International Journal of Software Engineering
and Knowledge Engineering
Vol. 23, No. 3 (2013) 361–385

DOI: 10.1142/S0218194013500113

World Scientific
www.worldscientific.com

USING TEST ORACLES AND FORMAL SPECIFICATIONS WITH TEST-DRIVEN DEVELOPMENT

SHADI G. ALAWNEH* and DENNIS K. PETERS†

Faculty of Engineering and Applied Science
Memorial University, St. John's NL, Canada A1B 3X5
**shadi.alawneh@mun.ca, http://www.engr.mun.ca/∼alawneh*
†*dpeters@mun.ca, http://www.engr.mun.ca/∼dpeters*

Received 27 April 2012
Accepted 10 December 2012

This paper illustrates how Test Oracles and Formal Specifications, with appropriate tool support, can be used with Test-Driven Development (TDD). In TDD, the test code is a formal documentation of the required behavior of the component or system that is being developed, but this documentation is necessarily incomplete and often over-specific. We describe an alternative approach to TDD that is to develop the specification of the required behavior in a formal notation as a part of the TDD process and to generate test oracles from that specification. We present the results of using this approach to develop programs used in a project at the Faculty of Engineering and Applied Science at Memorial University.

Keywords: Test driven development; extreme programming; open mathematical documents; test oracle.

1. Introduction

Based on [7], TDD is an iteration based development approach where the developer writes a test before he writes enough code to fullfill that test. The steps of TDD are illustrated in the UML activity diagram of Fig. 1. TDD is one of the core practices of Extreme Programming (XP)[7, 19]. 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. Although not all developers agree with all of the XP practices, the ideas of TDD have started to gain wide acceptance.

In TDD, the test code is a formal documentation that describes the required behavior for the component or the system that is being developed for the particular test cases included. However, tests alone describe the properties of a program only in terms of examples and thus, they are not sufficient to completely describe the

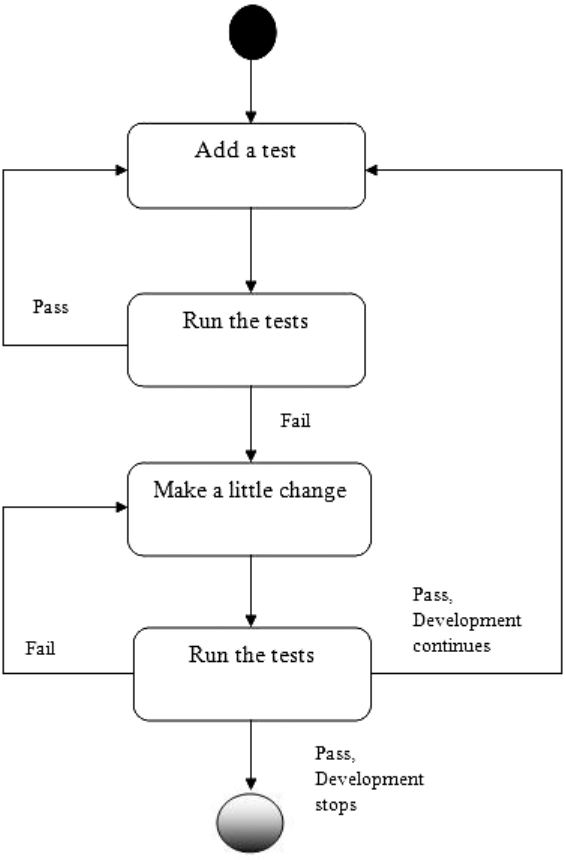

Fig. 1. The steps of test-driven development (TDD) [4].

behavior of a program. Consequently, 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 behavior as a 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 behavior 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. One of the most important goals of this approach is to develop the internal design documents and module interface specifications.

The work reported in this paper is an extension of the work reported in [3, 2]. In [3], we presented our approach but in this paper we evaluted our approach by applying it into real application. In [2], 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 behavior of the class. This will allow the use of oracles in class testing.

In this paper, we applied our approach for TDD to methods and classes which are the basic components for any software application. Our methods are applicable for programs written in different programming languages, but the prototype tools that we have implemented to describe and explain these techniques only work for those written in Java.

Section 2 presents the related work. Formal software specifications are described in Sec. 3. Section 4 presents the tool support that we have used in this work. Section 5 describes the oracle design. Section 6 describes our new approach for TDD. And, finally Secs. 7 and 8 concludes with the ongoing research and future plans.

2. Related Work

2.1. *Oracle generation*

Most previous research aimed at improving the efficiency of software testing falls into two categories: one is focused on the test case selection [15, 14, 25, 26], the other has concentrated on developing tools to help generate, maintain and track the testing documentation or run tests in simulated environments [9, 16, 27, 28]. This previous research is supportive to, but different from the work that is reported in this paper.

Several researchers have developed tools that give the user the ability to determine if the results of a test is correct or not. In [27], Panzl explained three different kinds of automatic software test drivers that can be used to automate the verification of test results. In [16], Hamlet described another automatic testing system based on finite test-data sets, implemented by modifying a compiler. The disadvantages of these testing systems are: (1) The user should specify the expected result, which may be hard to acquire, and (2) Relational specifications, which may accept more than one acceptable result for a given input, can't be used because these systems only compare the expected and actual result.

The last disadvantage is partly solved by Chapman in [9], which describes the design and implementation of a program testing assistant which aids a programmer in the definition, execution, and modification of test cases during incremental program development. Moreover, it gives the programmer the ability to set the success criteria for a test case or use the default criterion *equal*, which checks for simple equality of a result and its correct value. Examples of other success criteria are *set-equal*, which checks two sets to see that they contain the same elements and *isomorphic*, which checks that arbitrary structures, possibly including pointer cycles, are topologically identical.

In [33], Peters and Parnas discussed the use of test oracles generated from program documentation. They described an algorithm that can be used to generate a test oracle

from program documentation, and presented the results of using a tool based on it to help test part of a commercial network management application. The results demonstrated 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 the prototype test oracle generator they described allows using only the C programming language. If there is a need to choose among several programming languages, one must add several additional sub modules, one for each language.

Other systems, such as JML [21], ANNA [22] and APP [36], give the user the ability to write a code annotated with assertions that are evaluated while the code is executed. These assertions can be used as an oracle if they are completely specified and accurately placed so that they define the program specification. In [10], Cheon and Leavens described a new approach for generating test oracles from formal specifications. In their work, they used JML which is based on pre/postcondition approaches to write the specifications. In our work, we used relational specification to describe the behavior of the program. Based on the study in [12], pre/postcondition approaches have limited abilities for specifying program behaviors since they don't consider liveness properties or non-terminating behaviors. Relational specification methods can handle both deterministic and non-deterministic programs. Moreover, they have powerful expressive abilities that enable specifiers to deal with all three kinds of terminating behaviors of programs. In addition, relational specification methods are capable of specifying both partial and total correctness. Pre/Postcondition approaches have to ignore some complicated programming issues (e.g. pointers, aliasing problems, parameter passing and scope rules) otherwise the complexity of the formulae will increase considerably. Relational approaches do not have this problem. Also, in our work we use tabular expressions which are a powerful means for constructing and presenting program documents.

In [38], Stocks and Carrington described a Test Template Framework (TTF) which is a structured strategy and a formal framework for Specification-based Testing (SBT) which is using the Z notation. In [35], Richardson *et al.* encourage the process of generating test oracles from formal specifications. In [23], they described an approach for generating C++ test oracle classes from Object-Z specifications.

Other researchers have explained the process of generating test oracles for abstract data types (ADTs) that are defined using algebraic specifications, e.g. [5, 8, 13] or "trace" specifications [39]. These kinds of specification approaches discuss how to specify the desired properties of an ADT which is implemented by a group of programs. The specification approaches that are used in this work are used to specify the effect of a single program on some data structure also can be used to specify the desired properties of an ADT which is implemented by a group of programs.

The work reported in this paper is similar to the work in [33] 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 can 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.
- 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 Fig. 3. 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 easily 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.

2.2. *Test driven development*

Recently, there are studies to analyze the efficiency of the TDD approach. Muller and Hagner [24] report an experiment to compare TDD with traditional programming. The experiment, done with 19 graduate students as subjects who were divided into two groups, TDD and control, evaluated the efficiency of TDD in terms of (1) programming speed, (2) program reliability and (3) program understanding. They found no difference between the groups in overall development time. The TDD group had lower reliability after the implementation phase and higher reliability after the acceptance-test phase. However, the TDD groups had statistically significant fewer errors when the code was reused. Based on these results the researchers concluded that writing programs in test-first manner neither leads to quicker development nor provides an increase in quality. However, the understandability of the program increases, measured in terms of proper reuse of existing interfaces.

There are some researchers who have described tools that can be used to combine formal specifications with test-driven development without loosing the agility of test-driven development. In [6], Baumeister describes a tool that provides support to combine formal specifications with test driven development. This is done by using the tests that drive the development of the code to also drive the development of the formal specification. By generating runtime assertions from the specification it is possible to check for inconsistencies between code, specifications, and tests. Each of the three artifacts improves the quality of the other two, yielding better code quality and better program documentation in the form of a validated formal specification of the program. This method is exemplified by using the primes example with Java as the programming language, JUnit as the testing framework, and the Java Modeling Language (JML) [21] for the formulation of class invariants and pre- and post-conditions for methods. They use JML since JML specifications are easily understood by programmers, and because it comes with a runtime assertion checker [11], which allows them to check invariants and pre- and postconditions of methods at runtime.

Our work is different from the work above in the sense that we use relations for the specifications, which characterize the acceptable set of outcomes for a given input. In addition, we use test oracles that are generated automatically from the program specifications to determine if the software behavior is correct or not for a given test input and output. Hence, 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.

In [37], Staff proposes that developer testing frameworks should support, as first order constructs alongside traditional tests, partial specifications over large or infinite sets of values, called Theories. Theories look like test methods, but are universally quantified: all assertions must hold for any possible parameter values that pass the assumptions. They used the theories as specifications of the behavior.

In [17], Herranz and Moreno-Navarro have studied how the technology of Formal Methods (FM) can interact with an agile process in general and with Extreme Programming (XP) in particular. They have presented how some XP practices can admit the integration of Formal Methods and declarative technology (functional and logic programming).

3. Formal Software Specifications

Formal Specifications are documentation methods that use a mathematical description of software or hardware, which may be used to develop an implementation and to drive automated testing. The emphasis is on what the system should do, not necessarily how the system should do it. Moreover, formal software specifications are expressed in a language whose vocabulary, syntax and semantics are formally defined. Examples of such languages (or notations) are VDM, Z, and B.

With reference to the set of documents described in [31], in this work, we focus on deriving test oracles from the module internal design document [32] and module interface specification [34]. These two types of documents can be used to verify the workability of the design. Additionally, they describe the module's data structure, state the intended interpretation of that data structure (in terms of the external interface), and specify the effect of each access-program on the module's data structure.

The nature of computer system behavior often is 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 that must be described. It has been recognized for some time that tables can be used to help in the effective presentation of such mathematics [30, 1, 29, 18]. In our work, we show such tabular representation of relations and functions as a significant factor in making the documentation more readable, and so we have specialized our tools to support them.

A complete discussion of tabular expressions is beyond the scope of this paper, so interested readers are referred to the cited publications. In their most basic form,

tabular expressions represent conditional expressions. For example, the function definition on the left below, could be represented by the tabular expression on the right.

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

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

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

Although this function is clearly a nonsensical example, it is representative of the kind of conditional expressions that often occur in the documentation of software-based systems. We have found that the tabular form of expressions is not only easier to read but most importantly, the expressions are also easier to write correctly. Of particular importance is the fact that they make it very clear what the cases are, and that all cases are considered.

Modern general purpose documentation tools obviously have support for tables as part of the documents, but they are often not very good at dealing with tables as part of mathematical expressions. These tools also encourage authors to focus efforts on the wrong things: authors will work very hard to try to get the appearance of the table right, sometimes even to the detriment of readability (e.g. shortening variable names so that expressions fit in the columns).

In this work, a program specification describes the required behavior of a program either in terms of the internal data structure and the effect of each program access on it, or in terms of the externally observable behavior 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. For more details about these components see [3, 2].

3.1. *Sample program specification*

Figure 2, specifies a program "`ggcd`" which compares an integer value "`i`" with another integer value "`j`", returns the greatest common divisor of them if "$\texttt{i} > \texttt{0} \wedge j > 0$", 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`".

Program Specification

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

	$\mathtt{i} > 0 \wedge \mathtt{j} > 0$	$\mathtt{i} \leq 0 \vee \mathtt{j} \leq 0$
gcdvalue =	$\max(\{x \in [0, \min(\mathtt{i}, \mathtt{j})] \mid \mathrm{cDiv}(\mathtt{i}, \mathtt{j}, x)\})$	0
result =	TRUE	FALSE

Auxiliary Predicate Definitions
Boolean cDiv(**Integer** a, **Integer** b, **Integer** x)
$\stackrel{\mathrm{df}}{=} (\mathrm{a}\%\mathrm{x} = 0) \wedge (\mathrm{b}\%\mathrm{x} = 0)$

Fig. 2. ggcd program specification.

4. Tool Support

The goal of this project is to develop techniques and tools to facilitate the production of software design documentation that is (1) readable and understandable by the users, (2) complete and accurate enough to allow analysis, both manually and mechanically and (3) suitable for use as a specification from which an acceptable program can be produced. We cannot get these benefits with general purpose word processors.

4.1. *OMDoc document model*

As described in [20], the OMDoc (Open Mathematical Documents) format is a content markup scheme for (collections of) mathematical documents including articles, textbooks, interactive books, and courses. OMDoc also serves as the content language for the communication of mathematical software. OMDoc is an extension of the OpenMath and (content) MathML standards and concentrates on representing the meaning of mathematical formulae instead of their appearance. OpenMath and MathML are formats for individual mathematical expressions and OMDoc is a format for documents that include mathematics. 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. For more details about our specification model see [2].

4.2. *The Eclipse framework*

Eclipse is a software platform that consists of 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 with an integrated development environment (IDE). 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.

4.3. *Specification editor*

As a part of our tools, we are developing a specification editor to support production of software documents, which is illustrated in Fig. 3. 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 including the RIACA OpenMath Library.[a]

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

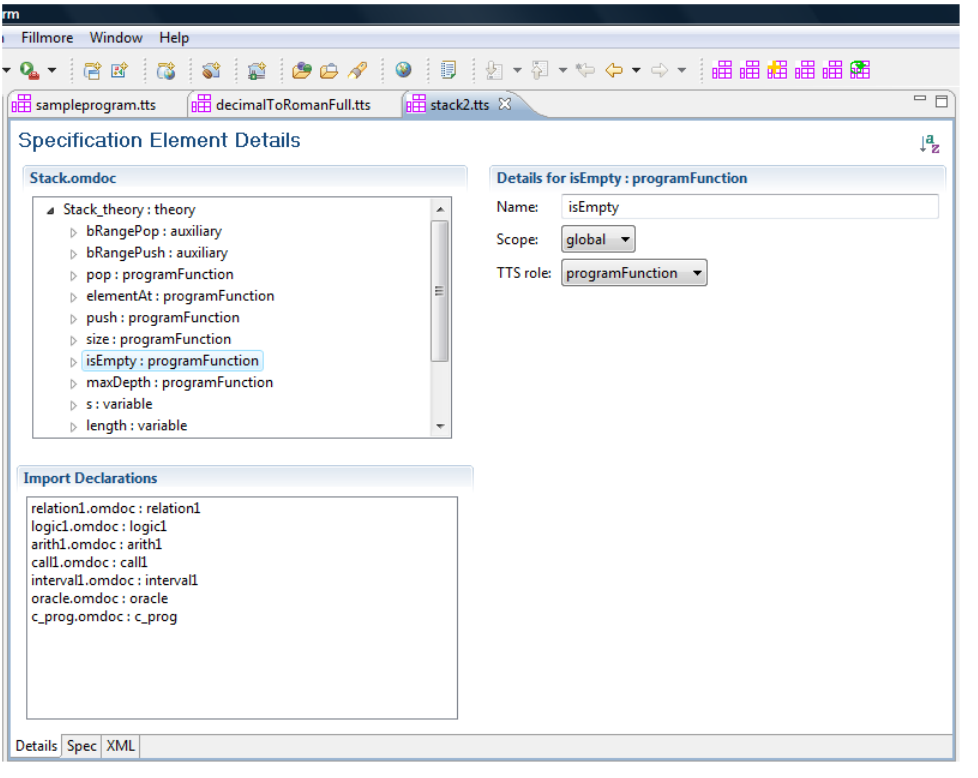

Fig. 3. Screenshot of editor.

[a] http://www.mathdox.org/projects/openmath/lib/2.0/index.html

5. Oracle Design

The oracle is implemented using Java. This decision should not be seen as a significant feature of the design — if the intended application were different, the oracle design could be translated with some changes.

5.1. *Internal design overview*

The oracle can be viewed as a "compiled" version of the specification in the sense that it is generated by translating the "source" specification into an executable form (Java code). The oracle can be executed without reference to the specification from which it was derived. Therefore, it can be integrated smoothly with test frameworks (e.g. JUnit). This design has an advantage in the sense that it reduces the time required for oracle execution by giving the user the ability to use optimization techniques.

An alternative approach to designing the oracle is to build it as an "interpreter" which would represent the specification by data and evaluate it directly. This kind of design has the advantage of making the oracle generation process relatively simple and, since there is no generated code involved in the oracle, the oracle programs will be the same for any specification, except for the fact that the data used is dependent on the specification. A disadvantage of this design is that the oracle will need to interpret the semantics of the documentation during evaluation, and hence, it would probably be comparatively slower to execute.

5.1.1. *Expression implementation*

Any expression consists of one or more sub-expressions, the complexity of implementing this expression is managed by decomposing each expression into its sub-expressions and implementing each sub-expression individually. The oracle code thus consists of a set of internal functions and objects, each of which implements a sub-expression and may call other internal functions or object methods.

All programming languages in general, and Java in particular, provide support for basic logical and relational operators (i.e. $\wedge, \vee, \neg, >, <, =$ etc.). Where the operators in the specification expression are directly representable using Java logical and relational operators, they are implemented as such.

Each expression is implemented by instantiating a Java object, which includes references to objects implementing the sub-expressions. This helps to simplify the oracle generation process for expressions that have complex semantics such as tabular expressions. Consequently, the test oracle generator only need to translate the expression into the suitable object constructor.

5.1.2. *Quantification*

Quantifier expressions are implemented by using loops that call the suitable procedures to enumerate the elements of the (finite) set over which the quantification is bounded. In the example below, the set is the integer interval bounded by 0 and 10. One distinction between the work reported in this paper and that in [33] is that the

previous work used Inductively Defined Predicate to specify the range for the quantification but we used a Java collection.

The quantification "$(\forall i : \{0..10\}.p\_B[i] = p\_x)$", can be implemented as follows.

```
boolean result=true;
Integer_Interval bRange =new Integer_Interval(0,10);
Integer i=new Integer(0);
for(Iterator<Integer> it=bRange.iterator();
    it.hasNext()&&result;)
{
     i=it.next();result=((p_B[i]==p_x)&&result);
}
```

5.1.3. *Tabular expressions*

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, hence there is no need for this knowledge to be in the TOG. The expression in each cell of the table is implemented as a Java class that extends a CellBase class and therefore contains a procedure, eval, which evaluates the expression in the cell.

Table objects have the following method, which is used to evaluate the table:

evaluateTable finds the index for the main cell that should be evaluated and returns the contents of that cell.

The expression "$\texttt{i} > 0 \wedge \texttt{j} > 0$", which is in the first cell of the column header of the ggcd tabular expression in Fig. 2, is implemented as follows.

```
package oracles;
import ca.Fillmoresoftware.plugin.OracleUtilities.*;
public class ggcd1_Grid_2_Cell_0 extends
CellBase{
     private VarMap vars;
     public ggcd1_Grid_2_Cell_0(VarMap vars){
          this.vars=vars;
     }
     public Object eval(){
          Integer  i=(Integer)vars.getValue("i");
          Integer  j=(Integer)vars.getValue("j");
          return ((i>0)&&(j>0));
     }
}
```

The other cells in each table are implemented in a similar fashion.

5.1.4. *Auxiliary functions*

An auxiliary function is implemented as a procedure, with the expression, implemented as described above, forming the body of the procedure.

Suitable calls to this procedure are used in the code that implements expressions using the auxiliary function.

5.2. *Compilation and execution*

The oracle in our approach consists of two kinds of code: that generated by the Test Oracle Generator (TOG), and object classes (e.g. Integer Interval.java, InvertedTable.java, NormalTable.java and VectorTable.java), previously manually implemented and used by the TOG generated code. These table classes contain all knowledge of the semantics of tabular expressions and provide several methods (addHeaderCell, addMainCell, getMainCell, evaluateTable) which give the user the ability to create and evaluate the tabular expressions. The Integer Interval class is a Java collection used to implement the finite set containing the integers in a specified range for the quantifications.

The code below shows the implementation of the root class for the oracle (ggcdOracle.java) for the sample program specification described in Sec. 3.1.

```
package oracles;
import ca.Fillmoresoftware.plugin.OracleUtilities.*;
import static org.junit.Assert.*;
public class ggcdOracle{
    private VarMap vars;
    private Outggcd1 t0;
    public ggcdOracle(){
        vars=new VarMap();
        t0=new Outggcd1(vars);
    }
    private Boolean ggcdTOracle(Integer i,
            Integer j,Integer gcdvalue,
            Boolean result){
        Boolean resultOracle;
        vars.setValue("i",i);
        vars.setValue("j",j);
        vars.setValue("gcdvalue",gcdvalue);
        vars.setValue("result",result);
        resultOracle=t0.ggcdT1();
        return resultOracle;
    }
```

```
    public void assertggcdTOracle(Integer i,
            Integer j,Integer gcdvalue,
            Boolean result){
        assertTrue(ggcdTOracle(i,j,gcdvalue,result));
    }
}
```

Using the oracle involves implementing test code that calls the program under test and then calls the oracle procedures. In this work, the JUnit framework is used since it has a number of advantages. One important advantage of JUnit is that it is widely used, which will make it easier for others to understand the test cases and write new ones. In addition, it provides a graphical user interface (GUI) which makes it easier to write and test the program 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. This is makes it easy for the software developer to quickly identify failing test cases as they are found. The code below shows how to run the oracle generated from the sample program specification in Sec. 3.1 with JUnit:

```
package oracles;
import org.junit.Before;
import org.junit.Test;
public class OracleTest extends junit.framework.TestCase{
    ggcdOracle com;
    @Before
    public void setUp() throws Exception {
        com=new ggcdOracle();
    }
    @Test
    public void testCon(){
        Integer gc=GCD.gcd(25,20);
        com.assertggcdTOracle(25,20,gc,true);
    }
}
```

The previous code contains one test case to test if the program correctly finds the greatest common divisor of $(25, 20)$ which is 5. The greatest common divisor is computed by the static method GCD.gcd(int,int) which is meant to implement the specification. The user can add any number of test cases. The result for the previous code is shown in Fig. 4.

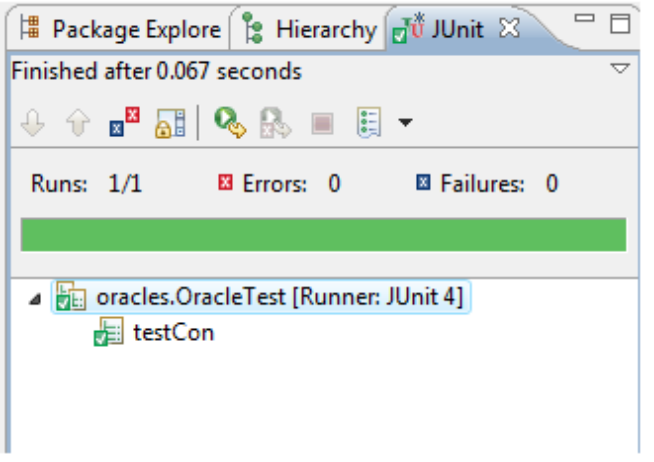

Fig. 4. Test result.

6. Test Driven Development with Oracles

This section describes our new approach for TDD. It also describes a real problem in a real-world development environment which shows how to apply this approach.

The process is as described below:

- Write the specification for some required behavior.
- 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.

6.1. *Trial application*

We have applied our TDD approach to build programs used in a project at Memorial University called Sustainable Technology for Polar Ships & Structures (STePS^2)[b] which supports sustainable development of polar regions by developing direct design tools for polar ships and offshore structures. Direct design improves on traditional design methods by calculating loads and responses against defined performance criteria. The software used in our test are to analyze particle interactions for the purpose of developing ice-structure models. We hope that this application will help

[b] http://www.engr.mun.ca/steps2/index.php

us to demonstrate the practicality and effectiveness of this approach, and to gain appreciation of its strengths and weaknesses.

Now, we will work through the example to show the whole process for specification supported TDD. According to our approach, the first step is to write a specification for some required behavior. Consequently, we have initiated this specification as:

Double `newPosition`(**Double** `p`, **Double** `v`, **Double** `deltaTime`)
$\stackrel{\text{df}}{=}$ `result` $=$ `p` $+$ `v` $*$ `deltaTime`

The above specification consists of the definition for `newPosition`(`p`,`v`,`deltaTime`) function which represents the program function. We have used the convention that `result` represents the value returned by the function. The required behavior that is represented by this specification is to find the position for a particle after certain amount of time based on its old position and velocity.

After writing the specification, we generate the test oracle from it and run the test oracle to make sure that the program behavior is consistent with the required behavior. Following the TDD approach, the test cases should initially fail since we haven't implemented the program yet. We then implement enough of the program to make the cases pass.

The previous specification only defines the calculation of the particle position in one dimension but in this work, we are concerned with the position in three dimensions. Therefore, we need to add a specification for a **Vector3D** class. We can rewrite the previous specification as follows:

Vector3D `newPosition`(**Vector3D** `p`, **Vector3D** `v`,
Double `deltaTime`)
$\stackrel{\text{df}}{=}$ `result` $=$ `p.add(v).scalarMul(deltaTime)`

Vector3D Specifications

Data Structure

Double `x`
Double `y`
Double `z`

Program Functions

Vector3D `Vector3D`(**Double** `a`, **Double** `b`, **Double** `c`)
$\stackrel{\text{df}}{=}$ (`result.getX()` $=$ `a`) $\wedge$ (`result.getY()` $=$ `b`) $\wedge$ (`result.getZ()` $=$ `c`)

void `setX`(**Double** `a`)
$\stackrel{\text{df}}{=}$ `x` $=$ `a`

void `setY`(**Double** `b`)
$\stackrel{\text{df}}{=}$ `y` $=$ `b`

void `setZ`(**Double** `c`)
$\stackrel{\text{df}}{=}$ `z` = `c`

Double `getX`()
$\stackrel{\text{df}}{=}$ `result` = `x`

Double `getY`()
$\stackrel{\text{df}}{=}$ `result` = `y`

Double `getZ`()
$\stackrel{\text{df}}{=}$ `result` = `z`

Double `magnitude`()
$\stackrel{\text{df}}{=}$ `result` = `Math.sqrt(x * x + y * y + z * z)`

Vector3D `normalize`()
$\stackrel{\text{df}}{=}$ (`result.getX()` = `this.getX()/this.magnitude()`)∧
(`result.getY()` = `this.getY()/this.magnitude()`)∧
(`result.getZ()` = `this.getZ()/this.magnitude()`)

Vector3D `add`(**Vector3D** `a`)
$\stackrel{\text{df}}{=}$ (`result.getX()` = `this.getX()` + `a.getX()`)∧
(`result.getY()` = `this.getY()` + `a.getY()`)∧
(`result.getZ()` = `this.getZ()` + `a.getZ()`)

Vector3D `subtract`(**Vector3D** `a`)
$\stackrel{\text{df}}{=}$ (`result.getX()` = `this.getX()` − `a.getX()`)∧
(`result.getY()` = `this.getY()` − `a.getY()`)∧
(`result.getZ()` = `this.getZ()` − `a.getZ()`)

Vector3D `dotProduct`(**Vector3D** `a`)
$\stackrel{\text{df}}{=}$ (`result.getX()` = `this.getX()` ∗ `a.getX()`)∧
(`result.getY()` = `this.getY()` ∗ `a.getY()`)∧
(`result.getZ()` = `this.getZ()` ∗ `a.getZ()`)

Vector3D `crossProduct`(**Vector3D** `a`)
$\stackrel{\text{df}}{=}$ (`result.getX()` = (`this.getY()` ∗ `a.getZ()`)−
(`this.getZ()` ∗ `a.getY()`)) ∧ (`result.getY()` = (`this.getZ()`∗
`a.getX()`) − (`this.getX()` ∗ `a.getZ()`)) ∧ (`result.getZ()` =
(`this.getX()` ∗ `a.getY()`) − (`this.getY()` ∗ `a.getX()`))

Vector3D `scalarMul`(**Double** `c`)
$\stackrel{df}{=}$ (`result.getX()` = `this.getX()` $*$ `c`)$\wedge$
(`result.getY()` = `this.getY()` $*$ `c`)$\wedge$
(`result.getZ()` = `this.getZ()` $*$ `c`)

Note that the above functions return a new instance of the **Vector3D**. Now, the specification defines the calculation of the position in three dimensions. After we have refined the initial specification, we repeat the same steps as in the previous one. Again, we refine the implementation until the behavior is consistent with the specification. As we have seen the previous specification doesnot encapsulate the particle properties e.g. position and velocity in a class. To do that, we need to add a specification for the **Particle** class and then revise the previous specification as follows:

Vector3D `newPosition`(**Particle** `p`, **Double** `deltaTime`)
$\stackrel{df}{=}$ `result` = `p.getPos().add(p.getVel().scalarMul(deltaTime))`

Particle Specifications

Data Structure

Double `radius`
Double `mass`
Vector3D `pos`
Vector3D `vel`

Program Functions

Particle `Particle`(**Double** `r`, **Double** `m`, **Vector3D** `p`,
Vector3D `v`)
$\stackrel{df}{=}$ (`result.getRadius()` = `r`) $\wedge$ (`result.getMass()` = `m`)$\wedge$
(`result.getPos()` = `p`) $\wedge$ (`result.getVel()` = `v`)

void `setRadius`(**Double** `r`)
$\stackrel{df}{=}$ `radius` = `r`

void `setMass`(**Double** `m`)
$\stackrel{df}{=}$ `mass` = `m`

void `setPos`(**Vector3D** `p`)
$\stackrel{df}{=}$ `pos` = `p`

void `setVel`(**Vector3D** `v`)
$\stackrel{df}{=}$ `vel` = `v`

Double `getRadius`()
$\stackrel{\text{df}}{=}$ `result = radius`

Double `getMass`()
$\stackrel{\text{df}}{=}$ `result = mass`

Vector3D `getPos`()
$\stackrel{\text{df}}{=}$ `result = pos`

Vector3D `getVel`()
$\stackrel{\text{df}}{=}$ `result = vel`

The above specification encapsulates the particle properties in one class. Again, we generate the test oracle and implement a test case. The particle code is then developed until the test case passes, and so it implements the specified behavior. Continuing the development, we add the specification for `newVelocity` function:

Vector3D `newVelocity`(**Particle** `p`, **Vector3D** `gravity`, **Double** `deltaTime`)
$\stackrel{\text{df}}{=}$ `result = p.getVel().add(gravity.scalarMul(deltaTime))`

Continuing in this manner, we eventually reach the full specification of the programs, which is shown in the appendix, and we have simultaneously developed a full implementation and a full suite of test cases.

We have used the test oracles generated from specifications such as those above to test the software that we are developing in the simulation project. Such simulation software typically requires a lot of calculations so manual oracles are quite cumbersome to develop because the tester is required to spend a lot of time on doing the calculations, which are also prone to human errors. Specifying the behavior of the software formally and using the generated test oracles to test the software solves these problems.

Using this approach of TDD helped us to build full specification of the programs, as shown above, and to develop a full implementation and a full suite of test cases. But if we apply the traditional TDD to the same application, we will end up with only a full implementation and a full suite of test cases. The traditional TDD has disadvantages of lack of documentation and over-specific test cases. Therefore, using this approach of TDD will overcome these two disadvantages by assoicating the formal specification and test oracles in the TDD process.

While this small trial is clearly not a sufficient empirical study to draw conclusions about the impact of these techniques on software development cost or quality, we are convinced that using this approach has helped us to improve the quality of the software that we are developing.

7. Conclusion

The problems mentioned in this work are central to software testing. The use of formal specification techniques has been shown to improve the quality of software but the industrial community has been slow to accept them because they are seen as greatly increasing the up-front cost of system development without significant measurable benefit. The ability to test a program using its specification as an oracle will greatly increase the value of such formal specification by reducing the cost of testing and helping to ensure that errors that occur during testing are detected. The test oracle generator can also be used to ensure that specification is kept up to date: if a program is always tested against the specification then everyone is assured that the specification is consistent with the program behavior.

In test-driven development, tests are used to specify the behavior of the program, and the tests are additionally used as a documentation of the program. However, tests are not sufficient to completely define the behavior of a program because they only define the program behavior by example and they do not state general properties. Therefore, the latter can be achieved by using our TDD approach, which uses a formal specification to specify the behavior of the program and supports testing directly against that specification by generating oracles. The outcome of this technique is that, at the end of the development period, the developer has produced not only a working implementation, but also a complete specification and a full set of test cases.

We are convinced that using this approach in large industry projects will help to improve the quality of the software.

8. Future Work

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. Also, applying this approach in a large industry projects.

Other possible improvements in the tool set (e.g. better visual editing, etc.) could be done in the future development of these tools. In addition to that, part of the future work is using these tools to do analysis of the test cases (e.g. coverage of the specification).

Acknowledgments

This research was supported by the School of Graduate Studies and 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).

Appendix A. Full Specification of the Programs

Program Functions

Vector3D `totalAngMom`(**Particle** `p`[ ])
$\stackrel{\text{df}}{=}$ `result` = centerOfMass(`p`).`crossProduct`(
velocityOfCMass(`p`).`scalarMul`(totalMass(`p`))).`add`(relMom(`p`))

Vector3D `newPosition`(**Particle** `p`, **Double** `deltaTime`)
$\stackrel{\text{df}}{=}$ `result` = `p`.`getPos`().`add`(`p`.`getVel`().`scalarMul`(`deltaTime`))

Vector3D `newVelocity`(**Particle** `p`, **Vector3D** `gravity`,
Double `deltaTime`)
$\stackrel{\text{df}}{=}$ `result` = `p`.`getVel`().`add`(`gravity`.`scalarMul`(`deltaTime`))

Vector3D `springForce`(**Particle** `a`, **Particle** `b`,
Double `spring`)
$\stackrel{\text{df}}{=}$ `result` = getRelPos(`a`, `b`).`normalize`().`scalarMul`(
$-$`spring` $*$ (getCollideDist(`a`, `b`) $-$ getRelPos(`a`, `b`).`magnitude`()))

Vector3D `shearForce`(**Particle** `a`, **Particle** `b`,
Double `shear`)
$\stackrel{\text{df}}{=}$ `result` = getRelVel(`a`, `b`).`subtract`(getRelPos(`a`, `b`).
`normalize`().`scalarMul`(getRelVel(`a`, `b`).`dotProduct`(
getRelPos(`a`, `b`).`normalize`()))).`scalarMul`(`shear`)

Vector3D `dampingForce`(**Particle** `a`, **Particle** `b`, **Double** `damping`)
$\stackrel{\text{df}}{=}$ `result` = getRelVel(`a`, `b`).`scalarMul`(`damping`)

Vector3D `attractionForce`(**Particle** `a`, **Particle** `b`, **Double** `attraction`)
$\stackrel{\text{df}}{=}$ `result` = getRelPos(`a`, `b`).`scalarMul`(`attraction`)

Auxiliary Function Definitions

Double getCollideDist(**Particle** p1, **Particle** p2)
$\stackrel{\text{df}}{=}$ (p1.`getRadius`() + p2.`getRadius`())

Vector3D getRelPos(**Particle** p1, **Particle** p2)
$\stackrel{\text{df}}{=}$ (p2.`getPos`().`subtract`(p1.`getPos`()))

Vector3D getRelVel(**Particle** p1, **Particle** p2)
$\stackrel{\text{df}}{=}$ (p2.`getVel`().`subtract`(p1.`getVel`()))

Vector3D centerOfMass(**Particle** p[ ])
$\stackrel{\text{df}}{=}$ (1/totalMass(p)) $\sum_{i=0}^{n}$ p[i].`getPos`().`scalarMul`(p[i].`getMass`())

Double totalMass(**Particle** p[ ])
$\stackrel{\text{df}}{=}$ $\sum_{i=0}^{n}$ p[i].`getMass`()

Vector3D velocityOfCMass(**Particle** p[ ])
$\stackrel{\text{df}}{=}$ (1/totalMass(p)) $\sum_{i=0}^{n}$ p[i].`getVel`().`scalarMul`(p[i].`getMass`())

Vector3D relMom(**Particle** p[ ])
$\stackrel{\text{df}}{=}$ $\sum_{i=0}^{n}$ p[i].`getPos`().`subtract`(centerOfMass(p)).
`crossProduct`(p[i].`getVel`().`subtract`(velocityOfMass(p)).`scalarMul`(p[i].`getMass`()))

Particle Specifications

Data Structure

Double `radius`
Double `mass`
Vector3D `pos`
Vector3D `vel`

Program Functions

Particle `Particle`(**Double** `r`, **Double** `m`, **Vector3D** `p`, **Vector3D** `v`)
$\stackrel{\text{df}}{=}$ (`result.getRadius()` = `r`) $\wedge$ (`result.getMass()` = `m`)$\wedge$
(`result.getPos()` = `p`) $\wedge$ (`result.getVel()` = `v`)

void `setRadius`(**Double** `r`)
$\stackrel{\text{df}}{=}$ `radius` = `r`

void `setMass`(**Double** `m`)
$\stackrel{\text{df}}{=}$ `mass` = `m`

void `setPos`(**Vector3D** `p`)
$\stackrel{\text{df}}{=}$ `pos` = `p`

void `setVel`(**Vector3D** `v`)
$\stackrel{\text{df}}{=}$ `vel` = `v`

Double `getRadius`()
$\stackrel{\text{df}}{=}$ `result` = `radius`

Double `getMass`()
$\stackrel{\text{df}}{=}$ `result` = `mass`

Vector3D `getPos`()
$\stackrel{\text{df}}{=}$ `result` = `pos`

Vector3D `getVel`()
$\stackrel{\text{df}}{=}$ `result` = `vel`

Vector3D Specifications

Data Structure

Double `x`
Double `y`
Double `z`

Program Functions

Vector3D `Vector3D`(**Double** `a`, **Double** `b`, **Double** `c`)
$\stackrel{\text{df}}{=}$ (`result.getX()` = `a`) $\wedge$ (`result.getY()` = `b`) $\wedge$ (`result.getZ()` = `c`)

void `setX`(**Double** `a`)
$\stackrel{\text{df}}{=}$ `x` = `a`

void `setY`(**Double** `b`)
$\stackrel{\text{df}}{=}$ `y` = `b`

void `setZ`(**Double** `c`)
$\stackrel{\text{df}}{=}$ `z` = `c`

Double `getX`()
$\stackrel{\text{df}}{=}$ `result` = `x`

Double getY()
$\stackrel{\text{df}}{=}$ result = y

Double getZ()
$\stackrel{\text{df}}{=}$ result = z

Double magnitude()
$\stackrel{\text{df}}{=}$ result = Math.sqrt(x ∗ x + y ∗ y + z ∗ z)

Vector3D normalize()
$\stackrel{\text{df}}{=}$ (result.getX() = this.getX()/this.magnitude())∧
(result.getY() = this.getY()/this.magnitude())∧
(result.getZ() = this.getZ()/this.magnitude())

Vector3D add(**Vector3D** a)
$\stackrel{\text{df}}{=}$ (result.getX() = this.getX() + a.getX())∧
(result.getY() = this.getY() + a.getY())∧
(result.getZ() = this.getZ() + a.getZ())

Vector3D subtract(**Vector3D** a)
$\stackrel{\text{df}}{=}$ (result.getX() = this.getX() − a.getX())∧
(result.getY() = this.getY() − a.getY())∧
(result.getZ() = this.getZ() − a.getZ())

Vector3D dotProduct(**Vector3D** a)
$\stackrel{\text{df}}{=}$ (result.getX() = this.getX() ∗ a.getX())∧
(result.getY() = this.getY() ∗ a.getY())∧
(result.getZ() = this.getZ() ∗ a.getZ())

Vector3D crossProduct(**Vector3D** a)
$\stackrel{\text{df}}{=}$ (result.getX() = (this.getY() ∗ a.getZ())−
(this.getZ() ∗ a.getY())) ∧ (result.getY() = (this.getZ()∗
a.getX()) − (this.getX() ∗ a.getZ())) ∧ (result.getZ() =
(this.getX() ∗ a.getY()) − (this.getY() ∗ a.getX()))

Vector3D scalarMul(**Double** c)
$\stackrel{\text{df}}{=}$ (result.getX() = this.getX() ∗ c) ∧ (result.getY() = this.getY() ∗ c)∧
(result.getZ() = this.getZ() ∗ c)

References

1. R. F. Abraham, Evaluating generalized tabular expressions in software documentation, M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, Feb. 1997.
2. S. Alawneh and D. Peters, Specification-based test oracles with JUnit, in *Canadian Conference on Electrical and Computer Engineering (CCECE)*, Canada, May, 2010.
3. S. Alawneh and D. Peters, Test driven development with oracles and formal specifications, in *Proc. IFIP International Conference on Testing Software and Systems (ICTSS)*, Natal, Brazil, Nov. 2010.
4. S. Ambler, *Agile Database Techniques: Effective Strategies for the Agile Software Developer* (Wiley, 2003).
5. S. Antoy and D. Hamlet, Self-checking against formal specifications, in W. W. Koczkodaj, P. E. Lauer and A. A. Toptsis (editors), *Proc. Int'l Conf. Computing and Information* (1992), pp. 355−360.
6. H. Baumeister, Combining formal specifications with test driven development, in *Extreme Programming and Agile Methods — XP/Agile Universe 2004, 4th Conference on Extreme Programming and Agile Methods*, Lecture Notes in Computer Science, Vol. 3134, 2001, pp. 1−12.
7. K. Beck, *Extreme Programming Explained: Embrace Change* (Addison-Wesley, 2000).
8. G. Bernot, M. Gaudel and B. Marre, Software testing based on formal specifications: A theory and a tool, *Software Engineering Journal* **6** (1990) 387−405.
9. D. Chapman, A program testing assistant, *Communications ACM* **25**(9) (1982) 625−634.
10. Y. Cheon and G. T. Leavens, A simple and practical approach to unit testing: The jml and junit way, ECOOP 2002, pp. 231−255.
11. Y. Cheon and G. T. Leavens, A runtime assertion checker for the Java modeling language (JML), in H. R. Arabnia and Y. Mun (editors), *International Conference on Software Engineering Research and Practice (SERP-02)*, CSREA Press, Las Vegas, 2002, pp. 322−328.
12. H. Duan, A comparative study of pre/post condition and relational approaches to program development, Master of Science thesis, McMaster University, Hamilton, ON, Dec. 2004.
13. J. Gannon, P. McMullin and R. Hamlet, Data-abstraction implementation, specification, and testing, *ACM Trans. Programming Languages and Systems* **3**(3) (1981) 211−223.
14. D. Gelperin and B. Hetzel, The growth of software testing, *Communications ACM* **31**(6) (1988) 687−695.
15. J. B. Goodenough and S. L. Gerhart, Toward a theory of test data selection, *IEEE Trans. Software Engineering* **1**(2) (1975) 156−173.
16. R. Hamlet, Testing programs with the aid of a compiler, *IEEE Trans. Software Engineering* **3**(4) (1977) 279−290.
17. A. Herranz and J. J. Moreno-Navarro, Formal extreme (and extremely formal) programming, in *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003*, LNCS, Vol. 2675, 2003, pp. 88−98.
18. R. Janicki, On a formal semantics of tabular expressions, CRL Report 355, Communications Research Laboratory, Hamilton, Ontario, Canada, Oct. 1997.
19. R. Jeffries, A. Anderson and C. Hendrickson, *Extreme Programming Installed* (Addison-Wesley, 2001).
20. M. Kohlhase, *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*, Lecture Notes in Artificial Intelligence, Vol. 4180, 2006.

21. 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, 1999, pp. 175–188.
22. D. Luckham, F. von Henke, B. Krieg-Brückner and O. Owe, *ANNA A Language for Annotating Ada Programs Reference Manual*, Lecture Notes in Computer Science, Vol. 260, 1987.
23. J. McDonald, L. Murray and P. Strooper, Translating object-z specifications to object-oriented test oracles, *4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC '97/ICSC '97)*, 2–5 December 1997, Clear Water Bay, Hong Kong.
24. M. Muller and O. Hagner, Experiment about test-first programming, *IEEE Software*, October 2002.
25. G. J. Myers, *The Art of Software Testing* (John Wiley & Sons, 1979).
26. T. J. Ostrand and M. J. Balcer, The category-partition method for specifying and generating functional tests, *Communications ACM* **31**(6) (1988) 676–686.
27. D. Panzl, Automatic software test drivers, *Computer*, 1978, pp. 44–50.
28. D. J. Panzl, A language for specifying software tests, in S. P. Ghosh and L. Y. Liu (editors), *Proc. National Computer Conf.*, 1978, pp. 609–619.
29. D. L. Parnas, Tabular representation of relations, CRL Report 260, Communications Research Laboratory, Hamilton, Ontario, Canada, Nov. 1992.
30. D. L. Parnas, Inspection of safety critical software using function tables, in *Proc. IFIP Congress*, Vol. I, 1994, pp. 270–277.
31. D. L. Parnas and J. Madey, Functional documentation for computer systems, *Science of Computer Programming* **25**(1) (1995) 41–61.
32. D. L. Parnas, J. Madey and M. Iglewski, Precise documentation of well-structured programs, *IEEE Trans. Software Engineering* **20**(12) (1994) 948–976.
33. D. K. Peters and D. L. Parnas, Using test oracles generated from program documentation, *IEEE Trans. Software Engineering* **24**(3) (1998) 161–173.
34. 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*, 2006, p. 50.
35. D. J. Richardson, S. L. Aha and T. O. O'Malley, Specification-based test oracles for reactive systems, in *Proc. Int'l Conf. Software Eng. (ICSE)*, 1992, pp. 105–118.
36. D. S. Rosenblum, A practical approach to programming with assertions, *IEEE Trans. Software Engineering* **21**(1) (1995) 19–31.
37. D. Saff, Theory-infected: Or how I learned to stop worrying and love universal quantification, in *Companion to Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, 2007, pp. 846–847.
38. P. A. Stocks and D. A. Carrington, Test templates: A specification-based testing framework, in E. Straub (editor), *Proc. Int'l Conf. Software Eng. (ICSE)*, 1993, pp. 405–414.
39. Y. Wang, *Specifying and Simulating the Externally Observable Behavior of Modules*, PhD thesis, Dept. of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1994.