Tool Support for Test-Driven Development Using Formal Specifications

Shadi Alawneh and Dennis Peters
Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University
{*shadi.alawneh, dpeters*}*@mun.ca*

Abstract— **This paper describes how Test-Driven Development (TDD) can be conducted using formal specifications with appropriate tool support. In TDD, the test code is a formal documentation of the required behaviour of the component or system that is being developed, but this documentation is necessarily incomplete and often over-specific. We propose 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. This approach has the advantage that the specifications can be complete and appropriately abstract but still support TDD.**

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

I. Introduction

Test-Driven Development (TDD) is a software development technique that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards. The steps of test-driven development (TDD) are illustrated in the UML activity diagram of Figure 1. TDD is one of the core practices of Extreme Programming (XP)[1], [2]. 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 behaviour 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 behaviour 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 behaviour 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 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.

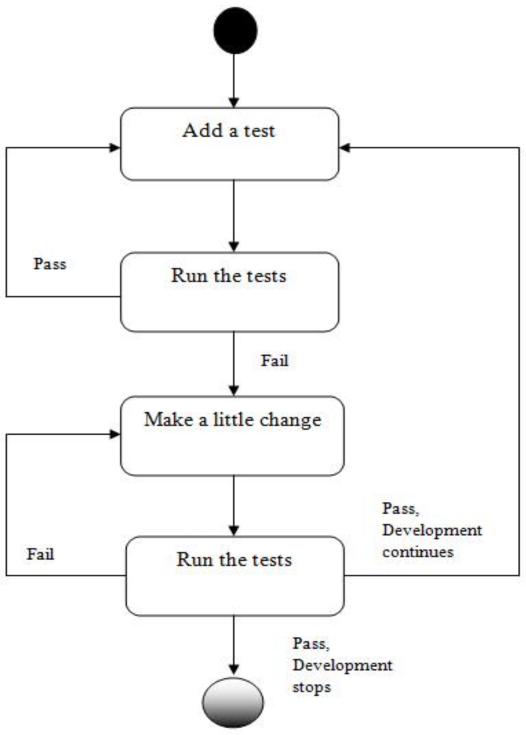

Fig. 1. The Steps of test-driven development (TDD)[3]

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’.

II. 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 [4], in this work, we are focused on deriving test oracles from the module internal design document [5]. 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 [6], [7], [8]. 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 [7], [8]. 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.

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

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

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

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.

A. 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 '$\mathtt{i} > 0 \wedge \mathtt{j} > 0$', 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.

Program Specification

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

	$\mathtt{i} > 0 \wedge \mathtt{j} > 0$	$\mathtt{i} \le 0 \vee \mathtt{j} \le 0$
`gcdvalue` =	$\mathrm{gcd}(\mathtt{i},\mathtt{j})$	0
`result` =	`TRUE`	`FALSE`

Auxiliary Function Definitions

Integer $\mathrm{gcd}($**Integer** a, **Integer** $\mathrm{b})$

$$\overset{\text{df}}{=}$$

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

Fig. 2. Ggcd Program Specification

III. Tool Support

A. OMDoc Document Model

As described in [9], 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. 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.

B. 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.

C. Specification Editor

As a part of our tools, we are developing a specification editor to support production of software documents. 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.

IV. Oracle Generation

As described in [10], 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.

The second author has previously addressed this problem together with David Parnas [11]. That work 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 parts 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 prototype test oracle generator they used 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.

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 can 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.
- 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'. 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 behaviour 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), and object classes (e.g., Integer Interval.Java, InvertedTable.Java, NormalTable.Java and VectorTable.Java), previously manually implemented and are 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.

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 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.

V. Test Driven Development With Oracles

This section describes our new approach for TDD. It also describes an example which shows how to apply this approach.

The process is as described below:

- 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.

The illustration of TDD provided in [12], [13], in which a program is developed to convert decimal numbers into their roman numeral equivalent, serves as a good, although somewhat simplistic, illustration of our method.

Here, we 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 behaviour. So, we have started with this specification:

String `dToR`(**Integer** `i`)
$\overset{\text{df}}{=}$

	$\mathtt{i} = 1$	$\mathtt{i} = 2$	$\mathtt{i} = 3$
`result` =	"`I`"	"`II`"	"`III`"

The above specification shows the definition for `dToR(i)` function which represents the program function. In program function definitions, we use the convention that `result` represents the value returned by the function. The required behaviour that is represented by this specification is to support the conversion of numbers (1–3) into their corresponding roman numerals (I, II, III).

After we write the specifications, we generate the test oracle from it and we 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 yet implemented the program. We then implement enough of the program to make the cases pass.

The previous specifications only specifies a behavior for numbers in the range 1–3, so if a test case outside that range is used then the test oracle will give an error that says "NoSuchElementException".

The pattern used in the previous specification (i.e., explicitly specifying the corresponding roman numeral representation for each decimal number) is clearly not practical for a very broad range of inputs. We can re-write the previous specification, as follows (where "+" on **String**s is used to represent concatenation):

String `dToR`(**Integer** `i`)
$\overset{\text{df}}{=}$

	$\mathtt{i} \geq 1 \wedge \mathtt{i} < 4$
`result` =	$\mathrm{subDToR}(\mathtt{i})$

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

$\mathrm{i} > 0 \wedge \mathrm{i} < 4$	"I" + subDToR(i − 1)
$\mathrm{i} = 0$	""

Then we can broaden the domain of the previous specification:

String `dToR`(**Integer** `i`)
$\overset{\text{df}}{=}$

	$\mathtt{i} \geq 1 \wedge \mathtt{i} < 5$	$\mathtt{i} \geq 5 \vee \mathtt{i} < 1$
`result` =	$\mathrm{subDToR}(\mathtt{i})$	"`NA`"

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

$\mathrm{i} = 4$	"IV"
$\mathrm{i} > 0 \wedge \mathrm{i} < 4$	"I" + subDToR(i − 1)
$\mathrm{i} = 0$	""

Now, the specification defines the conversion of numbers from (1–4) into their corresponding roman numerals (I, II, III, IV) and handles the error where subDToR is not defined by specifying the behavior for those inputs. After we have refined the initial specification, we do the same steps as we did in the previous one. Again we refine the implementation until the behavior is consistent with the specification, then continue to revise the specification, as follows.

String `dToR`(**Integer** `i`)
$\overset{\text{df}}{=}$

	$\mathtt{i} \geq 1 \wedge \mathtt{i} < 9$	$\mathtt{i} \geq 9 \vee \mathtt{i} < 1$
`result` =	$\mathrm{subDToR}(\mathtt{i})$	"`NA`"

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

$\mathrm{i} \geq 5 \wedge \mathrm{i} < 9$	"V" + subDToR(i − 5)
$\mathrm{i} = 4$	"IV"
$\mathrm{i} > 0 \wedge \mathrm{i} < 4$	"I" + subDToR(i − 1)
$\mathrm{i} = 0$	""

The specification defines behaviour for the conversion of numbers from (1–8) into their corresponding roman numerals (I, II, III, IV, V, VI, VII, VIII). We do the same steps as before and after that, we continue to revise the specification, as follows.

String `dToR`(**Integer** `i`)
$\overset{\text{df}}{=}$

	$\mathtt{i} \geq 1 \wedge \mathtt{i} < 10$	$\mathtt{i} \geq 10 \vee \mathtt{i} < 1$
`result` =	$\mathrm{subDToR}(\mathtt{i})$	"`NA`"

String subDToR(**Integer** i)

$\stackrel{\text{df}}{=}$

$i = 9$	"IX"
$i \geq 5 \wedge i < 9$	"V" $+$ subDToR$(i - 5)$
$i = 4$	"IV"
$i > 0 \wedge i < 4$	"I" $+$ subDToR$(i - 1)$
$i = 0$	""

Now, the specification defines the conversion of numbers from (1–9) into their corresponding roman numerals (I, II, III, IV, V, VI, VII, VIII, IX). So, in every step we revise the specification to describe new behavior and the specification is represented in a formal way. Also, if the tests fail after we revise the specification we have to write some code to satisfy the specification, and after that we continue to revise the specification.

We keep doing this process until the specification is complete and the code behavior is consistent with the required behavior that is described by the specification. After we have done several steps using our TDD approach to develop the specification and code together, the complete specification is as follows.

String `dToR`(**Integer** `i`)

$\stackrel{\text{df}}{=}$

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

String subDToR(**Integer** i)

$\stackrel{\text{df}}{=}$

$i \geq 1000$	"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$(i - 1)$
$i = 0$	""

Now, we have a complete specification that describes the whole required behavior for the program, and presumably the working implementation developed along with it using TDD. So, using our approach results in a complete specification, implementation and suite of test cases for the program.

VI. Conclusion

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 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 behaviour 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.

VII. 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.

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).

VIII. 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).

References

[1] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

[2] Ron Jeffries, Ann Anderson, and Chet Hendrickson, *Extreme Programming Installed*, Addison-Wesley, 2001.

[3] Scott Ambler, *Agile Database Techniques:Effective Strategies for the Agile Software Developer*, Wiley Publishing, United States of America, 2003.

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

[5] 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.

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

[7] 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.

[8] 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.

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

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

[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.

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

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