CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Binus University Repository

Simple Spreadsheet Test Case Application to Test Spreadsheet Formula in End-User Software Engineering

Noprianto
Doctor of Computer Science
Bina Nusantara University
Jakarta, Indonesia
nop@noprianto.com

Benfano Soewito
Master in Computer Science
Bina Nusantara University
Jakarta, Indonesia
bsoewito@binus.edu

Ford Lumban Gaol
Doctor of Computer Science
Bina Nusantara University
Jakarta, Indonesia
fgaol@binus.edu

Harco Leslie Hendric
Spits Warnars
Doctor of Computer Science
Bina Nusantara University
Jakarta, Indonesia
shendric@binus.edu

***Abstract*—Using spreadsheet software, users may do some forms of programming using formula and function. However, formula definition is often considered correct after tested with only one or few inputs. This may be fine with simple formula, but for more complex ones, more testings should be performed, in order to prevent solution that riddled with errors. However, writing a formula test case in spreadsheet software is not a simple task, mainly because there is no standard way to do that. In this paper, a simple spreadsheet test case application is proposed. User can define a formula, with many input variants, along with expected results. After that, application will generate a spreadsheet document, with all the needed contents for testing, along with test result. That way, user may re-check the formula and make necessary modifications (then run the test, again). Using this method, a formula can be tested first, with many input as needed, before it put in real document. All of these will impact in more productive programmers, with less time spent for debugging.**

Keywords—spreadsheet, formula, spreadsheet test case, testing, end-user software engineering, end-user programming

I. INTRODUCTION

In spreadsheet application, users may work with raw tabular data, which can be processed with formula and function. One formula can be applied to one or more input, but testing is often performed only for one input variant. If there is no error, and the result is considered correct, then the formula is applied to more data, optionally by dragging the mouse. Furthermore, if there is no errors shown, then the formula is considered valid. This is applied to real data.

However, such real data may not provide all the possibilities to prove that one formula is correct. Only when users can write test cases for many input variants, they can be sure that one formula is correct. This is particularly true for complex formula definition.

But, writing test cases is not a simple task. Users may populate the data with some dummy or test data, but it will only test if there is any error when a formula is applied to all data. There is no verification, compared with expected result. Meanwhile, writing an expected result along with all the data and formula will populate the worksheet with unneeded contents. Putting the test case in another worksheet may work, but modifying a formula also means modifying all the test cases, manually.

All of these could be complex in end-user software engineering, where requirements and specifications are implicit, and testing or verification is overconfident [9]. Therefore, an easy to use test case tool could be useful to test a formula with several input data before it put in real document. With more testing, programmers could be more productive [6].

II. LITERATURE STUDY

Programming is defined as the process of planning or writing a program. A program itself is defined as a collection of specifications that may take variable inputs, and that can be executed or interpreted by a device with computational capabilities [9]. Based on this definition, tools being used in programming are not limited to programming languages, but also covers spreadsheet software with formula or function development, database tools with programmable features, report generators, web authoring tools, and many others.

Based on the goal, there are professional programmers and end-user programmers. The former is being paid to ship and maintain software overtime, while the latter write program to support some goal in their own domain of expertise. Thus, end-user programming is a programming to achieve the result of a program primarily for personal use. This is contrast with professional programming whose goal of producing code for others to use [9].

In end-user programming, experience is an independent concern. It is important not to conflate end-user programming with inexperience [9]. A professional programmer with years of experience can also become an end-user programmer, for example when writing program for personal use, using new technology or tool.

In similar context, it is also important not to relate end-user programming with the usage of simple or not serious programming languages. Scientists may use C programming language in some personal research projects (end-user programming) and professional programmers may use plain HTML to create commercial website [9].

End-user programmers is also important based on data pictured in figure 1. In year 2006, in United States, there were only 3 million professional programmers, compared with 12 million people who said they do programming at work, and over 50 million spreadsheet and database users [4].

978-1-5090-5130-4/16/$31.00 ©2016 IEEE

End-user programming has become a widespread phenomenon [5].

Spreadsheets are particularly interesting because they provide computational techniques that match users' tasks (that shield users from low-level details of programming) and their table-oriented interface (that serves as a model for application) [11]. Although spreadsheets are considered for small scratch pad applications, they are also used to develop many serious applications. A survey conducted in 2003 found that 47% mid-size companies use stand-alone spreadsheet for planning and budgeting [12]. Spreadsheets are used in almost all businesses [13].

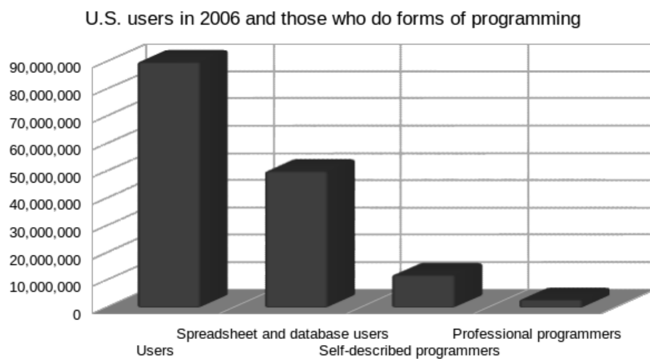

Fig. 1. U.S. users in 2006 and those who do forms of programming (adopted from [4]).

Related with end-user programming, there is end-user development. Lieberman et al. [10] defined end-user development as a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify, or extend a software artifact. According to that definition, focus is not put only on the creating a new program.

This is also in line with software engineering process, where programming is just a part of the whole activities. As defined by IEEE Standard 610.12, software engineering is the application of systematic, disciplined, quantifiable approaches to the development, operation, and maintenance of software.. Therefore, if only the programming activity is concerned, then quality of software developed by end-user programmers may not be considered, resulted in new software (which empowers end-users) that may riddled with errors. These errors may not catastrophic, but their effects can matter [4].

TABLE I. QUALITATIVE DIFFERENCES BETWEEN END-USER AND PROFESSIONAL SOFTWARE ENGINEERING

Software Engineering Activity	*End-user*	*Professional*
Requirements	Implicit	Explicit
Specifications	Implicit	Explicit
Reuse	Unplanned	Planned
Testing / Verification	Overconfident	Cautious
Debugging	Opportunistic	Systematic

Adapted from: [9]

Considering the quality issue, another research area, end-user software engineering, has emerged. As professional software engineering, end-user software engineering still involves systematic and disciplined activities that address software quality issues, but all of these activities are secondary to the goal of the program. Because of this goal, amount of attention given to software quality concerns is different with professional software engineering [9].

Furthermore, qualitative differences between end-user software engineering and professional software engineering are listed in Table 1.

III. RELATED WORKS

One of popular methods for testing spreadsheet is the What You See Is What You Test [15], implemented in Forms/3. The WYSIWYT method can provide visual feedback about the “testedness” of a spreadsheet [16], and can be applied to individual cells. Automated test generation is also possible [8]. This method can also be extended with several tools such as GoalDebug and AutoTest for debugging, and automated test generation, respectively [1]. WYSIWYT method is really useful for very serious spreadsheet testing, particularly for individual cell test.

Another utility that can help end users is UCheck, a unit reasoning system that allows end users to identify and correct errors in their spreadsheets [2]. In other way, to allow users to work safely with tables based on templates, Gencel, an extension to Excel, can be used [7]. Both can be used to work with 'correct' spreadsheet.

Users need validating the unit correctness of Excel spreadsheet can also use XeLda. This tool highlights cells if their formulas process values with incorrect units [3].

IV. PROPOSED METHOD

We want to make testing a formula in spreadsheet easier, by using an application that can be used to write test cases for spreadsheet formula. This application does not require any spreadsheet software to be installed in the system, in order to generate test cases. It will run as standalone graphical user interface application. To make it easier for users, it comes with simple user interface: the only needed inputs are the formula definition, where to put formula result, where to put comparison between formula and expected result, and one or more test cases along with expected result.

Output of this application is in Office Open XML (ISO/IEC 29500) spreadsheet, with .xlsx file name extension. Number of sheets in this document will depend on number of test cases entered by user, plus one summary worksheet. In first sheet, there will be summary which shows failed or passed test cases. If there is any mismatch between formula calculation result and expected result, it will be reported. That way, users may recheck the formula and make necessary modifications, then run the test, again.

This also promotes test-driven development style (where test cases are written before the actual codes), but applied to spreadsheet. Rust, Bishop, and McDaid [17] have concluded that this methodology has the potential to improve the development of spreadsheet.

This application is implemented using Python Programming Language, using Openpyxl library (for

working with Office Open XML spreadsheet) and PySide (for user interface). To make it accessible for wider audience, it will be released as free/open source software, and may be downloaded from https://github.com/nopri/code.

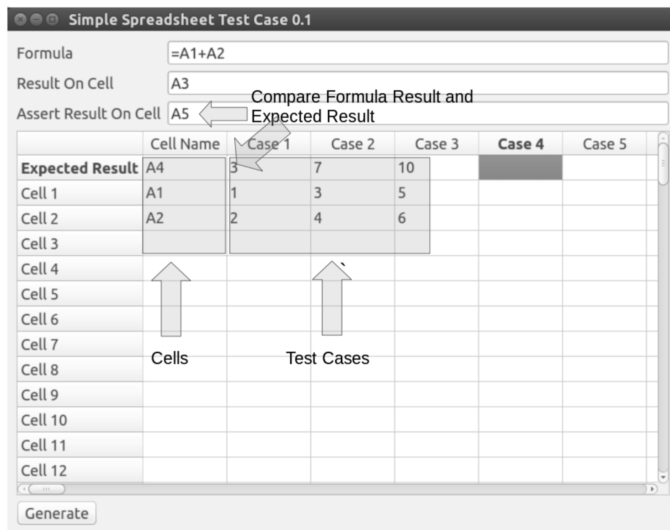

Fig. 2. Simple Spreadsheet Test Case Application.

In Figure 2, we illustrate that we want to test one trivial formula (=A1+A2), whose result is put on cell A3. These are two first inputs in this application:

- Formula: this is where we can input the formula to be tested. It must be a valid spreadsheet formula, as understood by compatible spreadsheet software. We use =A1+A2 formula, but any valid formula will do.
- Result On Cell: this is the cell, where we actually want to put the formula. If we enter A3 in this field, then, in generated spreadsheet document, value of A3 will be =A1+A2.

Let's take a look at Cell Name column in Figure 2 (we will get back to Assert Result On Cell field later). This is where we can input all the cells involved in defined formula. In =A1+A2 formula, at least we have to specify A1 and A2 cells. And, don't forget to specify where we want to put the value, that is expected as correct result, which is A4 in this illustration.

Using values from Case 1 column, value of A1 cell will be 1, an value of A2 cell will be 2. Based on what defined as “Formula” and “Result On Cell”, value of A3 cell will be =A1+A2. And, value of cell A4 will be 3 (because we enter and expect this value). In this test case, value of both A3 and A4 cell will be 3. In other words, what we expect (3, in A4 cell) and what the result (3, in A3 cell, because 1+2=3) are the same. So, this test case will pass.

Then we get back to “Assert Result On Cell” field. In Figure 2, we enter A5. Basically, it means that, we want to test whether value of A3 cell (the formula) and A4 cell (the expected result) are the same, then put the comparison in A5 cell. That way, value of A5 will be =A3=A4 formula. Based on values from Case 1 column, since A3=A4, then value of A5 will be 1. It means that, what we expect and result of formula are identical.

As another example, test case 3 illustrates wrong expected result (because A4 cell in Case 3 should expect 11, not 10). Normally, the subject of testing is the formula (that is the reason why test cases are written). However, on the other side, we should also provide correct test cases. This erroneous test case is shown, only to illustrate the test summary, not how spreadsheet testing should be done.

In the spreadsheet-like table interface, user can input cell names (which should be related with formula definition, as mentioned). Test cases, along with their expected results, can be specified, and will be read and processed, column by column, until empty cell is found. Each test case will be put in its own worksheet.

When user clicks the “Generate” button, a spreadsheet document will be generated. When this file is opened in compatible spreadsheet software, user can view the test result in first worksheet (Result). Number of failed and passed tests will be shown, along with individual test case result, as shown in Figure 3. Each test case will be automatically named as [<worksheet name>] <cell name>, and will be put in column A, starting from second row. Actual comparison result will be put in column B. If everything is working as expected, then column B, for every test case line in corresponding column A, will show value of 1, and number of failed test cases should be zero. In Figure 3, we can see that there is one failed test and two passed tests.

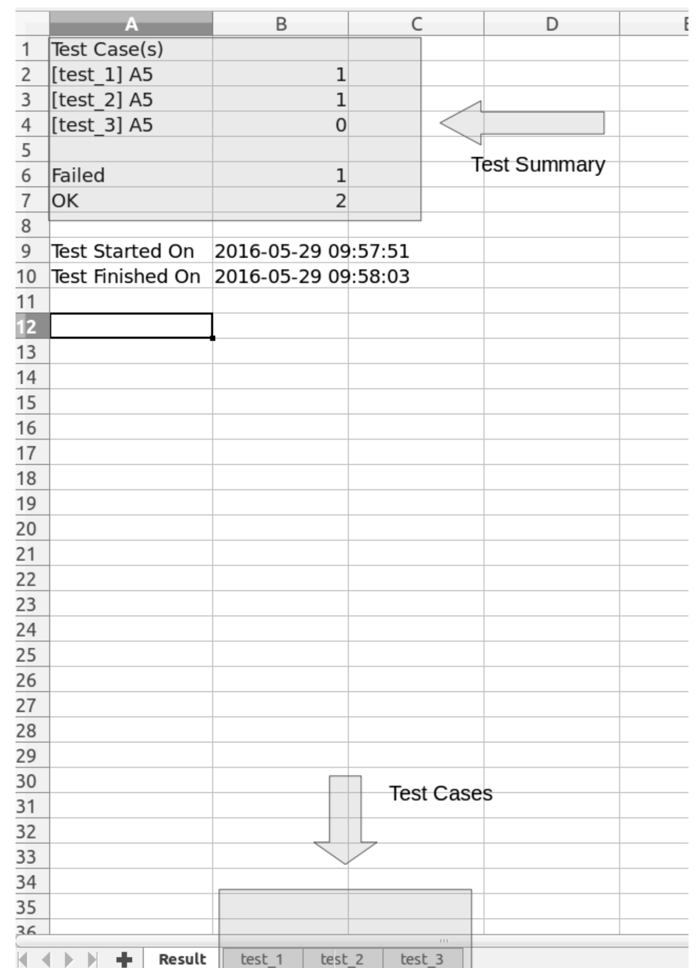

Fig. 3. Test result (Office Open XML spreadsheet), with one failed test and two passed tests.

If detail of a test case is needed, user can go to its corresponding worksheet, as shown in Figure 4 and Figure 5. In Figure 4, everything is working as expected. Formula

result (A3 cell) and expected result (A4 cell) are identical. We can see that formula =A1+A2 returns the correct result (A3=3), because value of A1 cell and A2 cell is 1 and 2, respectively. And, the most important one, comparison formula in A5 cell returns 1. This value is referenced in Result worksheet, in cell B2, as shown in Figure 3.

And, since we purposely put an erroneous expected result in Case 3, we can see that the comparison formula returns 0, in Figure 5. This also referenced in Result worksheet, in cell B4, as shown in Figure 3.

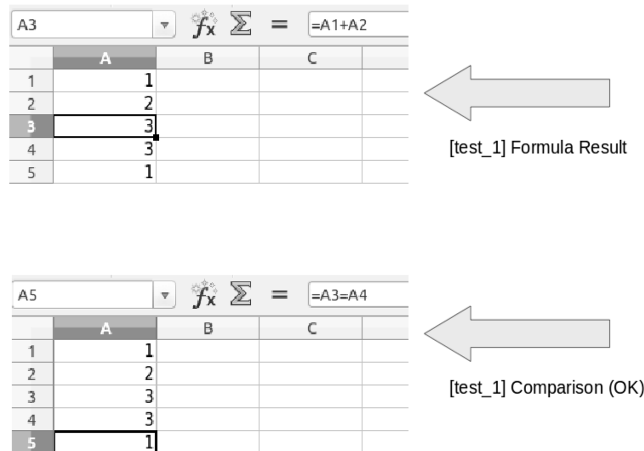

Fig. 4. Worksheet test_1 illustrating formula result and comparison.

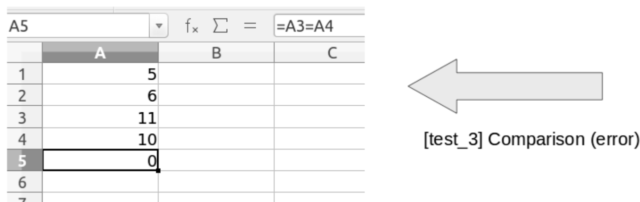

Fig. 5. Worksheet test_3 illustrating comparison.

In Figure 2, we illustrate a trivial formula, =A1+A2, that should always "correct". But, as a formula might get more and more complex, involving deep nested function calls, and so on, at some point, we might unsure whether a formula is still "correct". This is where this application might be useful, since we can test a formula using several test cases. If we maintain a number of test cases with their own expected result, then any changes to the formula should still return values as expected.

Below is the algorithm (for main functionality only) when "Generate" button is clicked:

- Value of each required inputs is checked for valid format. This, however, only applied to cell name. Value of formula definition is only checked for empty value, and the actual interpretation is left for spreadsheet engine.
- File dialog for output file name, will be shown. It is possible to save test cases result to as many files as needed.
- For each column (which represents individual test case):
    - Get the expected result cell. If this cell is empty, then the iteration is stopped, assuming that there is no more test case to generate. If no expected result is specified, then it will be pointless because no comparison could be done, hence, no more test case must be assumed.
    - Create a new worksheet.
    - For each row in current column:
        - Get cell name (first column). If this value is empty or invalid (according to cell name format), then iteration is stopped, assuming invalid input.
        - If row number is equal to zero, then comparison formula (=<cell_1>=<cell_2>) will be put in target worksheet, according to 'Assert Result On Cell' value.
        - For another cells, put the corresponding value in specified cell. Value checking is done mainly to detect its type. If it looks like number, then it will be put as number (in spreadsheet document). Otherwise, it will be put as text. This is needed because, currently, for simplicity, each input in this program is interpreted as text. Type detection is done by trying to convert a value to a floating point number (if the conversion is successful, then it is a number).
    - Put a summary line to "Result" worksheet. To refer a cell in another worksheet, we will use =<worksheet_name>!<cell> formula.
- Put summary on failed and passed tests using CountIf function. A test will be passed if cell value comparison between expected result and formula result is 1.
- Save the workbook into specified output file name, and show the information dialog. Program will not quit at this phase, so different test cases (based on current values), could be specified, and saved to another output file.

V. DISCUSSION

In this paper, to ease the testing of a spreadsheet formula, we propose a test case application. While we do not relate end-user programmer as inexperience, we argue that by using a test case application, testing a spreadsheet formula would be easier.

This application is still in early development phase. Many useful features are still lacking. For example, user cannot save the test cases into a file, and load them later. This is a useful feature because test cases could be very complex, and typing them again every time the formula is changed (or to be tested) is time consuming and error prone. This feature should be implemented in the next version.

Another feature that might be useful is allowing user to specify the type of value entered for each test case. As mentioned in proposed method, type detection will be

performed for each value. If a value looks like a number, then it will be put as a number in spreadsheet document. In certain condition, this might be not the expected behavior.

And, we can also add the formula parser, mainly to evaluate the syntax of defined formula. That way, formula whose errors in syntax will not be processed. Certainly, since this is not a spreadsheet application, after the syntax is already correct, we will not interpret the formula. This should be left to spreadsheet engine. After all, this is only a test case application.

By releasing this application as free/open source software, we hope that anyone can improve this application, to make it more useful.

However, we also realize that test case application is really just a tool. If testing is performed with few or not representative input data, then there is no more additional benefit, compared with just populate a worksheet with dummy data. Verification is the key, just like any unit testing in any programming language.

VI. References

[1] R. Abraham and M. Erwig. GoalDebug: A spreadsheet debugger for end users. In Proceedings of the 29th international conference on Software Engineering (pp. 251-260), IEEE Computer Society, May 2007.

[2] R. Abraham and M. Erwig. UCheck: A spreadsheet type checker for end users. Journal of Visual Languages & Computing, 18(1), 71-95, 2007.

[3] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the unit correctness of spreadsheet programs. In Software Engineering, 2004, ICSE 2004. Proceedings. 26th International Conference on (pp. 439-448), IEEE, May 2004.

[4] M. Burnett. What is end-user software engineering and why does it matter?. In End-User Development (pp. 15-28), Springer Berlin Heidelberg, 2009.

[5] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In Proceedings of the 25th international conference on Software engineering (pp. 93-103), IEEE Computer Society, May 2003.

[6] H. Erdogmus. On the effectiveness of test-first approach to programming, 2005.

[7] M. Erwig, R. Abraham, S. Kollmansberger, and I. Cooperstein. Gencel: a program generator for correct spreadsheets. Journal of Functional Programming, 16(03), 293-325, 2006.

[8] M. Fisher II, G. Rothermel, D. Brown, M. Cao, C. Cook, and M. Burnett. Integrating automated test generation into the WYSIWYT spreadsheet testing methodology. ACM Transactions on Software Engineering and Methodology (TOSEM), 15(2), 150-194, 2006.

[9] A.J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, and B. Myers. The state of the art in end-user software engineering. ACM Computing Surveys (CSUR), 43(3), 21, 2011.

[10] H. Lieberman, F. Paternò, M. Klann, and V. Wulf. End-user development: An emerging paradigm. (pp. 1-8), Springer Netherlands, 2006.

[11] B.A. Nardi and J.R. Miller. The spreadsheet interface: A basis for end user programming. Hewlett-Packard Laboratories, 1990.

[12] R.R. Panko. What we know about spreadsheet errors. Journal of Organizational and End User Computing (JOEUC), 10(2), 15-21, 2005.

[13] S.G. Powell, K.R. Baker, and B. Lawson. Errors in operational spreadsheets. Journal of Organizational and End User Computing (JOEUC), 21(3), 24-36, 2009.

[14] S.G. Powell, K.R. Baker, and B. Lawson. A critical review of the literature on spreadsheet errors. Decision Support Systems, 46(1), 128-138, 2008.

[15] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. What you see is what you test: A methodology for testing form-based visual programs. In Proceedings of the 20th international conference on Software engineering (pp. 198-207). IEEE Computer Society, April 1998.

[16] K.J. Rothermel, C.R. Cook, M. Burnett, J. Schonfeld, T.R. Green, and G. Rothermel. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In Software Engineering, 2000. Proceedings of the 2000 International Conference on (pp. 230-239). IEEE, 2000.

[17] A. Rust, B. Bishop, and K. McDaid. Investigating the potential of Test-Driven Development for spreadsheet engineering. arXiv preprint arXiv:0801.4802, 2008.