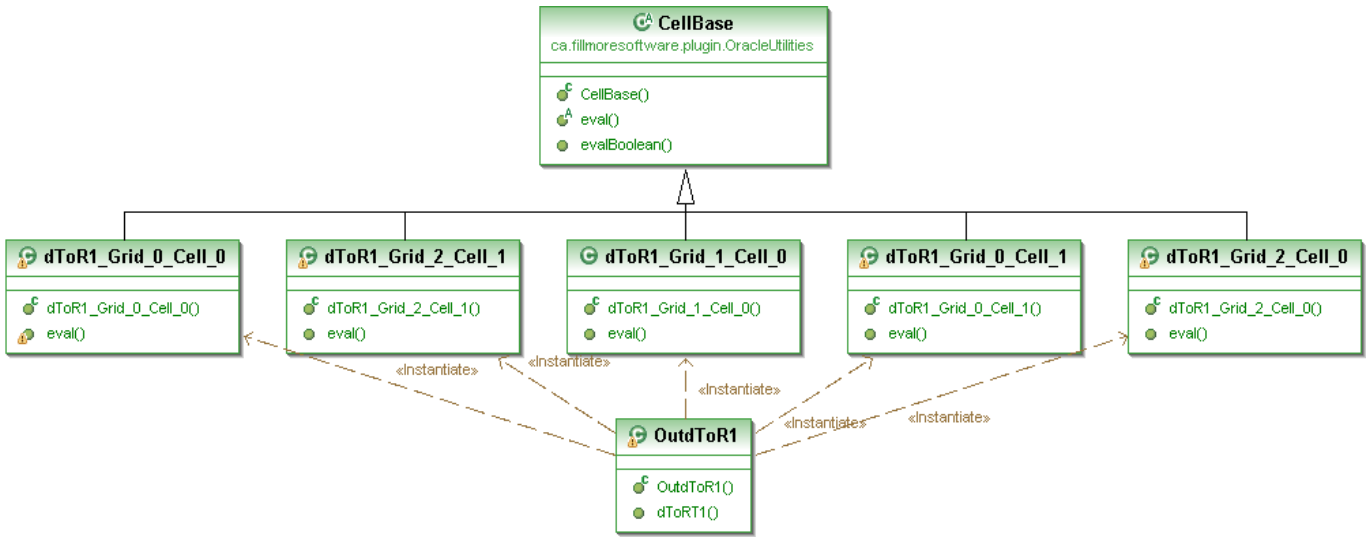

Fig. 4. Oracle Design of dToR Tabular Expression

```
  private Boolean dToRTOracle(
      Integer i,String result){

    Boolean resultOracle;
    vars.setValue("i",i);
    vars.setValue("result",result);
    resultOracle=t0.dToRT1();

    return resultOracle;
  }

  public void assertdToRTOracle(
      Integer i,String result){

    assertTrue(dToRTOracle(i,result));
  }
}
```

In the example described in Section 4, we are using two kinds of tabular expressions (Vector and Normal). 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 (abstract class) 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.

The expression "$\texttt{i} \geq 1 \wedge \texttt{i} \leq 3999$", which is in the first cell of the column header of the dToR table, is implemented as follows.

```
package oracles;
import ca.fillmoresoftware.plugin.
OracleUtilities.*;

public class dToR1_Grid_2_Cell_0
extends CellBase{

  private VarMap vars;

  public dToR1_Grid_2_Cell_0(
      VarMap vars){

    this.vars=vars;
  }

  public Object eval(){

    Integer i=(Integer)vars.
      getValue("i");

    return ((i<=3999)&&(i>=1));

  }
}
```

The other cells in each table are implemented in a similar fashion. The oracle design for the dToR tabular expression is illustrated in Figure 4 and the design for the subDToR tabular expression looks similar. Note that, since the oracle code is automatically generated from the specification, the developer should not need to read the oracle code itself and so readability of this code is not a primary concern.

6. CONCLUSIONS

We propose an approach for deriving and using test oracles to test the behavior of the program. By using this approach, we can improve the quality of the software and reduce the errors. Moreover, by extracting test oracles from formal specifications, we are convinced that the oracles check that the behavior is as specified.

Moreover, the actual knowledge about the system behavior can't be missed because the responsibility of checking whether the system has behaved correctly is done automatically by the test oracle and it is not left for someone to visually check whether the system has behaved correctly.

7. FUTURE WORK

A major extension that is planned for the test oracle generator would allow the users to generate test oracles from module (class) specifications, which are based on the externally observable behavior of the class. This will allow the use of oracles in class testing.

In test-driven development (TDD), the test code is a formal documentation of the required behavior of the component or system being developed, but this documentation is necessarily incomplete and often over-specific. An alternative approach to TDD is to write a complete specification of the required behavior in a formal notation and to generate test cases and oracles from that specification. This approach has the advantage that the specifications can be complete and appropriately abstract but still support TDD. So, another enhancement planned is to make our tools support this approach for TDD.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] William E. Howden, *Functional Program Testing and Analysis*, McGraw-Hill Book Company, 1987.

[2] David Lorge Parnas and Jan Madey, "Functional documentation for computer systems," *Science of Computer Programming*, vol. 25, no. 1, pp. 41–61, Oct. 1995.

[3] David Lorge Parnas, Jan Madey, and Michal Iglewski, "Precise documentation of well-structured programs," *IEEE Trans. Software Engineering*, vol. 20, no. 12, pp. 948–976, Dec. 1994.

[4] David Lorge Parnas, "Inspection of safety critical software using function tables," in *Proc. IFIP Congress*. Aug. 1994, vol. I, pp. 270–277, North Holland.

[5] Ruth F. Abraham, "Evaluating generalized tabular expressions in software documentation," M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, Feb. 1997.

[6] Ryszard Janicki and Ridha Khedri, "On a formal semantics of tabular expressions," *Science of Computer Programming*, vol. 39, no. 2–3, pp. 189–213, Mar. 2001.

[7] Michael Kohlhase, *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*, Number 4180 in Lecture Notes in Artificial Intelligence. Springer Verlag, 2006.

[8] Eric Clayberg and Dan Rubel, *Eclipse Plug-ins*, Addison-Wesley, 2008.

[9] Clarke Ching, "A brief introduction to test driven development using microsoft excel and vba," http://www.clarkeching.com/2006/04/test_driven_dev.html.

[10] Dave Nicolette and Karl Scotland, "Manager's introduction to test-driven development," Agile Conference, 2008, http://www.infoq.com/presentations/TDD-Managers-Nicolette-Scotland.

[11] Dennis K. Peters and David Lorge Parnas, "Using test oracles generated from program documentation," *IEEE Trans. Software Engineering*, vol. 24, no. 3, pp. 161–173, Mar. 1998.