ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/2532743

A Generalised Spreadsheet Verification Methodology

Article · February 2002

DOI: 10.1145/563857.563826 · Source: CiteSeer

CITATIONS
5

READS
14

3 authors, including:

John Morris
University of Auckland

430 PUBLICATIONS **13,140** CITATIONS

SEE PROFILE

All in-text references underlined in blue are linked to publications on ResearchGate, letting you access and read them immediately.

Available from: John Morris
Retrieved on: 19 September 2016

A Generalised Spreadsheet Verification Methodology

Nick Randolph
Software Engineering Australia (WA)
Enterprise Unit 5 De Laeter Way
Bentley 6102
Western Australia

randolph@seawa.org.au

John Morris and Gareth Lee
Department of Electrical & Electronic Engineering
University of Western Australia
Nedlands 6907
Western Australia

[morris, gareth]@ee.uwa.edu.au

Abstract

Although spreadsheets have been around for over thirty years, we are only just realising their importance. Most companies use spreadsheets in their decision-making processes, but rarely employ any form of testing. This paper shows how an "all-uses" test adequacy technique can be integrated into Microsoft's Excel. The modular technique adopted makes the implementation spreadsheet package independent. It also includes a user interface, to assist developers specify test cases and a technique for recording test cases and session information. In particular it presents a systematic technique for constructing test cases. As a key problem with spreadsheet development is the inexperience of developers, this paper describes an easy to use tool that will improve the standard of spreadsheets developed.

Keywords: spreadsheets, software testing, errors, verification.

1 Introduction

The evolution of spreadsheet languages through commercial applications such as Excel has meant that they appeal to a variety of users. Spreadsheets are being used in a range of applications, from home accounts, to business management, to data analysis. In each application the user has a different background from which to draw knowledge. With a focus on business outcomes and speed, it is likely that user spreadsheets will display problems such as being unreadable, inaccurate and inflexible. Accuracy is particularly important as it determines the usefulness of the spreadsheet.

Spreadsheets are inherently easy to navigate and manipulate, and there is a perceived simplicity with which both small and large tasks can be achieved. Conversely, spreadsheets that are poorly designed can be difficult to understand, hard to follow, often have errors and provide very little flexibility. A research survey by Panko and Halverson (1996) found evidence that up to 77% of spreadsheets that are considered 'finished' contain faults. The same research also included 'production' level spreadsheets and identified faults in up to 90% of those tested.

Copyright ©2002, Australian Computer Society, Inc. This paper appeared at the Twenty-Fifth Australasian Computer Science Conference (ACSC2002), Melbourne, Australia. Conferences in Research and Practice in Information Technology, Vol. 4. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

The development process for spreadsheets has remarkable parallels with traditional information systems. There are three key phases, designing, building and testing the spreadsheet. The development of a spreadsheet can be treated as a software engineering task where the sources of faults need to be identified and eliminated. Because of the diversity of spreadsheet users, the software engineering process must be presented in a form that is easy to apply in a wide range of situations.

There have been a number of methodologies that have been put forward to assist developers both develop and test spreadsheets. At the Integrity Research Centre, Rajalingham et al (1999) showed that spreadsheet errors could be classified and how a five stage methodology could be used to reduce the number of errors made when developing a spreadsheet. This research was followed up with four other papers, which look further into error classification (Rajalingham et al, 2000), the application of Jackson charts and modularisation in spreadsheet development (Knight et al, 2000), how visual presentation of cell formula affects accuracy (Chadwick et al, 2000) and another five step methodology that incorporates hierarchical decomposition (Edwards et al, 2000). Each of these papers targets a particular area of spreadsheet development, trying to address some of the issues that developers face.

Rothermel et al (1997) at the Oregon State University have examined a number of coverage techniques for testing spreadsheets. This has been extended into a practical testing methodology (Rothermel et al, 2001) that provides real-time feedback to the developer about the level of testing that has been carried out on the spreadsheet. The core methodology has also been scaled up to test large homogeneous grids (Burnett et al, 1999) and to provide debugging information to the developer (Reichwein et al, 1999). This methodology has been shown to improve the level of testing of spreadsheets (Rothermel et al, 2000).

There is marked difference between the focus of the two research groups. At the Integrity Research Centre, the focus is on improving the design process by providing visual guides and stepped methodologies. At the Oregon State University the approach was to concentrate on how people test spreadsheets. Their methodology concentrates on building test sets and promoting test coverage through visual guides. The common feature across both research groups is the use of visual aids to help improve spreadsheet accuracy.

This paper introduces a generalised implementation of the methodology developed at the Oregon State University. The original methodology was developed using a research spreadsheet package, Forms/3. Rothermel et al (2000) provides evidence that supports the use of this methodology, but its success is limited as few people use Forms/3. Our implementation extends the methodology to work with a popular commercially available spreadsheet, Microsoft's Excel.

Rapps and Weyuker (1985) introduce and compare different coverage criteria for generating test cases. The methodology put forward by Rothermel et al (2000) use the definition-use paths criteria to measures how well a cell has been tested. Our implementation goes to the next level by providing an easy to use graphical interface that assists the developer build test cases. We also extend the definition-use criteria by testing all combinations of cell definitions with their associated use.

2 Background

2.1 Spreadsheet Languages

Spreadsheets are a collection of cells that contain formulae or input data, which have been entered either by the developer or the end user. The value of each cell is defined by the evaluation of the formula it contains.

Although most commercial spreadsheets come with a variety of macro languages that can extend the capability of the spreadsheet, for the purposes of this paper only simple spreadsheets will be considered. The most significant restriction is that there is no form of recursion.

2.2 Cell Relation Graph

Rothermel et al define a cell relation graph that illustrates not only the execution path in individual cells, but also the relationships between cells.

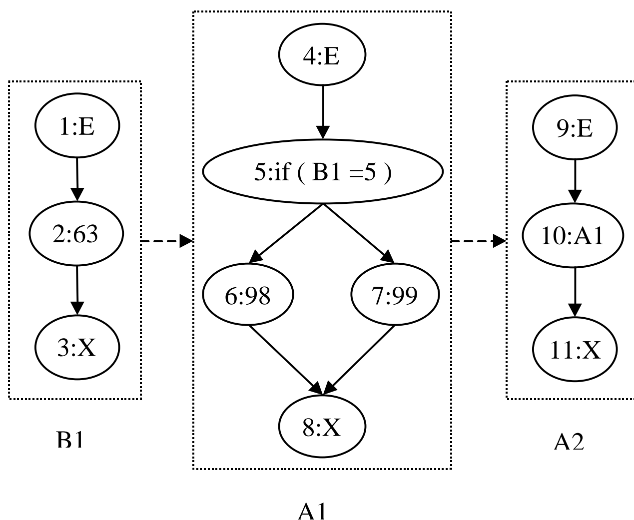

Figure 1: Example Cell Relation Graph

Figure 1 illustrates a Cell Relation Graph between the cells Al and A2 (in Excel these formulae are written as "`=if(B1=5,99,98)`" and "`=A1`" respectively). Execution of a cell starts with the Entry (E) node, follows a single path through the cell and ends with the Exit (X) node. Each dashed box holds the formula graph for a particular cell. The solid arrows illustrate the flow of control, while the dashed lines represent the flow of data between cells.

In cell A1 there are two paths through the graph, whereas B1 and A2 have only one. These are referred to as the possible *definitions* of cells A1, B1 and A2.

The dashed arrow linking the cell indicates that the cell B1 is used by cell A1 and that cell A1 is used by cell A2. A *Use* is defined as the node where the reference is made. For example, cell B1 is used in node 5 of cell A1, and cell A1 is used in node 10 of cell A2. In the case of cell A1, the use of B1 has to be extended to indicate which branch of the conditional statement is being executed. In other words, there are in fact two uses of B1 in node 5, one where the left (true) branch is executed, and one where the right (false) branch is executed.

The definitions and uses can then be paired to give a definition-use association (du-association). For example A1, defined by the right (false) path, is used in node 10 of cell A2, and B1 is used in node 5 of cell A1 in executing the right (false) path.

2.3 A Test Adequacy Criterion

Rothermel et al (1997) identifies three possible test adequacy criteria: node and edge adequacy, cell dependence adequacy and dataflow adequacy. Dataflow adequacy, which uses du-associations, was shown to provide better cell coverage. For complete coverage, each du-association needs to be executed at least once. A table tracking these associations can be stored for each cell and the percentage executed is one testing metric.

2.4 Methodology for Spreadsheet Verification

Rothermel et al (2001) in their What You See Is What You Test (WYSIWYT) methodology sets out an algorithm that can be used to collect du-associations when a cell is changed in the spreadsheet.

Testing spreadsheets relies upon the developer knowing the correct output. Weyuker (1982) calls this the "Oracle Assumption," where the developer is assumed to be able to distinguish between correct and incorrect values, or at least be able to identify if they are in the correct range. The process of examining a cell's value to determine if it is correct (or seems correct) is termed "verifying" the cell.

Although the du-associations can be automatically recalculated every time the spreadsheet is changed, it is up to the developer to mark each cell after it has been verified. A simple checkbox was used in the top right hand corner of the cell. Once the cell was verified, the developer simply checked the box. This calls a recursive algorithm, which marks the appropriate du-associations as executed.

The level of testedness of each cell is indicated to the developer by colouring the cell border on a continuum

from red (un-tested) to blue (tested), based on the percentage of du-associations that have been "verified."

3 Building a Generalised Implementation

WYSIWYT provides a systematic way that spreadsheets developed in the research language Forms/3 can be tested. However, there is a much wider need for spreadsheet testing as highlighted by Panko (1995) who cites some examples where millions of dollars were lost as a result of spreadsheet errors.

In theory, the WYSIWYT algorithms can be adapted to any spreadsheet package that supports a suitable macro language. The purpose of our generalised implementation is to build a package independent implementation.

In this work, we built on Rothermel et al's algorithms, extending it and applying it to a popular commercial spreadsheet package. Although our implementation is linked to Micorosoft's Excel, it has been designed as a number of independent modules that can be reused with any Spreadsheet package that provides an interface to access cell contents.

Our implementation, illustrated in Figure 2, has three main components: the Spreadsheet Package, some Event Handler routines and the Verification Tool. In order to make the implementation spreadsheet package independent, a separate application, the Verification Tool, was developed in Java. This provided not only package, but also machine architecture independence. In order to provide the necessary visual feedback, the Verification Tool has a communication interface that can be tailored to specific spreadsheet packages.

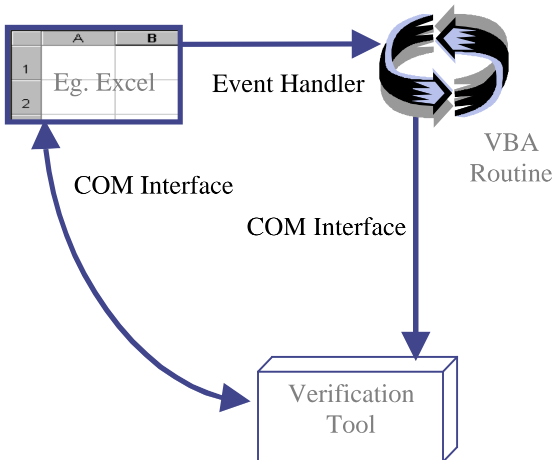

Figure 2: Component Interfaces

The event handlers are used to trap any events such as a cell formula changing or when the developer "verifies" a cell. As the event handlers trap events within the spreadsheet environment, they must be written using the built-in macro language.

For example: Someone enters a formula (eg "5 + 6") into cell A1 in an Excel spreadsheet. This triggers the "cellchanged" macro, which in turn calls the Verification Tool's "cellChanged" function, passing the cell address, A1, and its formula, "5+6", as parameters. The Verification Tool updates all the necessary du-associations and sets the new border colours of cells on the spreadsheet that have changed.

In the example, Microsoft's Excel spreadsheet was used, with the connections made using two COM interfaces. The Verification Tool was compiled using Visual J++, allowing the generation of a DLL file. By loading this file into Excel, the necessary functions could be called from the macro language. Similarly, an Excel DLL file can be referenced from the Verification Tool to get direct access to cell information and to be able to change border colours.

3.1 Parsing Cell Formulae

In the Forms/3 implementation, the cell information could be extracted straight from the evaluation engine. With an external implementation, an alternative method needed to be established to extract that information.

The first part of processing a new cell formula was to parse the formula up into its components. This could be achieved by manually creating a parser for a particular spreadsheet language. Instead, a grammar was written for Excel formulae in the format accepted by an automatic parser generator, such as Parr's ANTLR (Parr, 2000). This ensures portability of the tool as the syntax tree used in our verification tool is independent of the spreadsheet language.

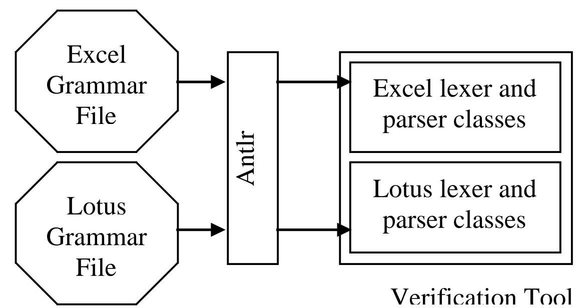

Figure 3: Automatic Parser Generator

Figure 3 illustrates the process of building the necessary lexer and parser classes for different spreadsheet languages.

3.2 Evaluation Information

Once the cell formula have been parsed it is necessary to determine the current execution path through multi-path cells. For example whether the left (true) or the right (false) path of cell A1, in Figure 1, is being executed. To determine which path was being executed would require the evaluation of the condition. We used the "evaluate" function available in Excel to simplify development of our tool. In principle, we could evaluate expressions using the syntax tree created by the parser. The large number of library functions available in a spreadsheet package makes writing a comprehensive evaluator time consuming. The "evaluate" function provided an efficient short cut to a working system, which enabled us to investigate other aspects of the tool. It is likely that most commercial systems will provide an equivalent capability.

For example: In the cell A1, to determine which path is being executed, the input string to the evaluate function would be "B1=5," which would return true or false. The Verification Tool interprets this and marks which definition is currently being executed (the Trace for that cell).

A similar technique was used to resolve names and cell references. Instead of relying upon cells having a set format, such as A1, the Verification Tool uses the getCell routine to determine the row and column number of either a cell name or cell address.

Figure 4 illustrates the different interface levels that were established to make it easier to tailor the Verification Tool to a specific spreadsheet package. The Spreadsheet Interface, which is a Java interface, defines the functions that need to be implemented. The General Interface is an abstract class and implements functions that are common to all spreadsheet packages, while the Excel and Lotus Interfaces extend the General Interface to fully implement the Spreadsheet Interface.

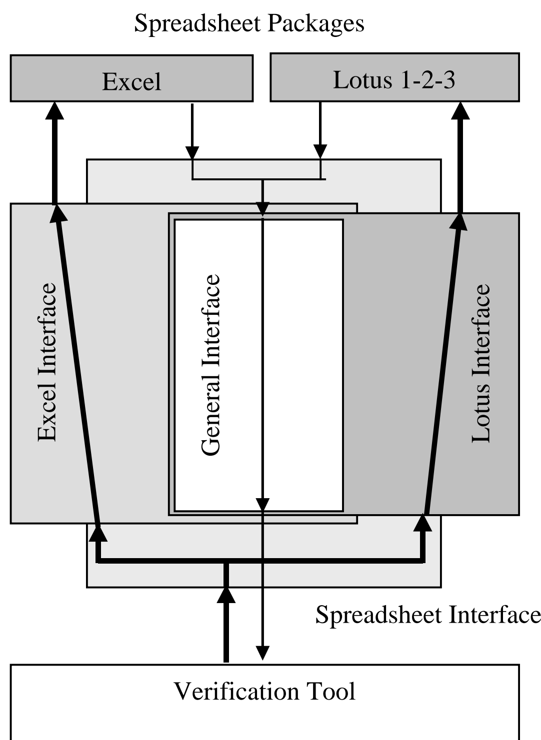

Figure 4: Interface Levels

There are five functions that can be called from the Spreadsheet packages:

New Creates a new instance of the Verification Tool, and establishes a reference to the spreadsheet package.

Restore If the Verification Tool has previously been used with the spreadsheet, this function allows the developer to load previous test data. Takes a string parameter, indicating the file to load from.

CellChange When a cell changes, this function should be called by the event handler. The cell address is provided as a parameter.

CellVerify Called when the developer verifies a cell. Again the cell address is passed as a parameter.

Cleanup Used to shut the Verification Tool down neatly. If a string is passed, this will also save test data.

These functions are all implemented by the General Interface because they are simply passing an event through to the Verification Tool. This is illustrated in Figure 4 by the thin arrows that converge on the General Interface and subsequently pass through to the Verification Tool

There are also four functions that can be called from the Verification Tool:

EvalString Used to evaluate a string expression to determine which path of a condition statement is being executed. Takes a string parameter and returns either true or false.

GetCell Spreadsheet packages use different naming conventions. Instead of trying to cater for specific cases, this function must be customised to suit the spreadsheet package in use. Takes a string cell address or name and returns a reference to the cell data structure in the Verification Tool.

CellUpdate Sometimes when a cell is changed it has a cascade effect on other cells. This function is used to extract updated values and formulae directly from the spreadsheet.

BorderColor Used to update the border colour of a cell. Takes a cell reference as a parameter.

These functions must all be implemented in the package specific Interface (eg the Excel Interface). The general process is illustrated in Figure 4 by the thick arrows that diverge to the package specific implementations. The arrows then follow through to the API of the appropriate spreadsheet package.

There will usually be other functions that need to be implemented within the package specific interfaces. For example there is a function **SetExcelRef** that passes a reference to the Excel application to the Excel Interface. The Excel Interface then uses the reference to update cell information. These implementations can also make function calls on the Verification Tool. For example the **GetCell** function implemented in the Excel Interface (which takes a string address as a parameter) calls the **GetCell** function on the Verification Tool (which takes row and column numbers as parameters) to get a

reference to the correct Cell object in the Verification Tool.

4 Presentation Issues

Chadwick et al (2000) showed that visual aids provide significant improvement in the way that spreadsheets are developed. Rothermel et al (1997) showed that the technique of colouring cell borders to indicate the level of testedness improves the developers' ability to generate test sets. However, simply colouring the cell border gives no indication of which du-associations are tested or not-tested. Hence there is no systematic way for the developer to derive the remaining test sets. Our Verification Tool includes a technique for spreadsheet developers to derive test cases. An adaptation of the cell relation graph was used to show the flow of control and relations between cells, allowing developers to trace execution paths and extract test case information.

An important distinction has to be made between constant cells and those containing references to other cells. For the purpose of this paper, constant cells refer to cells that do not reference any other cells. This does not mean that they must have constant formula. For example:

A1:	5	B1:	A1
A2:	sqrt(9)	B2:	sqrt(A2)
A3:	if(sqrt(9)=3,1,2)	B3:	if(A3=3,1,2)

A1-A3 are all *constant* cells, while B1-B3 are all *computation* cells. The term *input* cell is used synonymously with *constant* cell.

One of the most important characteristics of simple spreadsheets is that there is no recursion. This means that all computation cells must reference, even if it is indirectly, at least one constant cell. Further, it is possible to unravel all the cell references and place them in a tree hierarchy with the constant cells at the top and the final computation cell at the bottom.

In the example in Figure 5 the cell C1 has been traced back to the constant cells A1-A3. To test all of the du-associations in this example, we simply need to provide input values for the cells A1, A2 and A3. By verifying cell D1, all du-associations in this tree are marked as correct.

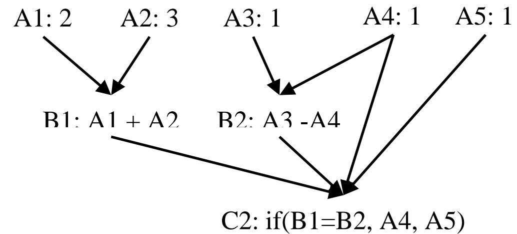

Figure 5: Reference Tree

Where there are multiple paths and more complex formulae, it becomes more difficult to trace the path back to the constant cells, which spurred the need for the tabular approach illustrated in Figure 6, a screen shot of a typical testing scenario.

The table in Figure 6 extends the reference tree by marking exactly where each cell is used in the subsequent cells. For example in the formula for C2, the black circle mark in the A4 column indicates that cell A4 is used in the "then" clause of the conditional statement. This table can be generated in a systematic way and can

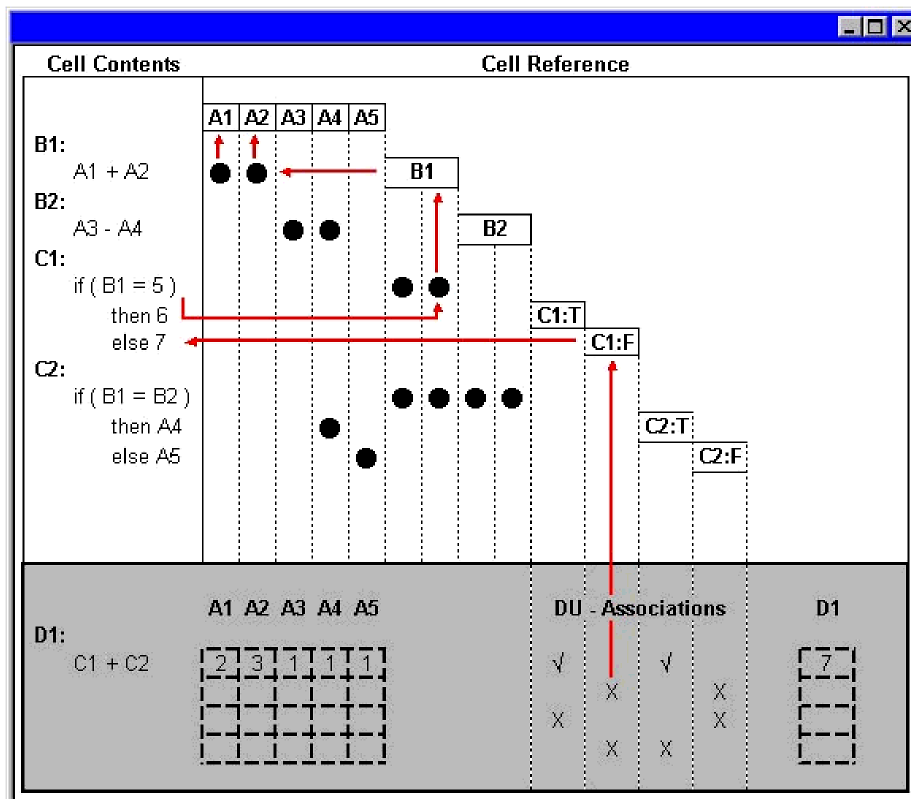

Figure 6: Screen shot shows test case derivation for a du-association.

subsequently be used to build test cases.

4.1 Deriving the Table for a Cell

The developer is trying to test cell D1. The first step in deriving this table is to trace the computation cell, D1, back to the constant cells. Bold indicates that the cell is a constant cell.

D1 references C1 and C2

C2 references B1, B2, **A4** and **A5**

C1 references B1

B2 references **A3** and **A4**

B1 reference **A1** and **A2**

Once the list has been computed, the cells are listed in the Cell Content pane (see Figure 6), in the reverse order that they were added to the list (ie B1, B2…), along with formulae. Where there is a conditional statement, multiple lines are used to place the condition on a separate line to the true and false paths.

The constant cells are placed in the Cell Reference pane in the line above the first computation cell. In line with the formula for each computation cell, the computation cell is added to the Cell Reference pane. Where there are multiple definitions (due to multiple paths in a conditional statement), the computation cell must be added to the Cell Reference pane multiple times. In such a case, each instance must be in a separate row and column, and should be numbered (eg C1:T and C1:F).

The last stage is to indicate, using black circles, ticks and crosses, where each cell is referenced. Cell D1 references the cells C1 and C2, but each of these has two definitions. A simple du-association table would have a separate test case for each du-association, even though a particular test case may execute multiple du-associations. This technique ensures that each du-association is executed at least once. The table in Figure 6 extends this idea by generating a new test case for each permutation of du-associations. Although this does not meet "all-paths" coverage criteria, as discussed by Rapps and Weyuker (1985), it is a stronger criterion than executing all the definition-use paths. A true "all-paths" criterion would generate test cases that trace all paths from the input cells through to the cell being tested (ie all paths from A1-A5 to D1). Our criteria covers all paths between the previous cells, C1 and C2, and the cell being tested, D1.

Each definition-use association is indicated with a cross initially, then a tick once the du-association has been executed and verified. Other cell references are simply marked with a black circle.

Where a cell is used in a condition there are two uses. These are marked twice, for the true and false execution paths. For example the cell B1 is used in the conditional statement "`=if(B1=5,6,7)`" in cell C1. There are two black circles marking the use of B1 by C1 in the execution of the True and False cases.

4.2 Generating Test Cases

In Figure 6, the first test case has been generated, which tests the use of C1:T and C2:T. The developer now needs to determine the test case for C1:F and C2:F. By clicking on the crosses, arrows appear that trace through all the du-associations. In the example, the developer has clicked on the cross marking the use of C1:F. Following the arrow from the cross up to the label, then to the left it can be seen that it is the false path of the condition statement for cell C1 that needs to be executed (condition is that B1=5). The arrows can be traced back to the two constant cells, A1 and A2, which sum to give B1. To generate the required test case, one of the conditions is that A1 and A2 do not sum to 5. This process can be repeated for C2:F

This process can be used to generate test cases for all of the du-associations for the cell D1. As the developer moves to a different cell, the graphical interface refreshes to display the information relevant to the new cell.

The technique for testing a cell can be summarised as:

1. Move to the cell in question and bring up the user interface.
2. For each du-association (cross), trace the uses back to the constant cells.
3. Enter values for the constant cells that execute the path followed in step 2.
4. Verify the cell (this should change the cross to a tick).
5. Repeat for the next du-association, until all have been verified (cell is completely tested)

4.3 Validating Cells

Rothermel et al use checkboxes located in the top right hand corner of the cell to indicate whether the cell has been verified. Initially the checkbox is empty. To verify the cell, the developer simply checks the box, which triggers a background process that marks the appropriate du-associations as verified.

The checkbox also provides visual feedback regarding the status of the cell. If it is empty, which it is initially, it indicates that the cell has not been verified. When the cell is verified a tick is placed in the checkbox. If other cells vary and the execution path of the cell changes, the checkbox changes to a question mark to indicate that the cell has been partially verified.

Unfortunately with commercial spreadsheet packages checkboxes cannot be placed on or even near a cell. Even if checkboxes could be placed on a cell, when a spreadsheet contains thousands of cells, it would quickly become confusing to navigate. The generalised implementation solves this problem by allowing the developer to double-right-click with their mouse, or by pressing Ctrl-Shift-V on the keyboard, with the cell highlighted, to indicate that the cell has been verified.

5 Session Recording

While WYSIWYT was created as an incremental technique that the developer should use throughout the development process, there is an algorithm for batch processing a spreadsheet. This simply collects cell information so that the developer can test an existing spreadsheet. If the spreadsheet is closed and re-opened, all the information would be lost.

As part of the generalised implementation, a save and restore technique was developed that allowed a session to be recorded. Using an XML data file to store the necessary information, previous sessions could be restored, allowing development and testing to continue.

When developing the structure of the data file it was observed that most of the cell information that was used to record state information in the Verification Tool was already saved in the spreadsheet. This meant that by simply reading the appropriate cells from the spreadsheet, the Verification Tool could be reinstated. However, the information that is not stored in the spreadsheet is which du-associations have been executed and the test case information.

5.1 DU-Associations

The du-associations that had been executed were recorded by converting the cell references into a string, that was subsequently stored in the data file. When restoring the Verification Tool, the executed du-association strings are compared with the du-associations generated from the spreadsheet. Where there is a match, the du-associations are marked as verified.

5.2 Test Cases

In addition to recording which du-associations have been verified, it is important to record test case data so that regression testing can be run on the spreadsheet. A test case usually consists of input and expected output data. Spreadsheets are usually thought of as a selection of cells that contain structured data. However, cells can be partitioned into constant and computation cells as discussed earlier.

The process of verifying a cell (only applicable to computation cells) is similar to specifying a test case for the spreadsheet. It states that "with the current values for the constant cells, this computation cell has the correct value." This information can be stored as test data for future reference.

In order to record this test case information, the input and output information needs to be clearly defined. The input data is relatively easy as it is simply the formulae in the constant cells. While it is possible to store the cell values for all cells on the path from the computation cell back to the constant cells, as output information, this is not relevant to the testing methodology being used. A better set of output information is the du-associations that should be executed by that set of input values.

To run a regression test the input data is applied to the constant cells and du-associations that are executed are observed. The results can be compared against the expected set of du-associations.

6 Future Work

To establish that the generalised implementation improves the development and testing processes, a controlled experiment needs to be carried out. A suggested format for this experiment would be for two groups of participants to build a spreadsheet from a brief requirements specification. The first, or control, group would be given a brief tutorial on how to design, build and test spreadsheets, but would not be given the generalised implementation. The second group would also be given the tutorial, but the generalised implementation would also be introduced and discussed. Both groups would be given a simple practice case study to try out the new techniques. This should be followed with the experiment case study.

Additional work also needs to be carried out to confirm that the generalised implementation can be used on other spreadsheet packages with only minor modifications to the interface classes.

6.1 Case Study

The generalised implementation concentrates on verifying all possible paths in the spreadsheet, and is hence known as a coverage tool. This means that the case study used in the experiment should have a reasonable number of alternate paths.

The other consideration for the experiment case study is that it should be a real world example. Participants in this experiment need to be able to relate to the case study problem as it makes it easier to understand the problem and thus less conceptually difficult. This means that errors found in the spreadsheets will be more likely to be a result of errors building the spreadsheet rather than in the earlier design work.

A good case study could be a tax calculation. With multiple tax rules and offsets to consider, the tax calculation provides a multitude of different combinations and paths to be tested. Not only does the tax calculation have a real-world application, the final product can be tested against the government distributed e-tax program.

While there are numerous tax rulings and adjustments in the legislation, these can be simplified so that the case can be completed within a reasonable time constraint. It is also important that the case study is easy to understand and that there is no ambiguity in the presentation of information to the developer.

7 Conclusion

Spreadsheets are one of the most invaluable decision support tools. Their ability to analyse data, generate summary information and graphs makes them essential for any business executive. While spreadsheets perform a vital role in organizations, there is very limited control over their development and use. Not only is there a need for higher accuracy spreadsheets, there is a need for a

development process to provide a guide for people developing spreadsheet.

Building on the WYSIWYT methodology proposed by Rothermel et al we have constructed a Verification Tool with specific emphasis placed on portability and the systematic generation of test cases. The ongoing research into this area should provide empirical evidence to support the use of this implementation across a number of different spreadsheet packages.

8 References

BURNETT, M., SHERETOV, A., and ROTHERMEL, G. (1999): Scaling up a 'What You See Is What You Test' Methodology to Spreadsheet Grids, *Proc. 1999 IEE Symposium on Visual Languages*, pp 30-37.

CHADWICK, D., KNIGHT, B. and RAJALINGHAM, K. (2000): Quality Control in Spreadsheets: A Visual Approach Using Color Codings To Reduce Errors in Formulae, *Proc. Eighth International Conference on Software Quality Management SQM'99*, Greenwich, London, British Computer Society.

EDWARDS, D., RAJALINGHAM, K., CHADWICK, D. and KNIGHT, B. (2000): Quality Control in Spreadsheets: A Software Engineering-Based Approach to Spreadsheet Development, *Proc. Thirty-Third Hawaii International Conference on System Sciences*, Maui, Hawaii, CD-ROM, IEEE Computer Society.

KNIGHT, B., CHADWICK, D. and D RAJALINGHAM, K. (2000): A Structured Methodology for Spreadsheet Modelling, *Proc. EuSpRIG 2000 Symposium on Spreadsheet Risks, Audit and Development Methods*, Greenwich, London, 43-50, University of Greenwich.

PANKO, R. (1995): Finding Spreadsheet Errors, *Information Week*, Final Word, May 20:100.

PANKO, R., and HALVERSON, R. (1996): Spreadsheets on Trial: A Framework for Research on Spreadsheet Risks. *Proc. Twenty-Ninth Hawaii International Conference on System Sciences*, Kihei, Hawaii, Los Alamitos, CA, **II**:326-335, IEEE Computer Society Press.

PARR, T. (2000): Another Tool for Language Recognition (ANTLR), *www.antlr.org*.

RAJALINGHAM, K., CHADWICK, D., KNIGHT, B. and EDWARDS, D. (1999): An Integrated Spreadsheet Engineering Methodology (ISEM). *Proc. IFIP TC11 WG11.5 Third Working Conference on Integrity and Internal Control in Information Systems*, Amsterdam, The Netherlands, 41-58, Kluwer Academic Publishers.

RAJALINGHAM, K., CHADWICK, D. and KNIGHT, B. (2000): Classification of Spreadsheet Errors, *British Computer Society (BCS) Computer Audit Specialist Group (CASG) Journal*, **10**(4):5-10.

RAPPS, S. and WEYUKER, E. (1985): Selecting Software Test Data Using Data Flow Information, *IEEE Transactions on Software Engineering*, **11**(4):367-375.

REICHWEIN, J., ROTHERMEL, G. and BURNETT, M. (1999): Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging, *Proc. 2nd Conference on Domain-Specific Languages*, 25-38.

ROTHERMEL, G., LI, L., and BURNETT, M. (1997): Testing Strategies for Form-Based Visual Programming, *Proc. Eighth Int'l Symposium on Software Reliability Engineering*, 96-107.

ROTHERMEL, K., COOK, C., BURNETT, M., SCHONFELD, J., GREEN, T. and ROTHERMEL, G. (2000): WYSIWYT Testing in the Spreadsheet Paradigm: An Empirical Evaluation, *Proc. 22nd Int'l Conference on Software Engineering*, 230-239.

ROTHERMEL, G., BURNETT, M., LI, L., DUPUIS, C. and SHERETOV, A. (2001): A Methodology for Testing Spreadsheets, *ACM Trans. on Software Engineering and Methodology* 10, 110-147.

WEYUKER, E. (1982): On Testing Non-Testable Programs, *The Computer Journal*, **25**(4):465-470.