How can non-technical end users effectively test their spreadsheets?

Pak-Lok Poon [1], Fei-Ching Kuo [2],Huai Liu [2], Tsong Yueh Chen [2]

1 School of Accounting and Finance, The Hong Kong Polytechnic University, Hong Kong

2 Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia;

Abstract

Purpose – An alarming number of spreadsheet faults have been reported in the literature, indicating that effective and easy-to-apply spreadsheet testing techniques are not available for "non-technical", end-user programmers. This paper aims to alleviate the problem by introducing a metamorphic testing (MT) technique for spreadsheets.

Design/methodology/approach – The paper discussed four common challenges encountered by end-user programmers when testing a spreadsheet. The MT technique was then discussed and how it could be used to solve the common challenges was explained. An experiment involving several "real-world" spreadsheets was performed to determine the viability and effectiveness of MT.

Findings – Our experiment confirmed that MT is highly effective in spreadsheet fault detection, and yet MT is a general technique that can be easily used by end-user programmers to test a large variety of spreadsheet applications.

Originality/value – The paper provides a detailed discussion of some common challenges of spreadsheet testing encountered by end-user programmers. To our best knowledge, the paper is the first that includes an empirical study of how effective MT is in spreadsheet fault detection from an end-user programmer's perspective.

Keywords End user computing, Decision support system, Metamorphic testing, Oracle problem

1. Introduction

Software development involves a series of production activities in which there are many opportunities to make mistakes (Boehm & Basili, 2001). Following the advent of PCs in the 1980s and the proliferation of end-user computing (Peng *et al.*, 2011; Taylor *et al.*, 1998), software development shifted from being something that only well-trained IT professionals could do, to something millions of departmental end users (hereafter referred to as *end-user programmers*) were responsible for. Many of these end users had little formal training in software development (Gripenberg, 2011). Scaffidi *et al.* (2005) report that the number of end-user programmers in the U.S. will increase to 90 million by 2012. This may explain why there are so many faulty end-user applications flooding into our society.

Spreadsheet-based systems, or simply spreadsheets, play an important part in end-user computing, and they are ubiquitous in many business activities such as accounting and financial reporting, asset recording, production scheduling, and engineering design (Mason & Willcocks, 1991; McDaid & Rust, 2009). Spreadsheets have become an essential tool for assisting high-stake management decisions in many types of organizations (Caulkins *et al.*, 2007).

In spite of the popularity and importance of spreadsheets, an alarming number of spreadsheet faults have been reported in the literature (McDaid & Rust, 2009; Panko, 1998; Rajalingham *et al.*, 2000). A study by McDaid and Rust (2009) has found that among 50 operational spreadsheets used in industry, 94% contained faults, with almost 1% of formula cells found to be incorrect. These spreadsheet faults indicate that there are not yet sufficient spreadsheet testing techniques which are effective and easy to apply for non-technical, end-user programmers.

To alleviate this problem, we introduce a metamorphic testing (MT) technique for spreadsheet fault detection, which can be effectively applied by end-user programmers with little software development training. Another advantage of this MT technique is that it can be used to test the specific properties of the application domain (e.g., accounting and finance, and production scheduling) of a spreadsheet. To our best knowledge, this paper is the first that includes an empirical study on how effective MT is in spreadsheet fault detection from an end-user programmer's perspective.

2. Organizational Risks Associated with Developing or Testing End-User Applications

Because spreadsheets are frequently encountered in end-user computing, we first present some major organizational risks associated with developing or testing end-user applications based on our literature review. These risks are listed below:

(a) **Poor coordination among individual end-user programmers.** The risk occurs when an end-user programmer does not know what the others have developed, or develops application-specific rather than generic models (Alavi & Weiss, 1989; Taylor *et al.*, 1998). Poor coordination prevents these programmers from getting a synergy effect of a team-based development/testing approach.[1]

(b) **Insufficient technical support on development and testing.** Lee *et al.* (1995) found that end-user training on software development and testing is generally weak within organizations. Furthermore, if end-user programmers encounter problems in development and testing, little technical support is offered by the internal IT Department. This causes the developed end-user applications to be inefficient and possibly faulty.

(c) **Little or no documentation.** End-user programmers typically avoid documentation because they consider it as a waste of time and a non-essential task (Caulkins *et al.*, 2007; Laudon & Laudon, 2010). Documentation is, however, important to reviews and future system maintainability (Pierson *et al.*, 1990). Based on a survey by Benson (1983), numerous end-user programmers admitted that if they quit the firm, then no one else could maintain their applications.

(d) **Lack of extensive testing.** Serious problems can occur when end-user programmers assume, after limited test, that their applications are free of faults and, hence, are ready for production use (Caulkins *et al.*, 2007; Laudon & Laudon, 2010). Two plausible reasons for the lack of extensive testing are that end-user programmers: (i) are unaware of the many possibilities for introducing software faults and, hence, are overconfident on the correctness of their applications (Barr *et al.*, 1994; Panko, 2009); and (ii) do not have good knowledge on the relevant testing techniques or guidelines (Cale, 1994; Harrison, 2004).

As stated in the Introduction, the main theme of this paper is to introduce an effective spreadsheet testing technique to end-user programmers, which is directly related to organizational risk (d) above. In Section 4, we will express organizational risk (d) in terms of four challenges to spreadsheet testers. We will then discuss how MT deals with these

challenges in Sections 5 to 7. Later in Section 8, we will discuss how organizational risk (a) can be minimized in MT, and how MT alleviates organizational risks (b) and (c).

3. Preliminaries

We first outline the important concepts that are essential for understanding MT. A *failure* is an "observed" malfunction of a program, which is caused by a *fault* in that program, which in turn is caused by a human *mistake* (or simply *mistake*) (IEEE, 1990).[2] Consider, for example, a spreadsheet formula which calculates the area of a rectangle given its length x (= 3) and its width y (= 4). The values of x and y are stored in the cells A1 and A2, respectively. The correct formula should be "= A1 * A2", and the correct output should be 12. Suppose the actual formula defined is "= A1 + A2" and, hence, the actual output is 7. In this case, the misuse of the arithmetic operator "+" (instead of "*") in the formula represents a fault, and the difference between the expected result (= 12) and the actual result (= 7) represents a failure. Unless we discuss with the spreadsheet developer, it is impossible to determine the underlying reason (e.g., a typing mistake or a wrong formula in the developer's mind) for making this mistake.

Testing is a verification technique used in software development, and is often categorized as being either dynamic or static. *Dynamic testing* involves *executing* the software system with test data, and then checking the output and the operational behavior of the software (Sommerville, 2011). *Static testing*, also known as *human testing*, however, does not involve software execution. Reviews, inspections, and audits are examples of static testing (IEEE, 2008; Myers, 2004). Neither dynamic testing nor static testing is considered sufficient on its own: whenever possible and applicable, it is recommended that both be used (Franz & Shih, 1994; Kandt, 2009).

The MT technique presented in this paper belongs to the paradigm of *dynamic* testing, and its main aim is to identify spreadsheet *faults*. In the rest of this paper, unless stated otherwise, dynamic testing is referred to simply as "testing".

4. Challenges to spreadsheet testers

Refer to the two plausible reasons for the lack of extensive testing as presented in organizational risk (d) in Section 2. For the first reason related to the overconfidence of end-user programmers on the correctness of their spreadsheets, Panko (2009) suggests to educate the end-user programmers on the high number of spreadsheet faults (McDaid & Rust, 2009; Panko, 1998; Rajalingham *et al.*, 2000), so that they can pay serious attention to "comprehensive" spreadsheet testing. For the second reason related to the lack of knowledge on the relevant testing techniques and guidelines, we argue that this problem is mainly caused by the absence of an appropriate testing technique for spreadsheets. This issue is reframed in terms of the following four challenges.

Challenge 1: small number of appropriate testing techniques and the required technical knowledge for their use

Until now, the only testing techniques specifically developed for spreadsheets (Ayalew, 2007) are the constraint-based spreadsheet testing method (Abraham & Erwig, 2006) and the "What You See Is What You Test" (WYSIWYT) methodology (Fisher *et al.*, 2006). The constraint-based spreadsheet testing method is developed with an implicit assumption that the testers are formally trained in software development and testing. Its target users are not end-user programmers. On the other hand, although the WYSIWYT methodology is developed for end-user programmers, it requires the testers to have some technical knowledge of data-flow adequacy criteria (Jee *et al.*, 2009) and coverage monitoring (Vilkomir *et al.*, 2003).

Challenge 2: inaccessibility to the associated automated tools

A core issue of testing is the trade-off between the comprehensiveness of the set of test cases (*T*) used for testing and the effort required for its generation and execution. Intuitively, a comprehensive *T* involving a large number of test cases is desirable because of its higher chance to detect faults. Manually generating and executing such a *T*, however, may not be feasible due to the enormous human effort required. In the absence of automated tools, the large amount of testing effort might be a great deterrent to users who wish to test their spreadsheets. This explains why both the constraint-based spreadsheet testing method and the WYSIWYT methodology mentioned above require the support of associated tools. This requirement poses great difficulty to end-user programmers because the associated automated tools are not easily accessible, and may be platform-dependent. It is also unrealistic for the end-user programmers themselves to implement such tools because such implementation often requires an in-depth understanding of the principles underlying the testing methods (Ayalew, 2007). For example, in addition to general computer programming skills, implementing the tool for the WYSIWYT methodology requires the knowledge of data-flow adequacy criteria (Jee *et al.*, 2009) and coverage monitoring (Vilkomir *et al.*, 2003).

Challenge 3: lack of specific focus on the intrinsic properties of application

Consider, for example, a trial-balance spreadsheet containing a list of beginning and ending balances for all accounts recording financial transactions. In accounting, a domain-specific property of a trial balance is that the total debits for a defined period must be in balance with the total credits. A difference between the two totals indicates the occurrence of a double-entry posting fault in the trial-balance spreadsheet.

Most testing techniques, such as the classification-tree method (Grochtmann & Grimm, 1993), combinatorial testing (Lei *et al.*, 2007), equivalence partitioning (Myers, 2004), and boundary value analysis (Jorgenson, 2008), do not explicitly consider the application domains (e.g., financial reporting and production scheduling) of the spreadsheets. In other words, these techniques are rather generic and, hence, may be ineffective in testing applications with domain-specific properties. (Note that these testing techniques are also not specifically developed for spreadsheet testing.) This problem also occurs in the constraint-based spreadsheet testing method and the WYSIWYT methodology mentioned above.

Challenge 4: the oracle problem

A well-known and commonly occurred problem of testing software applications, including spreadsheets, relates to the difficulty in determining the correctness of the system output. This situation is called the *oracle problem*, and is often associated with spreadsheet testing (Ayalew, 2007; Grossman & Özlük, 2010; Panko, 1998, 1999, 2006; Panko & Aurigemma, 2010; Pryor, 2004). Section 5 below contains a detailed discussion of this problem.

In view of the four challenges highlighted above, it is desirable to have a spreadsheet testing technique that caters for the needs of end-user programmers. This technique should be easy to automate, and should not require end-user programmers to have substantial technical knowledge. Furthermore, the technique should support testing the domain-specific properties associated with the spreadsheet, and should still be applicable even when the oracle problem exists.

5. Oracle problem in spreadsheet testing

Among the four challenges highlighted in Section 4, the oracle problem is possibly the most important, because it will seriously affect the effectiveness of testing. Thus, we discuss this challenge in more details in this section.

In software testing, a *test oracle* (or simply an *oracle*) refers to a procedure or a mechanism by which the tester can verify the correctness of the system output (Chen *et al.*, 2003). In this regard, the oracle is of utmost importance in testing. The *oracle problem* exists when either one of the following scenarios applies: **(Scenario 1)** An oracle does not exist for the tester to verify the correctness of the computed output. **(Scenario 2)** An oracle exists but it is not feasible to apply.

The frequent occurrence and severity of the oracle problem in spreadsheet testing (regardless of the spreadsheet types) have been confirmed by various spreadsheet researchers and practitioners (Ayalew, 2007; Grossman, 2003; Grossman & Özlük, 2010; Panko, 1998, 1999, 2006; Panko & Aurigemma, 2010; Pryor, 2004). Below we summarize some of their arguments:

> "*Such independent calculations* [*an oracle*] *are, in my experience, rarely available and so full* [*spreadsheet*] *system testing is not often performed.*" (Pryor, 2004)

> "*In most cases, there was no comparable calculation* [*an oracle*] *before spreadsheets. In nearly all spreadsheets, calculations are extended well beyond what had been done previously with manually calculations. In complex spreadsheets, then, there usually is no oracle other than the spreadsheet calculations, which may not be correct … This lack of a readily-found oracle is the most serious problem in spreadsheet execution testing. Without a strong and easy-to-apply oracle, execution testing simply makes no sense for error-reduction testing.*" (Panko, 2006)

> "*In the event that the correct outputs* [*the oracle*] *are not knowable, testing is of little value. For this reason software testing is not applicable to a large class of scientific and business models, including large financial planning models, because the only calculation of the outputs is computed by the software* [*spreadsheet*] *being tested.*" (Grossman & Özlük, 2010)

Example 1 below illustrates the oracle problem in detail.

Example 1 (Non-profitable stocks): Table I shows part of the inventory spreadsheet in a supermarket. In this spreadsheet, each row contains information about an item such as item number and description, unit cost, unit price, unit profit (= unit price – unit cost), stock-on-hand, stock-on-order, and a stock movement indicator. The value of the stock movement indicator is either "S" or "F". The value "S" indicates that the item is "slow-moving". On the other hand, the value "F" indicates that the item is "fast-moving". (The actual details of the formula to determine whether an item is slow-moving or fast-moving need not concern us.) Suppose there are 5000 items for sale in the supermarket. Thus, the size of the spreadsheet is very large.

	A	B	C	D	E	F	G	H	I
1	**Item number**	**Item description**	**Unit cost ($)**	**Unit price ($)**	**Unit profit ($)**	**Stock-on-hand**	**Stock-on-order**	**Stock movement indicator**	**…**
2	A0001	Mug	22.00	30.00	8.00	66	250	F	:
3	A0002	Toothbrush	13.35	16.50	3.15	176	80	S	:
4	A0003	Toothpaste	14.25	17.50	3.25	395	0	S	:
5	A0004	Shaver	13.10	13.10	0.00	38	600	F	:
6	A0005	Tissue pack	6.45	5.90	– 0.55	90	20	S	:

7	A0006	Cognac	125.50	210.00	84.50	3228	2000	F	:
:	:	:	:	:	:	:	:	:	:
5001	W0801	Vegetable oil	3.30	6.10	2.80	59	100	F	:

Table I. Inventory spreadsheet

In the supermarket, the unit profits of most of its items are greater than zero (i.e., *profitable*). However, in some rare cases, the unit profit of an item may be zero or negative (i.e., *non-profitable*). For instance, a supermarket may lower the unit price to or below the unit cost of a slow-moving item for stock clearance (e.g., item "A0005" in Table I).

Usually, a fast-moving item is profitable. A fast-moving, non-profitable item warrants investigation to determine whether the unit price has been set correctly. To calculate the number of fast-moving, non-profitable items, the formula SUMPRODUCT(--(H2:H5001 = "F"), --(E2:E5001 <= 0))[3] should be used (note that the 5000 items are stored in rows 2 to 5001 in the inventory spreadsheet).

Assume that the end-user programmer made a mistake; a different formula SUMPRODUCT(--(H2:H5001 = "S"), --(E2:E5001 = 0)) was used instead. Note the wrong operand ("S" instead of "F") and the wrong relational operator ("=" instead of "<=") in the second formula. Since the SUMPRODUCT function only outputs a numeric count as the result and the number of items (= 5000) in the supermarket is huge, it is not practical to perform manual counting for verifying the correctness of the result generated by the SUMPRODUCT function. In other words, Scenario 2 happens, resulting in the occurrence of the oracle problem. ■

The above example corresponds to Scenario 2 of the oracle problem. An example of Scenario 1 of the oracle problem is testing the built-in SIN function in a spreadsheet as discussed in Section 6.1 below.

6. Metamorphic testing (MT)

MT was originally developed by Chen *et al.* (1998, 2003). Since then, it has been successfully applied to various application domains and platforms such as bioinformatics (Chen *et al.*, 2009), bioimaging (Ding *et al.*, 2010), Web search engines (Zhou *et al.*, 2012), machine learning (Xie *et al.*, 2011), feature models (Segura *et al.*, 2011), and context-sensitive middleware-based applications (Chan *et al.*, 2005).

We observe that end-user programmers can make use of MT to solve the four challenges mentioned above when testing a spreadsheet. This section outlines the concept of MT and explains how it can be effectively applied to test spreadsheets by end-user programmers even when the oracle problem exists.

6.1 Overview of MT

Consider a program P, with D as its set of all possible test cases (or inputs). Very often, the size of D is huge or even infinite. Limited by resource constraints, we use a subset of D as the set of test cases (T) for testing P, that is, $T = \{t_1, t_2, \ldots, t_n\} \subset D$, where t_i is a test case for any $i = 1, 2, ..., n$. The value of n may depend on the amount of testing resources available. The more testing resources one has, the higher the value of n that might be chosen. Executing P with every test case $t_i \in T$ yields the results $P(t_1)$, $P(t_2)$, …, $P(t_n)$. In the presence of the oracle problem, these actual test results cannot be feasibly verified for their correctness.

MT is an innovative technique to alleviate the oracle problem in testing. The technique uses some specific properties of the implemented system to form their corresponding

metamorphic relations (*MR*). Such relations are then used to generate a new set of test cases (T') for testing, and to verify the test results.

Consider, for instance, a trigonometric function $F(x)=sin(x)$ and its corresponding built-in `SIN` function in a spreadsheet. Most people simply use the built-in `SIN` function by *assuming* that all its returned values are correct. However, if we decide to test this built-in function, we will encounter the oracle problem because the exact value of $sin(x)$ is unknown for most values of x (corresponds to Scenario 1 of the oracle problem mentioned in Section 5).[4] A property, or its corresponding MR, of $F(x)$ is $sin(x) = sin(x + 360°)$, where x is any angle in the unit of degree (the rest of the paper follows this notation). Suppose we execute the built-in `SIN` function with a test case $x = 49°$, and obtain an output of 0.7547, that is, `SIN`(49°) = 0.7547 (note again that the *exact* value of $sin(49°)$ is unknown, and "0.7547" is only an *approximate* value rounded up to four decimal places). In MT, we should next execute the `SIN` function with another test case $x' = 49° + 360° = 409°$. After the second execution, we should then check whether `SIN`(409°) = `SIN`(49°) = 0.7547, after allowing for rounding "error". If the identity does not hold, then a failure is revealed.

For ease of discussion, we will refer to the test cases used in the first execution of the program, such as x above, as *source* test cases; and the subsequent test cases generated in accordance with an MR, such as x' above, as *follow-up* test cases.

The above example shows that MT checks whether the identified metamorphic relations hold among *several* executions rather than focusing on the correctness of outputs from *individual* executions (which require the exact output values to be known). It is this characteristic of MT which makes the technique applicable to test software systems with the oracle problem. Furthermore, checking the fulfillment of metamorphic relations can be largely automated and, hence, a large amount of testing resources can be saved. Thus, MT is a simple-to-apply and yet effective testing technique.

Readers may note that the use of trigonometric functions (such as *sine*, *cosine*, and *tangent*) is fairly popular in many engineering spreadsheets, including those which are implemented for modeling circuits and other linear systems (Caulkins *et al.*, 2007; Christy, 2006; Dewey, 1998; Doll, 2000; Rahuma *et al.*, 2013). Some of these engineering spreadsheets are "mission-critical" and their correctness may even significantly affect human life. For example, McDonough (2004) reports that a nuclear plant expert system has been developed for Electricité de France (EdF), which is a national energy company headquartered in Paris. A main function of this system is to provide diagnosis of pipes in a nuclear plant. EdF has decided to build the model used in the expert system in the Excel spreadsheet platform. MT will be very useful for testing these engineering spreadsheets, which are often associated with the oracle problem (Scenario 1) during testing.

6.2 Applying MT to spreadsheet testing

Below we illustrate how to apply MT to test the spreadsheets with the oracle problem as presented in Example 1 above.

Every item in the supermarket can be classified into one of the following four groups:

(i) slow-moving and profitable items (stock movement indicator = "S" and unit profit > 0),
(ii) slow-moving and non-profitable items (stock movement indicator = "S" and unit profit ≤ 0),
(iii) fast-moving and profitable items (stock movement indicator = "F" and unit profit > 0), and
(iv) fast-moving and non-profitable items (stock movement indicator = "F" and unit profit ≤ 0),

where each of the four groups may be empty.

Metamorphic relations, such as the following, can be used to test the correctness of the formula for counting the number of fast-moving, non-profitable items: **(MR_1)** If we insert a row (between the existing row 2 and row 5001) into the spreadsheet for storing a new slow-moving and non-profitable item, followed by setting the stock movement indicators of all items to "F", then the output value of the corresponding variant formula will increase and must be greater than zero. **(MR_2)** If we insert a row (between the existing row 2 and row 5001) into the spreadsheet for storing a new fast-moving and profitable item, followed by setting the unit prices of all items to their unit costs so that all the unit profits become zero, then the output value of the corresponding variant formula will increase and must be greater than zero. **(MR_3)** If we insert *m* rows (between the existing row 2 and row 5001) into the spreadsheet for storing *m* new items so that all these items are fast-moving and non-profitable, then the output value of the corresponding variant formula will increase by *m* and must be greater than zero (see Note 5 for the details of variant formulae and variant criteria).

If the correct criterion (--(H2:H5001 = "F"), --(E2:E5001 <= 0)) or any of its variant criteria is used for the SUMPRODUCT function, only items in group (iv) will be counted by the formula SUMPRODUCT(--(H2:H5001 = "F"), --(E2:E5001 <= 0)) and any of its variant formulae (the "correct (variant) formula(e)"). Suppose, according to MR_1, we add a slow-moving and non-profitable item to the spreadsheet to guarantee that group (ii) is nonempty, followed by setting the stock movement indicators of all items to "F", then the item(s) previously in group (ii) will now move to group (iv). This will cause the output value of the correct variant formula (that is, SUMPRODUCT(--(H2:H5002 = "F"), --(E2:E5002 <= 0))) to increase and that value must be greater than zero. Similarly, with respect to MR_2, suppose we add a fast-moving and profitable item to the spreadsheet to guarantee that group (iii) is nonempty, followed by making the unit profits of all items to zero, then the item(s) previously in group (iii) will now move to group (iv). This will cause the output value of the correct variant formula to increase and that value must be greater than zero. The rationale of MR_3 is obvious and needs no further explanation.

Recall that the end-user programmer has used the wrong criterion (--(H2:H5001 = "S"), --(E2:E5001 = 0)) for the SUMPRODUCT function. Consider MR_1 first. Because of the first part of the wrong criterion (--(H2:H5001 = "S")), adding an item to group (ii) followed by setting the stock movement indicators of all items to "F" will cause the "incorrect variant formula" (that is, SUMPRODUCT(--(H2:H5002 = "S"), --(E2:E5002 = 0))) to output a zero value. (Note the use of the new variant criterion in this incorrect variant formula. See Note 5 for the details.) This violates MR_1, indicating that this metamorphic relation guarantees to detect the fault.

Now we turn to MR_2. Table II(a) classifies all the items into six categories, with respect to their stock movement indicators and unit profits. Table II(a) also shows that only slow-moving items with zero unit profit will be counted by the incorrect formula and its variant formulae. Table II(b) shows that, after setting all unit profits to zero, all slow-moving items will be counted by the incorrect formula and its variant formulae. Since group (i), group (ii), or both may be empty or nonempty, the output value of the incorrect formula and its variant formulae may increase or remain unchanged and that value may be zero or positive. In other words, MR_2 cannot guarantee to detect the fault.

(a)

Before setting all unit profits to zero	Slow-moving	Fast-moving
Unit profit < $0.00	× Group (ii)	× Group (iv)
Unit profit = $0.00	√ Group (ii)	× Group (iv)
Unit profit > $0.00	× Group (i)	× Group (iii)

(b)

Before setting all unit profits to zero		*After* setting all unit profits to zero	Slow-moving	Fast-moving
Unit profit < $0.00	→	**Unit profit = $0.00**	√	×
Unit profit = $0.00	→	**Unit profit = $0.00**	√	×
Unit profit > $0.00	→	**Unit profit = $0.00**	√	×

(√) Items in this category will be counted by SUMPRODUCT with the *wrong* criterion (--(H2:H5001 = "S"), --(E2:E5001 = 0)) or any of its variant criteria

(×) Items in this category will *not* be counted by SUMPRODUCT with the *wrong* criterion (--(H2:H5001 = "S"), --(E2:E5001 = 0)) or any of its variant criteria

Table II. Effect of setting the unit profits to zero on the output value of SUMPRODUCT with the *wrong* criterion (--(H2:H5001 = "S"), --(E2:E5001 = 0)) or any of its variant criteria

Finally, we consider MR_3. Because the incorrect formula and its variant formulae only count slow-moving items with zero profit, adding more fast-moving and non-profitable items will have no effect on the output value of these incorrect formulae. This violates MR_6, indicating that this metamorphic relation guarantees to detect the fault.

The above discussion shows that MT fulfils the requirements for effective spreadsheet testing by end-user programmers: (i) there is no requirement for users to have substantial technical knowledge (Challenge 1); (ii) the technique can easily be automated, even by non-technical users who can easily write their own testing scripts (Challenge 2); (iii) there is more focus on testing the domain-specific properties associated with the spreadsheet applications (most of the metamorphic relations listed in this section are domain-specific) (Challenge 3); and (iv) it is possible to test spreadsheets even when the oracle problem exists (Challenge 4). Note that, with regard to requirement (iii) above, because end-user programmers are the spreadsheet developers, they should have a good knowledge of the application domains. Thus, they should be able to identify domain-specific metamorphic relations. Table III summarizes the four challenges in spreadsheet testing discussed in Section 4 and how MT can be used to tackle them.

Challenges to spreadsheet testers	How MT solves these challenges
Challenge 1: Small number of appropriate testing techniques and required technical knowledge for their use	MT adds to the pool of existing few testing techniques for spreadsheets. MT has demonstrated promising effectiveness in detecting spreadsheet faults (see Section 7.2 for more details). Also, end-user programmers are not required to have substantial technical knowledge for using MT.
Challenge 2: Inaccessibility to the associated automated tools	To large extent, MT can be easily automated by end-user programmers themselves (see Section 7.3 for more details).
Challenge 3: Lack of specific focus on the intrinsic properties of application	Metamorphic relations can be identified based on the intrinsic properties of the spreadsheet application under test (see, for example, MR_1 to MR_3 in Section 6.2).
Challenge 4: Oracle problem	MT is effective for testing software applications with the presence of the oracle problem.

Table III. Spreadsheet testing challenges and how they are dealt with by MT

7. Experiment

Meyer (2008) argues that "a successful test is only relevant to quality assessment if it previously failed; then it shows the removal of a failure and usually of a *fault* … This keeps the testing process focused: Its single goal is to *uncover faults* by triggering failures." Meyer's argument is supported by many software researchers and practitioners such as Myers (2004) and Kaner *et al.* (1999). In other words, the ultimate goal of testing is to detect faults. Because of this, we evaluate the effectiveness of MT in terms of its ability in uncovering different types of spreadsheet faults from an end-user programmer's perspective.

Figure 1 outlines the major phases and steps of the experiment (phases are indicated in capital letters at the left side of the figure and steps are enclosed in boxes). Details of these phases and steps are described below.

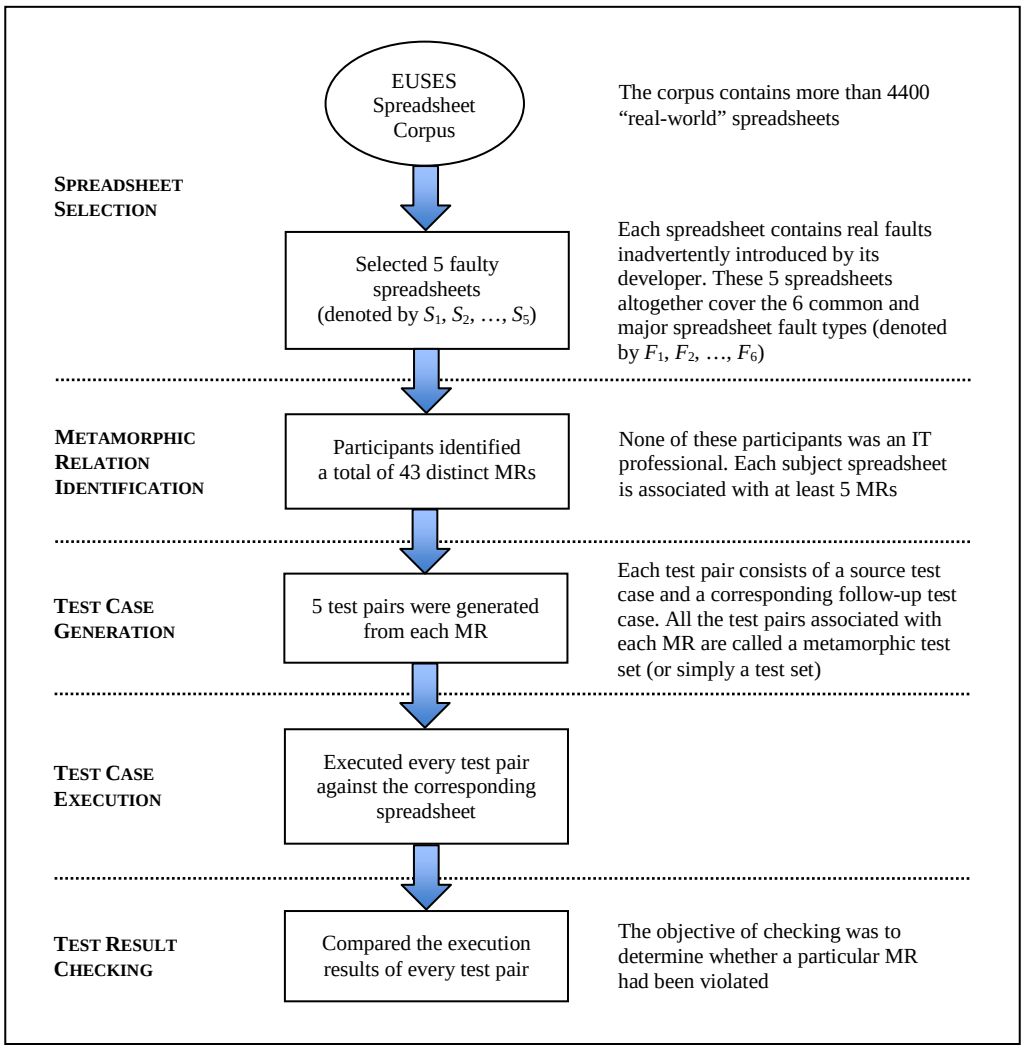

Figure 1. Major phases and steps of the experiment

7.1 Spreadsheets and participants

For our experiment, we used five spreadsheets with natural faults (i.e., real faults inadvertently introduced by the developers) from the EUSES Spreadsheet Corpus (Fisher & Rothermel, 2005).[6] We selected these five spreadsheets (denoted by S_1, S_2, …, and S_5, respectively) so that they altogether covered the six common and major spreadsheet fault types (see Table IV) according to Ronen's classification scheme (Ronen *et al.*, 1989).[7, 8]

Each of the five selected spreadsheets was developed for a different application domain. Table V gives some details about these spreadsheets, including the types of faults that each contains.

Fault ID	Fault description
F_1	Mistakes in logic
F_2	Incorrect ranges in formulae
F_3	Incorrect cell references
F_4	Incorrectly copied formulae
F_5	Accidentally overwritten formulae
F_6	Misuse of built-in functions

Table IV. Spreadsheet fault types

Spread-sheet	Application domain	Main function	Types of faults contained[a]					
			F_1	F_2	F_3	F_4	F_5	F_6
S_1	School equipment management	Calculate the 3-year and 4-year replacement costs of equipment in the library, laboratories, and classrooms	√				√	
S_2	Stores management and stock control	Perform calculations such as average stock levels, stock safety levels, and the annual inventory projection		√				
S_3	Air quality monitoring	Compute the air quality in different cities in terms of various measurements such as the levels of carbon monoxide, sulfur dioxide, ozone, and acid rain				√		
S_4	Database performance evaluation	Evaluate the performance of university database systems in terms of ease of access, response time, connectivity, and stability						√
S_5	Household expense management and analysis	Compute and analyze the amount of money spent on various categories of household expenses			√			

[a] For each fault type, the number of faults contained in the respective spreadsheet is one

Table V. Subject spreadsheets

Five participants were recruited for the experiment. None of them was an IT graduate, nor had any been formally trained as an IT professional. All these participants, however, had practical experience in spreadsheet development as part of their job duties. Thus, they were easily classified as "non-technical" end-user programmers, and as such, could help determine the effectiveness of MT from an end-user programmer's perspective.

Before the participants identified metamorphic relations for testing the subject spreadsheets, they had been offered a one-hour training session, during which a hands-on exercise on identifying metamorphic relations from a simple spreadsheet was given. The two purposes of this training session were to teach the participants the concept of MT and how to identify metamorphic relations.

For the second purpose, we advised the participants to identify metamorphic relations using the following "output-driven" approach:

(a) Identify the outputs of the spreadsheet.
(b) For each output *O* identified in (a), determine its corresponding inputs *I*.
(c) Determine whether varying *I* in a particular way will: (i) cause *O* to change in a definite manner, or (ii) keep *O* unchanged.
(d) If the answer is "yes" in (c)(i), define a metamorphic relation in the form: "If we change *I* to *I′*, then *O* will be changed to *O′*". On the other hand, if the answer is "yes" in (c)(ii), define a metamorphic relation in the form: "If we change *I* to *I′*, then

O will remain unchanged". (*I′* and *O′* represent the new input and new output, respectively.)

The above identification approach was then used by the participants to identify metamorphic relations for the five subject spreadsheets as listed in Table V.

7.2 Testing procedures and results

Each participant spent about an hour, alone, identifying as many metamorphic relations as possible for each subject spreadsheet. In some cases, some metamorphic relations identified by individual participants were found to be identical. After tallying, we found 9, 11, 5, 5, and 13 (total = 43) distinct metamorphic relations identified for S_1, S_2, …, and S_5, respectively. In other words, on average, 8.6 ($= \frac{9+11+5+5+13}{5}$) distinct metamorphic relations were identified for each spreadsheet.

Table VI gives some statistics on the numbers of metamorphic relations identified by each participant. Considering the five spreadsheets together, on average, the percentage of the number of metamorphic relations identified by each participant with respect to the total number of *distinct* metamorphic relations identified by all the participants is 43%.

*** * * Insert Table VI here * * ***

Table VII summarizes the number of *distinct* metamorphic relations identified by one or more participants for each spreadsheet. For example, for S_1, five distinct metamorphic relations were independently identified by three participants. A close examination found that the percentages of identical metamorphic relations identified by at least two participants are 67% (= 6/9), 82% (= 9/11), 80% (= 4/5), 80% (= 4/5), and 31% (= 4/13) for S_1, S_2, …, and S_5, respectively (with an average of 63% when considering the five spreadsheets altogether). The high percentages of identical metamorphic relations (except S_5) suggest that their identification can be easily and effectively performed by end-user programmers with different backgrounds and credentials.

*** * * Insert Table VII here * * ***

For ease of discussion, we use the notation $\mathrm{MR}_{m.n}$ to denote the *n*th metamorphic relation for spreadsheet S_m, where $1 \leq m \leq 5$. For illustration, we provide here two such metamorphic relations for S_3: **($\mathbf{MR_{3.1}}$)** Cells N5–N26 are used to store the levels of PM2.5 (particulate matter of up to 2.5 microns in diameter) in different cities taken in the first time in a day. If we change the order of these cells, the average level of PM2.5 will remain unchanged. **($\mathbf{MR_{3.4}}$)** If we multiply the levels of PM2.5 in cells N5–N26 by *k* times (where $k > 0$), then we have $A' = A \times k$ (where *A* and *A′* represent the average levels of PM2.5 before and after multiplication, respectively).

For each of the 43 identified metamorphic relations, we randomly generated five source test cases. Then, for every such source test case, we generated a follow-up test case in accordance with the respective MR. Thus, five follow-up test cases were generated for each MR. Each source test case and its corresponding follow-up test case are called a *metamorphic test pair* (or simply a *test pair*). In total, there were 45, 55, 25, 25, and 65 test pairs for S_1, S_2, …, and S_5, respectively. For ease of discussion, we call *all* the test pairs associated with each MR a *metamorphic test set* (or simply a *test set*). After generating the test pairs, we executed the five spreadsheets with their respective test sets and determined whether the associated metamorphic relations were satisfied or violated.

The results of the experiment are shown in Table VIII. Overall, we found that each fault type was detected by the test set of *at least one* MR. For example, F_1 was detected by the test sets of $MR_{1.7}$, $MR_{1.8}$, and $MR_{1.9}$, while F_6 was detected by the test set of $MR_{4.2}$. Considering the five spreadsheets together, an average of about one out of three (33%) of metamorphic relations, and about one out of four (24%) of the test pairs revealed a spreadsheet fault. This is in spite of the fact that most of the metamorphic relations identified by the participants were relatively simple as indicated, for example, by $MR_{3.1}$ and $MR_{3.4}$ above. This is an encouraging result, which clearly demonstrates the effectiveness of MT in spreadsheet fault detection from an end-user programmer's perspective. Furthermore, note that none of the participants was the developer of the subject spreadsheets. If they were the developer, they should have a good knowledge of the properties of the application domain of these spreadsheets and, hence, more "fine-grained" metamorphic relations would have been identified. In turn, the experiment results would be even better.

*** * * Insert Table VIII here * * ***

A close examination of Table VIII revealed that, with respect to our experiment, "incorrectly copied formulae (F_4)" was the most easily detected fault type by MT, while "incorrect ranges in formulae (F_2)" and "incorrect cell references (F_3)" were the least easily detected ones. Here, we caution that more experiments need to be done in order to produce statistically generalizable results about the relative ease of detecting these fault types by using MT.

7.3 Experiment automation

Although MT can be applied without the support of automated tools, their use would almost certainly increase the effectiveness of testing, especially when a large number of test cases is involved.

In our experiment, we automated the last three phases (test case generation, test case execution, and test result checking) in Figure 1 as far as possible. $MR_{3.4}$ in Section 7.2 is used for illustration. In the test case generation phase, we firstly used the built-in function RAND to randomly generate the levels of PM2.5 for the cells N5–N26 as part of the source test cases. This was followed by using RAND to randomly generate a value for the multiplication factor *k*, through which the source test cases were converted into their corresponding follow-up test cases automatically (each PM2.5 level was multiplied by *k* to obtain a new PM2.5 level in accordance with $MR_{3.4}$).

Automation of the test case execution phase was implemented by simply putting a formula in each of the cells N5–N26 so that this formula referred to the location storing a new PM2.5 level. To the extent possible, we also automated the phase of test result checking. With respect to $MR_{3.4}$, suppose the cells X30 and Y30 were used to store the average levels of PM2.5 before and after multiplication by *k*, respectively. We used the formula IF(Y30 = X30 * k, "pass", "fail") to automatically determine whether $MR_{3.4}$ was violated.

We note that a higher degree of automation is possible with appropriate tool support. For example, a Java Excel API is freely available on the Internet (http://www.andykhan.com/jexcelapi) which facilitates the development of Java applications for reading in a spreadsheet, modifying some cells, and generating a new spreadsheet. This API could support the integration of the last three phases in Figure 1.

8. Alleviation of Organizational Risks in MT

In Section 2, several organizational risks associated with the development/testing of end-user applications (including spreadsheets) are discussed. Here we discuss how risk (a) can be reduced when introducing MT in an organizational context, and how MT alleviates risks (b) to (d) in the context of spreadsheets.

First we consider organizational risk (a) (poor coordination among individual end-user programmers). In organizations, a team of end-user programmers may jointly work together for developing and testing a large and complex spreadsheet. If this happens, at least two team members should be *explicitly* appointed with the responsibility of *independently* identifying metamorphic relations for the spreadsheet under test.[9] This suggestion is made in view of our finding reported in Table VII that, considering the five subject spreadsheets together, about 37% ($=\frac{3+2+1+1+9}{9+11+5+5+13}$) of metamorphic relations were independently identified by one participant only. It is obvious that more metamorphic relations will result in a more comprehensive test set, which will in turn increase the chance of detecting spreadsheet faults. Therefore, MT is better to be conducted by a team of testers (preferably leaded by an experienced IT facilitator) instead of a single tester.[10]

Next we turn to organizational risk (b) (insufficient technical support on development/testing). To deal with this risk, Taylor *et al.* (1998) recommend to use a collaborative approach among end-user programmers or an IT help-desk function. Following their recommendation, we have the following suggestions:

- One or more end-user testing groups should be established, through which a mentor-apprenticeship scheme is implemented so that: (i) end-user programmers developing spreadsheets with similar application domains are grouped together in the same team,

and (ii) experienced end-user programmers with extensive domain knowledge are appointed as mentors. Alternatively, an IT help-desk function should be implemented so that the appointed IT facilitators, with experience in MT, could guide the end-user programmers through the individual steps of MT, particularly the identification of metamorphic relations.

- Regular meetings should be organized in which end-user programmers can share and exchange their experiences of using MT for spreadsheet testing.
- To build up a central knowledge base or repository of typical metamorphic relations for each application domain. This arrangement will greatly save the cost and effort in future spreadsheet testing using MT.

Now, we consider organizational risk (c) (little or no documentation). As explained in Section 6, metamorphic relations are the properties of the spreadsheet under test. In other words, these relations correspond to the implicit or explicit functional requirements of the spreadsheet. Thus, a set of well-prepared metamorphic relations can serve as a form of documentation of the spreadsheet.

Finally, as well discussed in Sections 4 to 7, particularly in Table III, MT can alleviate organizational risk (d) (lack of extensive testing) caused by the lack of appropriate testing techniques or guidelines.

9. Related work

In Section 3, we have briefly introduced the two board categories of testing: dynamic and static testing. As we have explained in Section 4 as Challenge 1, there are just a few dynamic testing techniques specifically developed for spreadsheets, such as the constraint-based testing method (Abraham & Erwig, 2006) and the WYSIWYT methodology (Fisher *et al.*, 2006). When compared to MT, the former two techniques are less easy to automate by the end-user programmers (Challenge 2), they do not focus on the intrinsic properties of the spreadsheet application (Challenge 3), and they cannot be applied when the oracle problem occurs (Challenge 4).

Until now, most research work of spreadsheet testing focuses on the static approach, including reviews, inspections, and audits. For example, Panko (1999) investigated the effectiveness of individual and group inspections on detecting spreadsheet faults. He observed that group inspection found 80 percent of all faults, while individual inspection only found 63 percent. Clermont *et al.* (2002) proposed an auditing approach based on three similarity criteria between formula: copy, logical, and structural equivalence. Their study showed that the auditing approach can help testers to find irregularities in the pattern of similar formulae.

By its very nature, MT differs from reviews, inspections, and audits because MT is a dynamic testing technique whereas the others are static techniques.

10. Summary and conclusion

In this paper, we first discussed some organizational risks associated with developing or testing end-user applications. We then reframed one of these risks (lack of extensive testing) in terms of four challenges encountered by end-user programmers when testing a spreadsheet: (i) the small number of appropriate testing techniques, and the required technical knowledge for their use; (ii) the inaccessibility to the associated automated tools; (iii) the lack of specific focus on the intrinsic properties of the application; and (iv) the presence of the oracle problem.

Thereafter, we discussed in detail the concept of MT, and illustrated with examples how MT overcomes the above four challenges. It is worth mentioning that other spreadsheet

testing techniques require end-user programmers to have an in-depth understanding of the principles underlying the testing techniques, and require the support of (hardware/software) platform-specific automated tools — tools which, due to their complexity, are often not possible for end-user programmers to develop. On the other hand, MT does not require end-user programmers to have substantial knowledge about software testing, and the execution of the source and follow-up test cases, and the subsequent comparison of the results, can easily be automated.

We have performed an experiment involving five "real-world" spreadsheets (with real faults introduced, inadvertently, by their developers) to evaluate the effectiveness of MT in detecting various types of spreadsheet faults. The positive results of the experiment have confirmed that MT is highly effective in spreadsheet fault detection, and yet MT is a general technique that can easily be used by end-user programmers to test a large variety of spreadsheet applications. We have also made some suggestions on how to introduce MT in organizations in order to minimize the organizational risk related to poor coordination among end-user programmers, and how MT alleviates other relevant organizational risks.

Notes

1. Because of poor coordination, an end-user programmer might spend days developing an application that another end-user programmer (relatively more experienced) could have developed in a few hours or using more efficient technology (Hill & Barnes, 2011). This results in waste of resources.
2. Senders and Moray (1991) refer a failure, a fault, and a mistake collectively as an *error*.
3. The double minus sign is a standard syntax of the SUMPRODUCT formula. The formula takes one or more arrays of numbers and gets the sum of products of corresponding numbers. In the formula SUMPRODUCT(--(H2:H5001 = "F"), --(E2:E5001 <= 0)), the first array corresponds to the column H2–H5001, and the second array corresponds to the column E2–E5001. The portion "--(H2:H5001 = "F")" looks for the value of "F" across the cells H2–H5001. It returns a bunch of ONEs and ZEROs; one if the value of the cell is "F", zero if the cell contains any other value. Similar explanation applies for the portion "--(E2:E5001 <= 0)".
4. Most modern computer systems include a built-in SIN function, by using sophisticated mathematical techniques (such as the combination of a polynomial or rational approximation with range reduction and a table lookup (Kantabutra, 1996)) to *approximate* the corresponding *sine* value for a given angle; the *exact sine* value is unknown, except for some special angles such as 0° and 90° where *sin*(0°) = 0 and *sin*(90°) = 1.
5. When testing the inventory spreadsheet with MR_1, MR_2, and MR_3, the original SUMPRODUCT formula will be *automatically* updated by the spreadsheet accordingly. Consider, for example, MR_1. Suppose that the original formula is SUMPRODUCT(--(H2:H5001 = "F"), --(E2:E5001 <= 0)), and a row is inserted immediately following the existing row 2 for storing a new slow-moving and non-profitable item. The spreadsheet will automatically change the original formula to SUMPRODUCT(--(H2:H5002 = "F"), --(E2:E5002 <= 0)) after the row insertion. In view of this, we need to introduce a new terminology to facilitate subsequent discussion. Suppose: (i) *F* denote an original formula, (ii) *C* denote the original criterion in *F*, and (iii) *F′* and *C′* denote a "new" formula and criterion, respectively, which are automatically created by the spreadsheet by updating *F* and *C*. If *F′* (or *C′*) has the same "logic" as *F* (or *C*), then the former is called a *variant formula* (or *variant criterion*).
6. This corpus allows researchers to validate their methodologies on a standardized collection of over 4400 "real-world" spreadsheets.

7. The paper by Ronen *et al.* (1989) is commonly regarded as a classical paper on spreadsheet analysis and design, and has been widely cited in the spreadsheet literature (Cragg & King, 1993; Mittermeir & Clermont, 2002; Panko, 1998; Sajaniemi, 2000; Tukiainen, 2000).
8. Ronen *et al.* (1989) have identified a total of eight fault types for spreadsheets. Six of these fault types are listed in Table IV. The other two fault types are "incorrect use of formats and column widths" and "confused range names". Obviously, these two fault types do not affect the *correctness* of the output data. Also, detecting these two fault types normally does not involve the *execution* of a spreadsheet (note that this paper focuses on *dynamic* testing). Thus, their detection is better left to *static* testing techniques such as reviews and inspections. In this regard, our experiment did not involve these two fault types.
9. Acuña and Juristo (2004) report that, in a team-based development approach, a key success factor for quality software process is a clearly defined role for every member of the development team.
10. Previous studies also showed that teamwork is important for improving the effectiveness of MT in fault detection (Liu *et al.*, 2013).

References

Abraham, R. and Erwig, M. (2006), "AutoTest: a tool for automatic test case generation in spreadsheets", in *Proceedings of the IEEE Symposium on Visual Languages & Human-Centric Computing, Brighton, UK*, pp. 43–50.

Acuña, S.T. and Juristo, N. (2004), "Assigning people to roles in software projects", *Software — Practice & Experience*, Vol. 34, No. 7, pp. 675–696.

Alavi, M. and Weiss, I.R. (1989), "Managing the risks associated with end-user computing", in Nelson, R.R. (Ed), *End-User Computing: Concepts, Issues, and Applications*, Wiley, NY, pp. 231–248.

Ayalew, Y. (2007), "A user-centered approach for testing spreadsheets", *International Journal of Computing & ICT Research*, Vol. 1, No. 1, pp. 76–84.

Barr, S., Foley, R. and McMullen, M. (1994), "Towards a quality management system for end-user application development", in *Proceedings of the Software Quality Management Conference, Edinburgh, UK*, pp. 26–28.

Benson, D.H. (1983), "A field study of end-user computing: findings and issues", *MIS Quarterly*, Vol. 7, No. 4, pp. 35–45.

Boehm, B.W. (1981), *Software Engineering Economics*, Prentice Hall, Upper Saddle River, NJ.

Boehm, B.W. and Basili, V.R. (2001), "Software defect reduction top 10 list", *IEEE Computer*, Vol. 34, No. 1, pp. 135–137.

Cale, E.G. (1994), "Quality issues for end-user-developed software", *Journal of Systems Management*, Vol. 45, No. 1, pp. 36–39.

Caulkins, J.P., Morrison, E.L. and Weidemann, T. (2007), "Spreadsheet errors & decision making: evidence from field interviews", *Journal of Organizational & End User Computing*, Vol. 19, No. 3, pp. 1–23.

Chan, W.K., Chen, T.Y., Lu, H., Tse, T.H. and Yau, S.S. (2005), "A metamorphic approach to integration testing of context-sensitive middleware-based applications", in *Proceedings of the 5th International Conference on Quality Software, Melbourne, Australia*, pp. 241–249.

Chen, T.Y., Cheung, S.C. and Yiu, S. (1998), *Metamorphic Testing: A New Approach for Generating Next Test Case*s, Technical Report (HKUST-CS98-01), Department of Computer Science, Hong Kong University of Science & Technology, Hong Kong.

Chen, T.Y., Ho, J.W.K., Liu, H. and Xie, X. (2009), "An innovative approach for testing bioinformatics programs using metamorphic testing", *BMC Bioinformatics*, Vol. 10, No. 24, doi:10.1186/1471-2105-10-24.

Chen, T.Y., Tse, T.H. and Zhou, Z.Q. (2003), "Fault-based testing without the need of oracles", *Information & Software Technology*, Vol. 45, No. 1, pp. 1–9.

Christy, C.T. (2006), *Engineering with the Spreadsheet: Structural Engineering Templates Using Excel*, American Society of Civil Engineers, Reston, VA.

Clermont, M., Hanin, C. and Mittermeir, R.T. (2002), "A spreadsheet auditing tool evaluated in an industrial context", in *Proceedings of the Annual Conference of the European Spreadsheet Risks Interest Group, Cardiff, UK*, pp. 35–47.

Cragg, P. and King, M. (1993), "Spreadsheet modelling abuse: an opportunity for OR?", *Journal of Operational Research Society*, Vol. 44, No. 8, pp. 743–752.

Dewey, B.R. (1998), "Problem-solving tools for engineering students", in *Proceedings of the 28th Annual Frontiers in Education Conference* (*FIE '98*), *Tempe, AZ*, pp. 1050–1055.

Ding, J., Wu, T., Lu, J.Q. and Hu, X.-H. (2010), "Self-checked metamorphic testing of an image processing program", in *Proceedings of the 4th IEEE International Conference on Secure Software Integration & Reliability Improvement, Singapore*, pp. 190–197.

Doll, A. (2000), "Performing metallurgical calculations in a spreadsheet", http://www.agd consulting.ca/CalcPaper.

Fisher II, M. and Rothermel, G. (2005), "The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms", in *Proceedings of the 1st Workshop on End-User Software Engineering, St. Louis, MO*, pp. 47–51.

Fisher II, M., Rothermel, G., Brown, D., Cao, M., Cook, C. and Burnett, M. (2006), "Integrating automated test generation into the WYSIWYT spreadsheet testing methodology", *ACM Transactions on Software Engineering & Methodology*, Vol. 15, No. 2, pp. 150–194.

Franz, L.A. and Shih, J.C. (1994), "Estimating the value of inspections & early testing for software projects", *Hewlett-Packard Journal*, Vol. 45, No. 6, pp. 60–67.

Gripenberg, P. (2011), "Computer self-efficacy in the information society", *Information Technology & People*, Vol. 24, No. 3, pp. 303–331.

Grochtmann, M. and Grimm, K. (1993), "Classification-trees for partition testing", *Software Testing, Verification & Reliability*, Vol. 3, No. 2, pp. 63–82.

Grossman, T.A. (2003), "Accuracy in spreadsheet modelling systems", in *Proceedings of the Annual Conference of the European Spreadsheet Risks Interest Group, Dublin, Ireland*, pp. 91–97.

Grossman, T.A. and Özlük, Ö. (2010), "Spreadsheets grow up: three spreadsheet engineering methodologies for large financial planning models", in *Proceedings of the Annual Conference of the European Spreadsheet Risks Interest Group, London, UK*.

Harrison, W. (2004), "The dangers of end-user programming", *IEEE Software*, Vol. 21, No. 4, pp. 5–7.

Hill, M.C. and Barnes, W.A. (2011), "End-user computing applications: implications for internal auditors and managers", *The CPA Journal*, Vol. 81, No. 7, pp. 67–71.

Institute of Electrical & Electronics Engineers (IEEE) (1990), *IEEE Standard 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*, IEEE, New York, NY.

Institute of Electrical & Electronics Engineers (IEEE) (2008), *IEEE Standard 1028™-2008: IEEE Standard for Software Reviews & Audits*, IEEE, New York, NY.

Jee, E., Yoo, J., Cha, S. and Bae, D. (2009), "A data flow-based structural testing technique for FBD programs", *Information & Software Technology*, Vol. 51, No. 7, pp. 1131–1139.

Jorgenson, P.C. (2008), *Software Testing: A Craftsman's Approach*, 3rd ed., Auerbach Publications, Boca Raton, FL.

Kandt, R.K. (2009), "Experiences in improving flight software development processes", *IEEE Software*, Vol. 26, No. 3, pp. 58–64.

Kaner, C., Falk, J. and Nguyen, H.Q. (1999), *Testing Computer Software*, 2nd ed., Wiley, New York, NY.

Kantabutra, V. (1996), "On hardware for computing exponential & trigonometric functions", *IEEE Transactions on Computers*, Vol. 45, No. 3, pp. 328–339.

Laudon, K.C. and Laudon, J.P. (2010), *Management Information Systems: Managing the Digital Firm*, 11th ed., Pearson, Upper Saddle River, NJ.

Lee, S.M., Kim, Y.R. and Lee, J. (1995), "An empirical study of the relationships among end user information systems acceptance, training and effectiveness", *Journal of Management Information Systems*, Vol. 12, No. 2, pp. 189–202.

Lei, Y., Carver, R.H., Kacker, R. and Kung, D. (2007), "A combinatorial testing strategy for concurrent programs", *Software Testing, Verification & Reliability*, Vol. 17, No. 2, pp. 207–225.

Liu, H., Kuo, F.-C., Towey, D. and Chen, T.Y. (2013), "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Transactions on Software Engineering* (in press).

Mason, D. and Willcocks, L. (1991), "Managers, spreadsheets & computing growth: contagion or control?", *Information Systems Journal*, Vol. 1, No. 2, pp. 115–128.

McDaid, K. and Rust, A. (2009), "Test-driven development for spreadsheet risk management", *IEEE Software*, Vol. 26, No. 5, pp. 31–36.

McDonough, D. (2004), "Spreadsheets and nuclear reactors must never mix!", http://it.toolbox.com/blogs/spreadsheets/spreadsheets-nuclear-reactors-must-never-mix-771.

Meyer, B. (2008), "Seven principles of software testing", *IEEE Computer*, Vol. 41, No. 8, pp. 99–101.

Mittermeir, R. and Clermont, M. (2002), "Finding high-level structures in spreadsheet programs", in *Proceedings of the 9th Working Conference on Reverse Engineering, Richmond, VA*, pp. 221–232.

Myers, G.J. (2004), *The Art of Software Testing*, 2nd ed., Wiley, Hoboken, NJ.

Panko, R.R. (1998), "What we know about spreadsheet errors", *Journal of End User Computing*, Vol. 10, No. 2, pp. 15–21.

Panko, R.R. (1999), "Applying code inspection to spreadsheet testing", *Journal of Management Information Systems*, Vol. 16, No. 2, pp. 159–176.

Panko, R.R. (2006), "Recommended practices for spreadsheet testing", in *Proceedings of the Annual Conference of the European Spreadsheet Risks Interest Group, Cambridge, UK*, pp. 73–84.

Panko, R.R. (2009), "Two experiments in reducing overconfidence in spreadsheet development", in Clarke, S. (Ed), *Evolutionary Concepts in End User Productivity & Performance: Applications for Organizational Progress*, Information Science Reference, Hershey, PA, pp. 131–149.

Panko, R.R. and Aurigemma, S. (2010), "Revising the Panko-Halverson taxonomy of spreadsheet errors", *Decision Support Systems*, Vol. 49, No. 2, pp. 235–244.

Peng, G., Wang, Y. and Kasuganti, R. (2011), "Technological embeddedness & household computer adoption", *Information Technology & People*, Vol. 24, No. 4, pp. 414–436.

Pierson, J.K., Forcht, K.A. and Teer, F.P. (1990), "Determining documentation requirements for user developed applications," *Information & Management*, Vol. 19, No. 1, pp. 21–31.

Pryor, L. (2004), "When, why and how to test spreadsheets", in *Proceedings of the Annual Conference of the European Spreadsheet Risks Interest Group, Klagenfurt, Austria*, pp. 145–151.

Rahuma, K.M., Sondi, F., Milad, M. and AbuGheit, M. (2013), "Comparison between spreadsheet and specialized programs in calculating the effect of scale deposition on the well flow performance", *Journal of Petroleum & Gas Engineering*, Vol. 4, No. 4, pp. 69–80.

Rajalingham, K., Chadwick, D., Knight, B. and Edwards, D. (2000), "Quality control in spreadsheets: a software engineering-based approach to spreadsheet development", in *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, Maui, Hawaii*.

Ronen, B., Palley, M.A. and Lucas Jr., H.C. (1989), "Spreadsheet analysis & design", *Communications of the ACM*, Vol. 32, No. 1, pp. 84–93.

Sajaniemi, J. (2000), "Modeling spreadsheet audit: a rigorous approach to automatic visualization", *Journal of Visual Languages & Computing*, Vol. 11, No. 1, pp. 49–82.

Scaffidi, C., Shaw, M. and Myers, B. (2005), "Estimating the numbers of end users & end user programmers", in *Proceedings of the IEEE Symposium on Visual Languages & Human-Centric Computing, Dallas, TX*, pp. 207–214.

Schultheis, R. and Sumner M. (1994), "The relationship of application risks to application controls: a study of microcomputer based spreadsheet application", Journal of End User Computing, Vol. 6, No. 2, pp. 11–18.

Segura, S., Hierons, R.M., Benavides, D. and Ruiz-Cortés, A. (2011), "Automated metamorphic testing on the analyses of feature models", *Information & Software Technology*, Vol. 53, No. 3, pp. 245–258.

Senders, J.W. and Moray, N.P. (1991), *Human Error: Cause, Prediction, & Reduction*, Lawrence Erlbaum, Hillsdale, NH.

Sommerville, I. (2011), *Software Engineering*, 9th ed., Pearson, Boston, MA.

Taylor, M.J., Moynihan, E.P. and Wood-Harper, A.T. (1998), "End-user computing & information systems methodologies", *Information Systems Journal*, Vol. 8, No. 1, pp. 85–96.

Tukiainen, M. (2000), "Uncovering effects of programming paradigms: errors in two spreadsheet systems", in *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group*, *Cosenza, Italy*, pp. 247–266.

Vilkomir, S.A., Kapoor, K. and Bowen, J.P. (2003), "Tolerance of control-flow testing criteria", in *Proceedings of the 27th Annual International Computer Software & Applications Conference*, *Dallas, TX*, pp. 182–187.

Xie, X., Ho, J.W.K., Murphy, C., Kaiser, G., Xu, B. and Chen, T.Y. (2011), "Testing & validating machine learning classifiers by metamorphic testing", *Journal of Systems & Software*, Vol. 84, No. 4, pp. 544–558.

Zhou, Z.Q., Zhang, S.J., Hagenbuchner, M., Tse, T.H., Kuo, F.-C. and Chen, T.Y. (2012), "Automated functional testing of online search services", *Software Testing, Verification & Reliability*, Vol. 22, No. 4, pp. 221–243.