Mutation Testing of Spreadsheets[*]

Robin Abraham and Martin Erwig
School of EECS, Oregon State University
[abraharo|erwig]@eecs.oregonstate.edu

August 1, 2006

Abstract

We present a catalog of mutation operators for spreadsheets drawn from research into mutation testing for general purpose programming languages and from spreadsheet errors that have been reported in literature. These operators are integrated into a system, called μTest, which allows users to create and maintain spreadsheet test cases. Three approaches to handling regions within spreadsheets are discussed, and we present a case study of how our system can be used to carry out mutation testing of spreadsheets. In addition to being useful in mutation testing of spreadsheets, the operators can be used in evaluation of error-detection tools and also for seeding spreadsheets with errors for empirical studies.

1 Introduction

Spreadsheets are among the most widely used programming systems [54]. Studies have shown that there is a high incidence of errors in spreadsheets [20], up to 90% in some cases [49]. Some of these errors have high impact leading to companies and institutions losing millions of dollars [55, 56, 28]. In this context, it is quite surprising that the by far most widely used spreadsheet system, Microsoft Excel, does not have any explicit support for testing. In particular, Excel spreadsheets do not have any provision by which the user can create and run a suite of tests. Moreover, Excel lacks any mechanism by which the user can associate a test suite with a spreadsheet or use more sophisticated techniques like regression and mutation testing. Therefore, users are forced to carry out ad hoc testing of their spreadsheets and come away with a very high level of confidence about the correctness of their spreadsheets [45, 44].

To reduce the incidence of errors in spreadsheets, research into spreadsheets has focussed on the following areas.

1. Recommendations for better spreadsheet design [50, 60, 33, 47, 49]
2. Auditing spreadsheets to detect and remove errors [43, 53, 36]
3. Automatic consistency checking [24, 7, 10, 14, 16, 1]
4. Error prevention techniques [22, 21]
5. Testing [51, 26, 15]

[*]This work is partially supported by the National Science Foundation under the grant ITR-0325273 and by the EUSES Consortium (`http://EUSESconsortium.org/`).

Traditional Software Engineering research has made considerable progress in the area of testing. Part of the EUSES [25] research collaboration's goal is to bring these benefits to the realm of spreadsheets. An already exisiting testing framework for spreadsheets is the WYSIWYT methodology [51] that has been implemented in Forms/3 environment [13]. The benefits of the WYSIWYT approach (and associated systems) have been demonstrated empirically [51, 52, 48]. Even so, they are not within the reach of the legions of spreadsheet users yet since Forms/3 is a research vehicle and not commercially available. Efforts are underway to integrate the WYSIWYT methodology into Excel, so this situation might change in the future. Forms/3 also has a system called "Help Me Test" that generates test cases automatically to help users test their spreadsheets [26]. In WYSIWYT, users provide input values to the system and then mark the output cells with a ✓ if the output is correct, or a ✗ if the output is incorrect. The system stores the test cases implicitly and gives the user feedback about the likelihood of faults in cells through cell shading—cells with higher fault likelihood are shaded darker than those with lower fault likelihood. WYSIWYT uses du-adequacy (definition-use adequacy) as the criterion for measuring the level of testedness of the spreadsheet. The testedness is reported to the user through a progress bar, that ranges from 0% to 100% testedness, and coloring of the cell border that indicates how many of the du pairs for the formula in the cell have been tested (ranging from red when none of the du pairs have been tested to blue when all the du pairs have been tested).

Various comparisons are available to guide the choice of strategies for testing programs [58]. Two aspects guide the choice of one *test adequacy criterion*[1] over another: *effectiveness* and *cost*. Effectiveness of a test adequacy criterion is related to the reliability of programs that meet the criterion. In other words, a test adequacy criterion is effective if it stops testing only when few failures remain in the program. The cost of an adequacy criterion, on the other hand, is related to the difficulty of satisfying the criterion and the human effort involved. Testing effort is always a tradeoff between the two since an ineffective criterion is a poor choice no matter how low the cost might be. Similarly, a very effective criterion is not a practical choice if the costs involved are prohibitive. One factor cited against using mutation coverage as an adequacy criterion is that mutation testing is costly. In the absence of automatic test case generation, this problem might be an even greater cause for concern in the context of end users testing their spreadsheets. However, empirical studies comparing data-flow and mutation testing have shown that mutation-adequate tests detect more faults [41]. In the empirical evaluation we carried out (described in Section 7), we saw some cases of faults that were discovered by a mutation adequate test suite that would not be discovered by a du-adequacy criterion. These results present a strong case in favor of mutation testing. Moreover, support for generating test cases automatically [26, 19, 37] would lower the costs involved in mutation testing.

In this paper, we propose a suite of mutation operators for spreadsheets. We describe an environment for specifying test cases within Microsoft Excel and demonstrate the use of these operators for mutation testing of spreadsheets. Along the lines of the discussion in [42], the ideal testing environment would be a system that inspects the spreadsheet and generates an *effective*[2] set of test cases (inputs and the corresponding outputs) that can be inspected to find failures. The input–output pairs can be used for debugging the spreadsheet, either manually or with the help of debugging tools. For example, in earlier

[1]A test adequacy criterion is basically a set of rules that determine if the testing is complete for a given program and specification.

[2]"Effective" from the point of view of some coverage criterion.

work, we have developed the spreadsheet debugger GoalDebug [2] that allows the user to mark a cell output as incorrect and then specify the expected value for the cell. The system uses this information to generate a list of change suggestions. The application of any one of the suggested changes would result in the expected value being computed in the marked cell. We believe that the testing and debugging approaches would complement each other very well since the expected output value from the test case can be used to generate change suggestions. In the approach described in this paper, the test cases are explicitly represented within *test sheets*, and our system (μTest) uses mutation adequacy as the coverage criterion.

The immediate technical contributions of this paper are threefold:

1. Suite of mutation operators
2. An Excel-based environment for specifying spreadsheet test cases and carrying out mutation testing
3. Strategies that allow more efficient testing of spreadsheet regions

The suite of mutation operators allows us to evaluate spreadsheet test suites on the basis of a mutation adequacy criterion. The operators also allow us to assess the effectiveness of automatic spreadsheet tools.

We present related work in the next section. In Section 3 we describe some background on mutation testing. We present the set of mutation operators we have developed for spreadsheets in Section 4. In Section 5 we discuss how μTest can be used to maintain spreadsheet test cases. In Section 6 we describe three strategies for testing *cp-similar* regions within spreadsheets. We describe two scenarios in which the mutation operators would be useful—in the evaluation of test suites (in Section 7), and in the automatic evaluation of tools (in Section 8). We present conclusions and discuss directions of future work in Section 9.

2 Related Work

Over the years, research into mutation testing has led to the development of mutation operators for several general-purpose programming languages, for example, Mothra (for FORTRAN programs) [18], Proteum (for C programs) [17], and μJava (for Java programs) [34].

As pointed out in [42], one of the main reasons why mutation testing has not been more widely adopted by industry is the overhead involved in mutation analysis and testing. The computational expense involved in generating and running the many mutant programs against the test cases is very high. Another problem with this approach is the potentially huge manual effort involved in detecting equivalent mutants and in developing test cases to meet the mutation adequacy criteria. Many researchers have focussed their effort on developing approaches for lowering the computational cost of mutation testing [59]. We briefly describe some approaches in the following.

- In general-purpose programming languages, mutant programs have to be compiled (or interpreted) prior to running the tests cases. Schema-based mutation was proposed to lower this cost by using a *metamutant* (a single program that, in effect, incorporates all the mutants) [57]. Since there is no explicit compilation step in the case of spreadsheets, this cost is not much of a cause for concern.

- In general, during mutation testing, mutant programs are run to completion, and their output is compared with the output of the original program. *Weak mutation* is an approximation technique in which the internal states of the mutant and the original program are compared immediately after execution of the mutated portion of the program [32]. This approach reduces the execution time during mutation testing in general-purpose programming languages. However, in the absence of recursion or iteration, the execution time is not so much of a cause for concern in spreadsheets.

- Based on the original proposal in [35], studies were carried out using Mothra that showed that of the 22 operators, 5 turn out to be "key" operators. That is, using the 5 key operators achieves testing that is almost as rigorous as that achieved while using the entire set of 22 operators [39]. This observation led the authors to propose the *selective mutation* approach in which only those mutants that are truly distinct from the other mutants are used in mutation testing to make the approach more economical.

- Techniques have been proposed that use a sampling of the set of non-equivalent mutants. These techniques have been shown to be only marginally less effective at fault detection when compared to the complete set of mutants [6, 11]. On the other hand, using the sampling techniques reduces the effort involved in mutation testing.

- To minimize the effort involved in developing new test cases, researchers have also focussed on algorithms for automatic test generation that would result in test suites that are close to being mutation adequate [19, 37].

- Some of the generated mutants might be equivalent to the original program, that is, they produce the same outputs for the same set of inputs. Equivalent mutants do not contribute to the generation of test cases and require a lot of time and effort from the tester since earlier approaches required the tester to go through and remove them by hand. No automated system would be able to detect all the equivalent mutants since the problem is in general undecidable. However, there has been some work that allows the detection of up to 50% of equivalent mutants in general-purpose programming languages [40, 29].

The applicability of mutation to the evaluation of testing has been explored in [9], and the authors have shown that generated mutants mirror real faults. However, mutants might be different from hand-seeded faults, which in turn seem to be harder to detect than real faults.

The widespread occurrence of errors in spreadsheets has motivated researchers to look into the various aspects of spreadsheet development. Some researchers have focussed their efforts on guidelines for designing better spreadsheets so errors can be avoided to some degree [50, 60, 33, 47, 49]. Such techniques are difficult to enforce and involve costs of training the user. While this might be feasible within companies, it is not feasible in the context of end users working in isolation.

We have looked at ways to prevent errors in spreadsheets by automatically enforcing specifications. This approach, implemented in the Gencel system [22, 23], captures a spreadsheet and all its possible evolutions in a template developed in the Visual Template Specification Language (ViTSL) [5]. Any spreadsheet generated using Gencel is guaranteed to be free from reference, range, and type errors. The user does not ever have to edit the spreadsheet formulas since they are automatically generated by the system based on the template. In the context of companies or other large organizations, the templates can be created by some domain expert (for example, the chief accountant of a company) and passed on to the other users (for example, junior clerks).

Most of the research that has been done in the area of spreadsheets has been targeted at removing errors from spreadsheets once they have been created. Following traditional Software Engineering approaches, some researchers have recommended code inspection for detection and removal of errors from spreadsheets [43, 53, 36]. However, such approaches cannot give any guarantees about the correctness of the spreadsheet once the inspection has been carried out, and empirical studies have shown that individual code inspection only catches 63% of the errors whereas group code inspection catches 83% of the errors [43]. Code inspection of larger spreadsheets might prove tedious, error prone, and prohibitively expensive in terms of the effort required.

Automatic consistency-checking approaches have also been explored to detect errors in spreadsheets. Most of the systems require the user to annotate the spreadsheet cells with extra information [7, 10, 14, 16, 24]. We have developed a system, called UCheck, that automatically infers the labels within the spreadsheet and uses this information to carry out consistency checking [1], thereby requiring minimal effort from the user.

3 Mutation Testing

Mutation analysis, strictly speaking, is a way to measure the quality of test suites. The actual testing of the software is a side effect that results from new test cases being designed to kill more and more mutants. In practical terms however, the software is well tested by the test suite, or the test cases in the suite do not kill mutants [42].

In mutation testing, faults are inserted into the program that is being tested. Each seeded fault generates a new program, a mutant[3], that is slightly different from the original. The idea behind mutation testing is that the test suite is adequate if it detects all the mutants. The fault seeding is done by means of *mutation operators*. Ideally, the mutation operators would be *adequate* in the sense that they would be able to show how effective the test suite is, and *efficient* in terms of testing time.

Let P be a program that has the input domain D and codomain D'. Running P on an input $x \in D$ is written as $P(x)$. We also use $P(x)$ to refer to the result of the program run, that is, $P(x) \in D'$. A *test case* is a pair $(x, y) \in D \times D'$. A program *passes* a test $t = (x, y)$ if $P(x) = y$, otherwise P *fails* the test t. A *test set* is a set of tests $T = \{t_1, \ldots, t_n\}$. A program *passes* a test set T if P passes every $t \in T$, otherwise P *fails* the test set T.

Assume we make n copies of P and introduce a single mutation in each one to get the mutated versions P_1 through P_n. Let $T \subset D \times D'$ be a passing test set, that is, P passes or satisfies every test in T. To test the *mutation adequacy* of the test set T, we run it against each of the mutants. We say a mutant is *killed* when it fails T, whereas mutants that pass T are said to be *alive*. The basic assumption is that if T kills a mutant, then it will detect real unknown faults as well. Extending the idea, if T kills all the *non-equivalent* mutants, it would be capable of detecting a wide variety of unknown faults as well. A mutant P_i is said to be equivalent to P if $\forall x \in D, P(x) = P_i(x)$, and this equivalence is expressed as $P_i \equiv P$. Obviously, an equivalent mutant cannot be killed by testing. *Mutation adequacy*

[3]First-order mutants are created by inserting one fault into the program. Higher-order mutants can be created by inserting more than one fault into the program.

or *mutation score* is computed as

$$\frac{\text{number of killed mutants}}{\text{total number of non-equivalent mutants}} \times 100\%$$

In choosing the mutation operators, we make the same assumptions laid out in [12] that are briefly described below.

1. The *competent programmer hypothesis*: Given a functional specification f, the programmer is competent enough to produce a program P that is within the immediate "neighborhood" of the program P^* that satisfies f. Any program that is far removed from the neighborhood of P^* is called *pathological.* The hypothesis allows us to limit the programs we need to consider by excluding the pathological ones.

2. The *coupling effect*: Complex faults within a program are linked to simple faults in such a way that a test suite that detects all simple faults within a program will also detect most complex faults.

It has been shown that if the program is not too large, only a very small proportion of higher-order mutants survives a test set that kills all the first-order mutants [38, 30, 31]. This result is helpful for cases in which multiple faults lead to a single point of failure—if we have tests that detect the isolated faults, we would be able to detect the compound fault with a high level of probability even when they are not isolated.

All these notions pertaining to programs in general-purpose programming languages have to be slightly adjusted for spreadsheets. For the purpose of this paper, a spreadsheet program can be considered as given by a set of formulas that are indexed by cell locations taken from the set $A = \mathbb{N} \times \mathbb{N}$. A set of addresses $s \subseteq A$ is called a *shape.* Shapes can be derived from references of a formula or from the domain of a group of cells and provide structural information that can be exploited in different ways.

The structure of formulas is not essential, except for the fact that formulas may contain references to other cells. Therefore, we assume a set F that contains all possible formulas. F includes the set of values V, that is $V \subseteq F$. We further assume a function $\sigma : F \to 2^A$ that computes for a formula the addresses of the cells it references. We call $\sigma(f)$ the *shape* of f.

A spreadsheet is given by a partial function $S : A \to F$ mapping cell addresses to formulas (and values). The function σ can be naturally extended to work on cells and cell addresses by $\sigma(a, f) = \sigma(f)$ and $\sigma(a) = \sigma(S(a))$, that is, for a given spreadsheet S, $\sigma(a)$ gives the shape of the formula stored in cell a. The cells addressed by $\sigma(c)$ are also called c's *input cells.*

To apply the view of programs and their inputs to spreadsheets, we can observe that each spreadsheet contains a program together with the corresponding input. More precisely, the *program part* of a spreadsheet S is given by all of its cells that contain (non-trivial) formulas, that is, $P_S = \{(a, f) \in S \mid \sigma(f) \neq \varnothing\}$. This definition ignores formulas like $2 + 3$ and does not regard them as part of the spreadsheet program, because they always evaluate to the same result and can be effectively replaced by a constant. Correspondingly, the *input* of a spreadsheet S is given by all of its cells containing values (and locally evaluable formulas), that is, $D_S = \{(a, f) \in S \mid \sigma(f) = \varnothing\}$. Note that with these two definitions we have $S = P_S \cup D_S$ and $P_S \cap D_S = \varnothing$.

Based on these definitions we can now say more precisely what test cases are in the context of spreadsheets. A *test case* for a cell (a, f) is a pair (I, v) consisting of values for all the cells referenced by f, given by I, and the expected output for f, given by $v \in V$.

Since the input values are tied to addresses, the input part of a test case is itself essentially a spreadsheet, that is $I : A \to V$. However, not any I will do; we require that the domain of I matches f's shape, that is, $dom(I) = \sigma(f)$. In other words, the input values are given by cells whose addresses are exactly the ones referenced by f. Running a formula f on a test case means to evaluate f in the context of I. The evaluation of a formula f in the context of a spreadsheet (that is, cell definitions) S is denoted by: $[\![f]\!]_S$.

Now we can define that a formula f *passes* a test $t = (I, v)$ if $[\![f]\!]_I = v$. Otherwise, f *fails* the test t. Likewise, we say that a cell (a, f) passes (fails) t if f passes (fails) t. Since we distinguish between testing individual formulas/cells and spreadsheets, we need two different notions of a test set. First, a *test set* for a cell c is a set of tests $T = \{t_1, \ldots, t_n\}$ such that each t_i is a test case for c. Second, a *test suite* for a spreadsheet S is a collection of test sets $\mathcal{T}_S = \{(a, T_a) \mid a \in dom(P_S)\}$ such that T_a is a test set for the cell with address a. A test suite $\mathcal{T}_S$ in which each test set T_a contains just a single test (that is, $|T_a| = 1$) is also called a *test sheet* for S. Running a formula f on a test set T means to run f on every $t \in T$. Running a spreadsheet S on a test suite $\mathcal{T}_S$ means to run for every $(a, f) \in P_S$, the formula f on the test set $\mathcal{T}_S(a)$.

A formula f (or cell c) *passes* a test set T if f (or c) passes every test $t_i \in T$. Likewise, a spreadsheet S *passes* a test suite $\mathcal{T}_S$ if for every $(a, f) \in P_S$, f passes $\mathcal{T}_S(a)$. Otherwise, S *fails* the test suite $\mathcal{T}_S$.

Finally, the concepts related to mutants can be transferred directly to formulas and spreadsheets. The notion of killing mutants by test sets has to be distinguished again for formulas/cells and spreadsheets, that is, a formula mutant f_i is *killed* by a test set T if f_i fails T. Likewise, a spreadsheet mutant S_i is *killed* by a test suite $\mathcal{T}_S$ if S_i fails $\mathcal{T}_S$.

4 Mutation Operators for Spreadsheets

Mutation operators are typically chosen to satisfy one or both of the following criteria (or goals).

1. They should introduce syntactical changes that replicate errors typically made by programmers.
2. They should force the tester to develop test suites that achieve standard testing goals such as statement coverage or branch coverage.

To meet the first goal, we have chosen mutation operators that reflect errors reported in the spreadsheet literature [8, 44]. To ensure that the operators meet the second goal, we have included operators that have been developed for general-purpose programming languages. Moreover, we have carried out the evaluation studies reported in Section 7, which show that mutation adequacy is a strong criterion for the level of testedness of the spreadsheet.

A “standard” set of 22 mutation operators for FORTRAN have been proposed in [39]. A subset of these that are applicable to spreadsheets are shown in Table 1. As mentioned in Section 2, it has been shown empirically in [39] that test suites killing mutants generated by the 5 operators ABS, AOR, LCR, ROR, and UOI are almost 100 % mutation adequate compared to the full set of 22 operators.

In adapting mutation operators for general-purpose languages to the spreadsheet domain, we draw the parallels shown in Table 2. The mutation operators we propose for spreadsheets are shown in Table 3. A few of them have been directly taken from the

Operator	Description
ABS	*ABS*olute value insertion
AOR	*A*rithmetic *O*perator *R*eplacement
CRP	*C*onstants *ReP*lacement
CSR	*C*onstants for *S*calar variable *R*eplacement
LCR	*L*ogical *C*onnector *R*eplacement
ROR	*R*elational *O*perator *R*eplacement
SCR	*S*calar for *C*onstant *R*eplacement
SDL	*S*tatement *DeL*etion
SRC	*S*ou*R*ce *C*onstant replacement
SVR	*S*calar *V*ariable *R*eplacement
UOI	*U*nary *O*perator *I*nsertion

Table 1: Subset of mutation operators for FORTRAN

General-purpose language	Spreadsheets
Input data	Data cells
Constants	Data cells
Variables	Cell references
Statement	Cell formula
Output data	Output of formula cells

Table 2: Correspondence of constructs in general-purpose programming languages and in spreadsheets

operators shown in Table 1. In some cases, we had to adapt the operators to match the constructs in spreadsheets. (The original operators are mentioned in parenthesis.)

In the current version of the system, we do not have a mechanism that would allow us to distinguish between the data cells that are equivalent to input data from the data cells that are equivalent to constants within a program written in a general-purpose programming language. We therefore treat all data cells within the spreadsheet as inputs to the spreadsheet *program*, which essentially is the collection of cells that have formulas or references to other cells. In the description of the operators, the use of "constant" (in the operators CRP, CRR, and RCR) refers to constants within formulas.

We consider references in spreadsheets equivalent to scalar variables in general-purpose programming languages. Therefore the equivalent of a scalar variable replacement in a general-purpose programming language would be a change in reference in spreadsheets. Mutation operators for references should ensure that they do not introduce cyclic references in spreadsheets since these are reported by spreadsheet systems like Excel. In the following we discuss three approaches to refine mutations of references.

1. It might be reasonable to only change a reference to another one that references a cell of the same type.[4] For example, a reference to a cell that has a numerical value should only be changed to a reference to a cell with a numerical value. On the other hand, it might make sense in some cases to remove this restriction on types since

[4]A type checker would be able to enforce this property at compile time in a general-purpose programming language.

Operator	Description
ABS	*ABS*olute value insertion
AOR	*A*rithmetic *O*perator *R*eplacement
CRP	*C*onstants *ReP*lacement
CRR	*C*onstants for *R*eference *R*eplacement (adapted from CSR)
LCR	*L*ogical *C*onnector *R*eplacement
ROR	*R*elational *O*perator *R*eplacement
RCR	*R*eference for *C*onstant *R*eplacement (adapted from SCR)
FDL	*F*ormula *DeL*etion (adapted from SDL)
FRC	*F*ormula *R*eplacement with *C*onstant
RFR	*ReF*erence *R*eplacement (adapted from SVR)
UOI	*U*nary *O*perator *I*nsertion
CRS	*C*ontiguous *R*ange *S*hrinking
NRS	*N*on-contiguous *R*ange *S*hrinking
CRE	*C*ontiguous *R*ange *E*xpansion
NRE	*N*on-contiguous *R*ange *E*xpansion
RRR	*R*ange *R*eference *R*eplacement
FFR	*F*ormula *F*unction *R*eplacement

Table 3: Mutation operators for spreadsheets

spreadsheet systems like Excel do not perform any systematic type checking, and spreadsheet programmers might actually have such errors in their spreadsheets.

2. In case of formulas that operate over cells within the same row (or column), it might be reasonable to change references only to other cells within the same row (or column) to reflect the user's lack of understanding of the specification. This refinement could be too restrictive if we are trying to model mechanical errors in which the user accidentally clicks a cell in the immediate neighborhood of the cell they meant to include in the formula.

3. While mimicking mechanical errors, it would be reasonable to change a reference to a cell to other cells in the immediate spatial neighborhood of the original reference. The "distance" between the original cell and the new cell could be considered a measure of the reference mutation, and could be tuned depending on the application.

In the current implementation of μTest, we impose the third constraint (and drop the second) discussed above since we are modeling mechanical errors in automatic evaluation of spreadsheet tools (see Section 8.1). Since Excel does not carry out type checking of formulas, we do not enforce the first constraint during mutation.

We discuss below the mutation operators we have included that are unique to the domain of spreadsheets.

Range mutation: Aggregation formulas in spreadsheets typically operate over a range of references. Ranges might be *contiguous* or *non-contiguous*. For example, the formula `SUM(A2:A10)` aggregates over the contiguous range from `A2` through `A10`, whereas the formula `SUM(A2,A5,A8,A11)` aggregates over the non-contiguous range that includes the references `A2`, `A5`, `A8`, and `A11`. We propose the following operators that mutate ranges.

1. Contiguous Range Shrinking (CRS): This operator shrinks a contiguous range by altering the reference at its beginning or end.

2. Non-contiguous Range Shrinking (NRS): This operator removes any reference from a non-contiguous range.
3. Contiguous Range Expansion (CRE): This operator expands a contiguous range by altering the reference at its beginning or end.
4. Non-contiguous Range Expansion (NRE): This operator introduces an extra reference into a non-contiguous range.
5. Range Reference Replace (RRR): This operator replaces a reference in a non-contiguous range with another reference not in the range.

We treat contiguous and non-contiguous ranges differently because of the competent programmer hypothesis—we only consider mutations to contiguous ranges that affect the first or last reference whereas mutations to non-contiguous ranges can happen to any reference in the range.

Formula Replacement: In addition to the reference and range operators, the FRC (Formula Replacement with Constant) operator is unique to spreadsheets. It overwrites the formula in a cell with the computed output value from the formula. It has been observed in empirical studies that when the spreadsheet specifications are not well understood, users sometime overwrite formula cells with constant values as a "quick fix" to get the output they expect. The FRC operator has been included in the suite of mutation operators to model this kind of error.

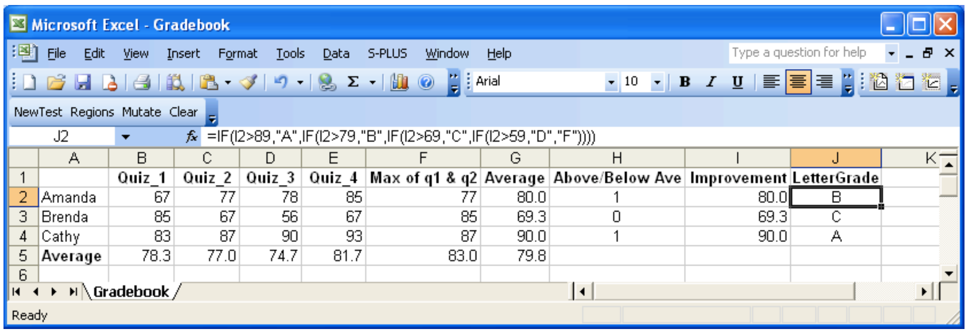
Function Mutation: Due to lack of programming expertise, users might use the incorrect function in their spreadsheet formulas. For example, the user might use SUM (and forget to divide by COUNT of the cells) instead of AVERAGE. We include the Formula Function Replacement (FFR) operator to simulate this effect. As in the case of mutation of cell references, we use a *distance* measure for the FFR operator. For example, replacement of SUM with AVERAGE (or vice versa) seems more likely than replacing SUM (or AVERAGE) with an IF-statement.

5 Maintaining Spreadsheet Test Cases in Excel

Consider the second row of the spreadsheet shown in Figure 1. Cells A2, B2, C2, D2, and E2 are all data cells. F2 contains the formula IF(B2<C2,C2,B2). The input values in the cells B2 and C2 together with the expected output value for F2 constitute a test case for F2. The row can be used to specify test cases for the formulas in F2, G2, H2, I2, and J2.

We have defined a test sheet in Section 3 formally as a test suite for a spreadsheet that contains one test case for each formula in the spreadsheet. A test sheet can be immediately obtained from a spreadsheet from the values (the values in the data cells and the computed output values in the formula cells) in the spreadsheet. To create a new test sheet in μTest, the user can enter input values in the spreadsheet and click the button labeled "NewTest". This action causes the system to copy the entire spreadsheet area and paste only the values into a new worksheet. The new worksheet is assigned a name starting with "Test". A test sheet created from the spreadsheet in Figure 1 is shown in Figure 2.

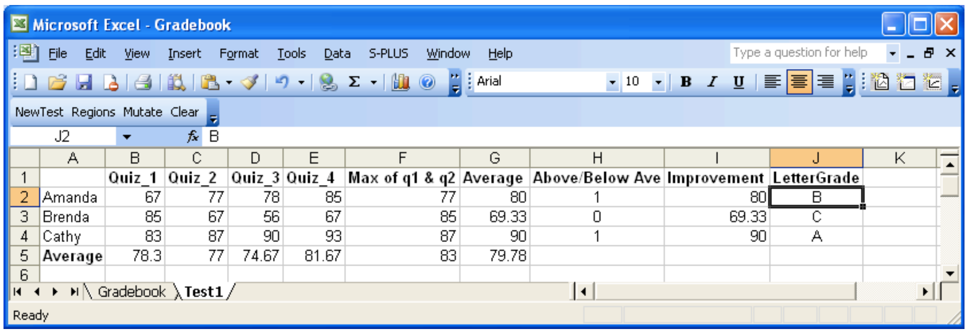
In addition to creating spreadsheet test cases in the form of test sheets, the user can interact with μTest using the buttons on the mutation testing menu bar: "Regions", "Mutate", and "Clear". The implementation architecture is the same one we have adopted in our other spreadsheet tools [22, 1, 2, 4]. We use the Excel spreadsheet as the frontend for capturing user inputs and have an inference engine, implemented in Haskell [46], as the


Microsoft Excel - Gradebook
File Edit View Insert Format Tools Data S-PLUS Window Help
Type a question for help
Arial 10
NewTest Regions Mutate Clear
Gradebook
Ready

J2 =IF(I2>89,"A",IF(I2>79,"B",IF(I2>69,"C",IF(I2>59,"D","F"))))

	A	B	C	D	E	F	G	H	I	J
1		Quiz_1	Quiz_2	Quiz_3	Quiz_4	Max of q1 & q2	Average	Above/Below Ave	Improvement	LetterGrade
2	Amanda	67	77	78	85	77	80.0	1	80.0	B
3	Brenda	85	67	56	67	85	69.3	0	69.3	C
4	Cathy	83	87	90	93	87	90.0	1	90.0	A
5	Average	78.3	77.0	74.7	81.7	83.0	79.8			
6										

Figure 1: Grade sheet.


Microsoft Excel - Gradebook
File Edit View Insert Format Tools Data S-PLUS Window Help
Type a question for help
Arial 10
NewTest Regions Mutate Clear
Gradebook
Test1
Ready

J2 B

	A	B	C	D	E	F	G	H	I	J
1		Quiz_1	Quiz_2	Quiz_3	Quiz_4	Max of q1 & q2	Average	Above/Below Ave	Improvement	LetterGrade
2	Amanda	67	77	78	85	77	80	1	80	B
3	Brenda	85	67	56	67	85	69.33	0	69.33	C
4	Cathy	83	87	90	93	87	90	1	90	A
5	Average	78.3	77	74.67	81.67	83	79.78			
6										

Figure 2: First test sheet.

backend. The spreadsheet uses VBA modules to communicate with the backend inference engine.

Clicking on the button labeled "Regions" allows the users to specify cp-simlar regions (vertically or horizontally repeating blocks of cells) within their spreadsheets. (Regions, and their impact on testing, are discussed in the next section.) In the current version of the system, the user has to click this button and explicitly *enable* use of regions by specifying cp-similar blocks. Recently, we have developed a system that allows the automatic detection of regions within spreadsheets [4]. Integrating μTest with this system would facilitate the automatic inference of regions within the spreadsheet and thereby further lower the amount of user interaction required to run the mutation tests. This integration is planned for a future version of μTest.

The user can click the button labeled "Mutate" to initiate mutation testing. Once the testing is complete, the system reports statistics on the test run that includes the following information:

1. number of mutants generated,
2. number of mutants killed,
3. number of mutants that survived the tests,
4. mutation coverage percentage, and
5. time taken.

The system also marks the formula cells with the mutants that have survived in those cells (see Figure 5). The user can inspect the marked cells to come up with more test cases targeted at the mutants that are alive. Clicking the "Clear" button erases all the mutation-testing feedback from the spreadsheet.

6 Testing Regions Within Spreadsheets

Consider the test case for the cell F2 that contains the formula IF(B2<C2,C2,B2). We have the values 67 (from B2) and 77 (from C2) as inputs and the expected output is 77. F3 contains the formula IF(B3<C3,C3,B3), and its test case in the test sheet has the values 85 (in cell B3) and 67 (in cell C3) as input and the expected output value 85. Since the formulas in F2 and F3 are *similar*, we can use the test case for F2 to test F3 and vice versa. Two cells are similar if their formulas could have resulted from a copy/paste action from one of the cells to the other. This requirement is the *cp-similarity* criterion described in [36, 15]. Extending this idea to rows, we see that from a spreadsheet program point of view, the rows 2, 3, and 4 are cp-similar since their formulas are cp-similar. In other words, we can consider the rows 2, 3, and 4 to be part of one cp-similar region.

Given a set of cp-similar formulas, we can identify any one of them as the region-representative formula and run test cases on it. Under this approach, regions of cells with formulas that are the result of copy-paste actions can be tested by testing a representative formula cell with the associated data cells. For example, we can pick the formula in F2 as the region-representative formula for F2, F3, and F4.[5] We can now run the test cases for F2, F3, and F4 on the formula in cell F2 and consider all three formulas equally well tested. This is the approach adopted in WYSIWYT for testing cp-similar regions [15]. The region information allows the system to come up with the two additional *derived test cases* (the test cases for F3 and F4) for the formula in F2.

For a more general example of cp-similar regions consider Figure 3. We define a *block* as a group of cells. Two blocks have been marked in the figure—the cells in the first block have been labeled b_1 and the cells in the second block have been labeled b_2. The two blocks can be considered cp-similar if one can be created by a simple copy-paste action from the other, in which case, we say the two blocks belong to the same cp-similar region. Within a cp-similar region, blocks can repeat vertically (as shown in Figure 3) and/or horizontally.

	A	B	C	D	E
1		b_1			
2	b_1	b_1	b_1	b_1	b_1
3	b_1		b_1	b_1	
4		b_2			
5	b_2	b_2	b_2	b_2	b_2
6	b_2		b_2	b_2	

Figure 3: Regions within spreadsheets

The shape of formulas and cells has been formalized in Section 3. The shape (σ) of a block has two components, the *core shape* and the *shadow*. We define the core shape (σ_c) of a block as the spatial arrangement of the cells within it, that is $\sigma_c(b) = \lceil dom(b) \rceil$, where $\lceil b \rceil$ moves the block b to the upper left corner. Formally, this means to adjust each

[5]We could have as well picked the formula in F3 or F4 as the region-representative formula.

cell index (c, r) in b to $(c - c_x + 1, r - r_y + 1)$ where c_x and r_y are the smallest column and row index, respectively, occurring in b. For example, for the blocks in Figure 3, we have $\sigma_c(b_1) = \sigma_c(b_2) = \{\mathsf{B1}, \mathsf{A2}, \mathsf{B2}, \mathsf{C2}, \mathsf{D2}, \mathsf{E2}, \mathsf{A3}, \mathsf{C3}, \mathsf{D3}\}$. The shadow (σ_s) of a block b is the list of cells referenced by formulas within the block b, that is, $\sigma_s(b) = \bigcup_{(a,f)\in b} \sigma(f)$. The shape of the block b is therefore given by the pair $\sigma(b) = (\sigma_c(b), \sigma_s(b))$.

A mutation operation m, applied to a formula f, is said to be *shape preserving* if $\sigma(m(f)) = \sigma(f)$. The core shape of a block remains unchanged under application of mutation operators since all the mutation operators presented in this paper act only on formulas. Some mutation operations are not shape preserving since they might mutate references within formulas, thereby modifying a block's shadow.

For mutation testing, we can either apply the mutation operators to representative formulas within cp-similar regions or to all the formulas in the entire spreadsheet. While running the test cases to kill the mutants, we can either run the test cases on the formulas within region-representative blocks in isolation or we can run test sheets on the entire spreadsheet. These options give rise to the four strategies shown in Table 4 which are discussed in detail in the following sections.

		Test Context	
		Isolated block	Complete spreadsheet
Mutation	Block	BI	BC
	Spreadsheet	SI	SC

Table 4: Strategies for testing regions in spreadsheets

6.1 BI & SI Strategies

In the BI strategy, we mutate formulas in a region-representative block and run test cases on them in isolation from the rest of the spreadsheet, whereas in the SI strategy we run the mutation operators on all the formulas in the entire spreadsheet and then run test cases on the mutated formulas in a region representative block in isolation. For example, under the BI strategy, we could pick F2 as the region representative formula and generate the mutants by applying the mutation operators to it. We could then run the test cases for F2, F3, and F4 on the mutants generated from F2 and see how many of the generated mutants are killed by the test suite. One could also imagine generating mutants for the entire sheet and then running the test cases for F2, F3, and F4 on the mutants generated from F2 (SI strategy). However, SI is actually wasteful since there is no advantage in running the mutation operators on the entire sheet when we are only testing the region-representative block. We have included SI in the discussion only for the sake of completeness. The calculated mutation adequacy score in both cases can be assumed to be applicable to all three cells F2, F3, and F4. As far as mutation testing of a region representative block is concerned, the effect of both BI and SI strategies is identical, and they yield the same results.

Even though these approaches lower the cost of mutation testing by requiring that only the region-representative formula be tested in isolation, they have one practical problem in the context of mutation testing. These approaches assume that the seeded faults in the cp-similar blocks are independent of one another. The non shape-preserving mutation operators violate this assumption. For example, the mutation operators that modify references might cause a reference in the region-representative block to point to a cell in another one of the cp-similar blocks, thereby altering the block's shadow. Test cases aimed at testing the original region-representative block independently from the rest of the spreadsheet cannot be run to detect the mutants that have a different shape. M1 and M2 discussed in Section 7.1 are examples of such mutants.

6.2 SC Strategy

In this approach we generate mutants from all the formula cells within the spreadsheet. We can then run all possible combinations of the test cases that have been specified for the cp-similar formulas. For example, since the rows 2, 3, and 4 are cp-similar, we can generate 3! (= 6) test sheets from the single user-defined test sheet shown in Figure 2 by permuting the rows in the cp-similar region. Following the notion of derived test cases discussed at the beginning of this section, the test sheets that are generated from the user-defined test sheet are called *derived test sheets*. The advantage of this approach is that at each step of mutation testing, we are running the entire spreadsheet with each of the derived test sheets while trying to kill the mutants. The downside is that this approach is computationally more expensive for the following reasons.

Even though the actual number of generated mutants depends on the content within the cells, it is obvious that their number increases with the number of cells. Therefore, generating mutants based on every cell of the spreadsheet might result in a very high number of mutants for large spreadsheets. The time taken to run test suites to kill the mutants would be proportional to the number of mutants.

Large cp-similar regions would result in many derived sheets. For example, n cp-similar rows would result in $n!$ test sheets, out of which all but one are derived test sheets. For a well-designed test sheet, we might not need all $n!$ test cases to kill all the mutants. Since we generate test sheets at any point during mutation testing "on demand" depending on whether or not any mutants are still alive, we might not need to generate and run all $n!$ test sheets in general. However, in the worst case scenario, we need all $n!$ test sheets to kill the mutants, and generating and running all the test sheets to kill the mutants would be computationally expensive.

6.3 BC Strategy

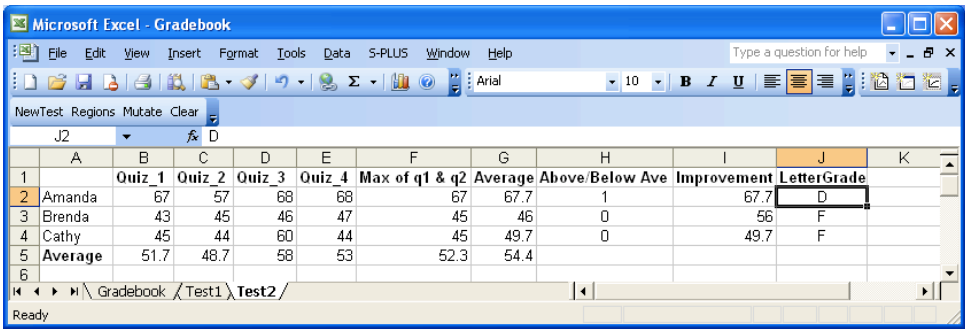
In this approach, we apply the mutation operators only to the formulas in the region-representative block for a cp-similar region. We then run the test sheets, both user specified and derived, on the full spreadsheet with the mutant versions of the region-representative block to try and kill the mutants. As with the SC strategy described above, in this approach, too, we might need to run $n!$ test sheets in the worst case scenario. To assess the savings in applying mutation operators to only the region-representative blocks, we looked at the analyses carried out on the EUSES spreadsheet corpus that have been reported in [27]. The corpus has 4498 spreadsheets collected from various sources. Out of the 1977 spreadsheets in the corpus that have formulas in them, 1797 have cp-similar regions. Among the sheets that have cp-similar regions, there are on average 5.2 regions per sheet, with an average of 13.1 regions in spreadsheets that had at least 1 region, a maximum of 414 regions in a spreadsheet, and 23845 regions in total in all the spreadsheets. No figures have been reported in [27] on the average size of the cp-similar regions, which are, by definition, greater than 1 cell in size. The average size of the largest regions in the spreadsheets has been reported as 63 cells. The size of the largest region is 23104 cells. Since cp-similar regions occur frequently in real world spreadsheets, using the BC strategy could potentially result in considerable time savings during mutation testing, both, because of the generation of the derived test cases and since the number of mutants generated is far less than is the case with SC.

As discussed in Section 6.1, the BI and SI strategies are not really conducive for mutation testing. In the next section we discuss how test cases can be developed using μTest to achieve higher mutation coverage under the SC and BC strategies. We also present results obtained from running the two strategies independently.

7 Evaluation of Test Suites through Mutation Testing

As discussed earlier, mutation testing has been traditionally used to evaluate the effectiveness of test suites. The main idea behind mutation testing is that a test suite that kills most of the mutants would be very effective at finding real faults as well.

7.1 Mutation Testing Using SC for Regions


Microsoft Excel - Gradebook
File Edit View Insert Format Tools Data S-PLUS Window Help
Type a question for help
Arial
10
NewTest Regions Mutate Clear
J2
D
Gradebook
Test1
Test2
Ready

	A	B	C	D	E	F	G	H	I	J	K
1		Quiz_1	Quiz_2	Quiz_3	Quiz_4	Max of q1 & q2	Average	Above/Below Ave	Improvement	LetterGrade	
2	Amanda	67	57	68	68	67	67.7	1	67.7	D	
3	Brenda	43	45	46	47	45	46	0	56	F	
4	Cathy	45	44	60	44	45	49.7	0	49.7	F	
5	Average	51.7	48.7	58	53	52.3	54.4				
6											

Figure 4: Second test sheet.

We initiate mutation testing by clicking the menu button labeled “Mutate”. Applied to the test sheet Test1 shown in Figure 2, the system reports that the test sheet killed 720 out of a total of 982 mutants that were generated (that is, Test1 is 73.34% mutation adequate).

To guide our efforts in developing additional test cases, we can use any of the standard coverage criteria like condition coverage, decision coverage, or du-adequacy[6]. The formula in I2 is IF(AND(B2<C2,C2<D2,D2<E2,H2<1),G2+10,G2). The test cases in Test1 cause the if-statement to evaluate to False. In order to satisfy du-adequacy, we need at least one test case that would cause the if-statement to evaluate to True. The formula in J2 is IF(I2>89,"A",IF(I2>79,"B",IF(I2>69,"C",IF(I2>59,"D","F")))). For this formula and the cp-similar ones in J3 and J4, we see that Test1 has test cases that cause the formula to evaluate to “B”, “C”, and “A”. Again, in order to satisfy du-adequacy, we need test cases that would cause the formula to evaluate to “D” and “F” as well. That is, we need a test case in which I2 evaluates to a number greater than or equal to 59 but less than 69, and another test case in which I2 evaluates to a number less than 59. We are now in a position to design the new test sheet (Test2) shown in Figure 4 that meets the requirements we have come up with. In the current version of μTest, the user has to develop test cases

[6]Du-adequacy is the criterion used by the WYSIWYT testing methodology [51] and involves testing every definition-use pair in the data flow.

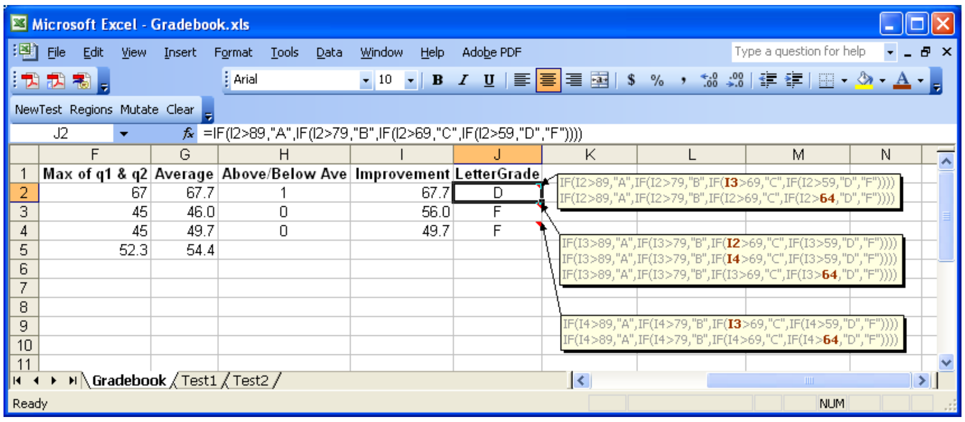
	Tests						
	1	1 (R)	2	2 (R)	1 & 2	1& 2 (R)	1,2 & 3 (R)
Total mutants	982	982	982	982	982	982	982
Test sheets	1	6	1	6	2	12	18
Mutants killed	720	883	721	895	842	975	979
Mutants alive	262	99	261	87	140	7	3
Coverage	73.34%	89.91%	73.42%	91.14%	85.74%	99.29%	99.69%
Time (sec)	1	4	1	5	2	5	6

Table 5: Summary of mutation test results with SC for regions

manually. To lower the effort involved in mutation testing, we plan to integrate automatic test case generation into future versions of μTest.

The statistics for the different test sheets are shown in Table 5. In the table, the column "Test 1 (R)" refers to the case in which test sheet Test1 was run with the regions enabled. The system generated 6 test sheets from the given single test sheet since it was specified that rows 2 through 4 are cp-similar. The 6 test sheets together achieved 89.91% mutation adequacy. The results from using test sheet Test2 with regions enabled have also been reported (91.14% mutation adequacy). Running test sheets 1 and 2 together resulted in 842 mutants being killed. Running test sheets Test1 and Test2 together with regions enabled leads to a total of 12 test sheets. We see that this set of test sheets achieves 99.29% mutation coverage—only 7 out of the original set of 982 mutants survived this test suite.

Mutants that have not been killed by the test suite are reported to the user by marking the corresponding cells. The user can then click the cells and see the mutants that have survived in that cell. Figure 5 shows the 7 mutants that survived the test suite. The mutated part of the formula is highlighted. We see that the 7 mutants can be grouped into two classes. We take a closer look at the three mutants in J3 (referring to them as M1, M2, and M3) to see why they survived the test suite and what the implications are since the test suite consisting of Test1 and Test2 is du-adequate.



Microsoft Excel - Gradebook.xls
File Edit View Insert Format Tools Data Window Help Adobe PDF
Type a question for help
Arial
10
NewTest Regions Mutate Clear
Gradebook
Test1
Test2
Ready
NUM

J2 =IF(I2>89,"A",IF(I2>79,"B",IF(I2>69,"C",IF(I2>59,"D","F"))))

	F	G	H	I	J
1	Max of q1 & q2	Average	Above/Below Ave	Improvement	LetterGrade
2	67	67.7	1	67.7	D
3	45	46.0	0	56.0	F
4	45	49.7	0	49.7	F
5	52.3	54.4			

IF(I2>89,"A",IF(I2>79,"B",IF(**I3**>69,"C",IF(I2>59,"D","F"))))
IF(I2>89,"A",IF(I2>79,"B",IF(I2>69,"C",IF(I2>**64**,"D","F"))))

IF(I3>89,"A",IF(I3>79,"B",IF(**I2**>69,"C",IF(I3>59,"D","F"))))
IF(I3>89,"A",IF(I3>79,"B",IF(**I4**>69,"C",IF(I3>59,"D","F"))))
IF(I3>89,"A",IF(I3>79,"B",IF(I3>69,"C",IF(I3>**64**,"D","F"))))

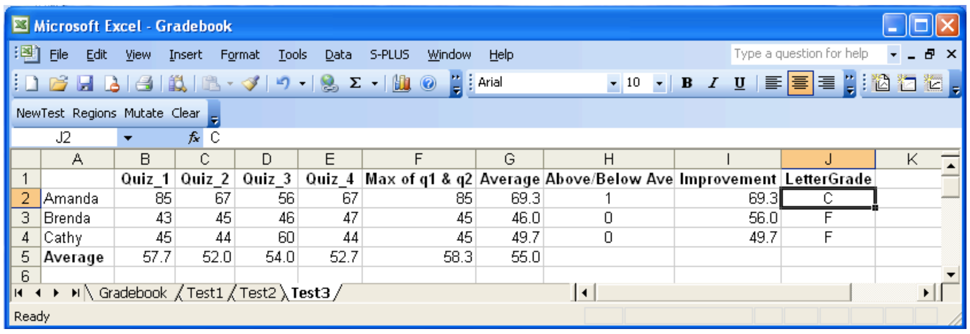
IF(I4>89,"A",IF(I4>79,"B",IF(**I3**>69,"C",IF(I4>59,"D","F"))))
IF(I4>89,"A",IF(I4>79,"B",IF(I4>69,"C",IF(I4>**64**,"D","F"))))

Figure 5: Mutants that have survived tests 1 & 2 with regions enabled.

M1: IF(I3>89,"A",IF(I3>79,"B",IF(**I2**>69,"C",IF(I3>59,"D","F"))))
Here a reference to I3 has been replaced with a reference to I2. In this case, we have

$\sigma(\text{M1}) \neq \sigma(S(\text{J3}))$, that is, the mutation is not shape preserving. This mutant reflects a mechanical error that could have arisen if the user had selected the wrong cell while constructing the formula. The test case that tests the mutated condition is the one in row 2 of the first test sheet (I2 contains 80.0, see Figure 2), which causes the resulting output in J3 to be "C". Even in the test cases that are generated when regions are enabled, when I3 evaluates to 69.3 (the case for which J3 evaluates to "C"), the possible values for the original reference I2 are 80.0 (from the test case in row 1) and 90.0 (from the test case in row 3). Both these test cases result in J3 evaluating to "C" as well since they cause the condition I2>69 to evaluate to true. Therefore this mutant survives the test suite even when regions are enabled.

In order to kill the mutant M1, we need a test case with an expected output value of "C" for J3, and I2 evaluating to a value less than or equal to 69. We could achieve this under the current scenario if we combined all the tests from Test1 and Test2 and ran all possible combinations. This approach would not be very efficient since we would have a total of 6! (= 720) test sheets. Another solution would be to combine the third row from Test1 with the third and fourth rows from Test2 and create the test sheet Test3 (shown in Figure 6) based on our observations regarding the surviving mutants. The results from


Microsoft Excel - Gradebook

J2 = C

	A	B	C	D	E	F	G	H	I	J	K
1		Quiz_1	Quiz_2	Quiz_3	Quiz_4	Max of q1 & q2	Average	Above/Below Ave	Improvement	LetterGrade	
2	Amanda	85	67	56	67	85	69.3	1	69.3	C	
3	Brenda	43	45	46	47	45	46.0	0	56.0	F	
4	Cathy	45	44	60	44	45	49.7	0	49.7	F	
5	Average	57.7	52.0	54.0	52.7	58.3	55.0				
6											

File Edit View Insert Format Tools Data S-PLUS Window Help
Type a question for help
Arial
10
NewTest Regions Mutate Clear
Gradebook
Test1
Test2
Test3
Ready

Figure 6: Third test sheet.

running the test sheets Test1, Test2, and Test3 with regions enabled is also shown in Table 5. We see that including Test3 resulted in a total of 18 test sheets being run, and brought the mutation coverage to 99.69%. The three mutants that survive are M3 and similar mutants in rows 2 and 4.

M2: IF(I3>89,"A",IF(I3>79,"B",IF(**I4**>69,"C",IF(I3>59,"D","F"))))
This mutant is similar to the one discussed above except that the reference to I3 has been replaced with a reference to I4 in this case. Test3 kills this mutant as well.

M3: IF(I3>89,"A",IF(I3>79,"B",IF(I3>69,"C",IF(I3>**64**,"D","F"))))
In this mutant the constant 59 has been replaced with 64. This mutation reflects a scenario in which the developer of the spreadsheet might have made an error or is not aware of the specifications. The mutant could not be killed because there is no test case that assigns I3 a value greater than 59 but less than 64. M3 also illustrates how mutation on constants can give rise to a high number of mutants since the constant 59 can be replaced by any number greater than 59 but less than 69. Each of these mutants can be killed by a single test case in which I3 evaluates to a number greater than 59 by a very small fraction.

	Tests						
	1	1 (R)	2	2 (R)	1 & 2	1 & 2 (R)	1, 2 & 3 (R)
Total mutants	181	181	181	181	181	181	181
Test sheets	1	6	1	6	2	12	18
Mutants killed	99	148	100	151	141	178	180
Mutants alive	82	33	81	30	40	3	1
Coverage	54.7%	81.77%	55.25%	83.43%	77.9%	98.34%	99.45%
Time (sec)	1	1	1	1	1	1	1

Table 6: Summary of mutation test results with BC for regions

7.2 Mutation Testing Using BC for Regions

In this approach, we run the mutation operators on the region-representative block (for example, row 3) of the spreadsheet shown in Figure 1. The results of mutation tests are shown in Table 6. Similar to the case when using the SC approach for regions, the only mutant that survives when we run test sheets Test1, Test2, and Test3 with regions enabled is M3. The key point to note in this approach is the savings in execution time. With this approach, each configuration of test sheets takes only about a second to run.

We see from the above discussion that both SC and BC approaches would lead to the design of the same (or at least similar) test suites. Moreover, both approaches yield similar mutation adequacy numbers for the different test sheets. The big difference between the approaches is in the execution time—with regions enabled, SC takes up to 6 times the time taken by BC for completion of mutation testing. For larger spreadsheets with big cp-simlar regions, simply the savings in execution time would constrain us to using the BC strategy.

8 Further Applications of Mutation Operators

In addition to evaluating test suites for spreadsheets, the suite of mutation operators described in Section 4 can be used for evaluating the effectiveness of error-detection mechanisms in spreadsheets. In Section 8.1, we first present two examples of how this can be done. In Section 8.2, we describe how the mutation operators can be used to design empirical studies using spreadsheets.

8.1 Evaluating the Accuracy of Spreadsheet Tools

8.1.1 Evaluation of GoalDebug

As described in the introduction, GoalDebug is a debugger for spreadsheets that allows the user to mark the output from a cell as incorrect and specify the expected output either as a single value, or more generally, as a constraint [2]. The system then propagates the user-specified constraint backward and generates change suggestions, any one of which, when applied, would result in the expected result being computed in the marked cell. In general, there are many change suggestions possible, but some are more reasonable than others. For example, changing a reference in a formula to another reference is more reasonable than replacing the entire formula with the expected output value. To present the user with only a few, likely suggestions, we have developed several heuristics to rank the generated change suggestions. For example, while ranking reference changes, we assume mechanical error by the user (accidentally selecting an incorrect cell in the immediate neighborhood of the intended cell) and rank the changes based on the distance from the original reference. Assume that solving the user-specified constraint generates references E5, G5, and B5 as possible changes to F5. Then changing a reference to F5 to a reference to E5 or G5 (a distance of one cell in both cases) would be ranked higher than a change to B5 (distance of 4 cells from F5). Although the heuristics seem to perform quite well in practice, we have not yet performed a systematic study to demonstrate their effectiveness. This is where the mutation operators come into play.

To evaluate the effectiveness of our ranking heuristics, we can first use the mutation operators to generate a set of mutants of a spreadsheet. For each test case that killed a mutant, we run GoalDebug with the expected output from the cell(s) concerned and examine the generated suggestions to evaluate how high each correct change suggestion[7] would be ranked.

8.1.2 Evaluation of UFix

The UCheck system described in [1] identifies inconsistencies in spreadsheet formulas based on inferred unit information. Whenever a unit that has been inferred for a spreadsheet formula cannot be transformed into a normal form, a unit error has been found and the formula is marked as potentially incorrect. An extension to UCheck currently under development is the UFix system [3] that examines unit errors and generates suggestions that would allow the error to be corrected. To evaluate the accuracy of unit checking and the change suggestions, we can use the suite of mutation operators to generate mutants of a spreadsheet. We can then run UCheck to see what classes of real-world errors can be detected through unit checking. We can then run UFix to see if the change suggestions generated by the system would reverse the mutation.

8.2 Mutation Operators for Seeding Errors

Oftentimes, spreadsheets seeded with errors are used in empirical studies to evaluate the effectiveness or usability of error-prevention or -detection mechanisms [52, 48]. The seeding of errors is usually done manually on the basis of accepted classifications [8, 44]. The mutation operators we have developed take the classification schemes into account. Therefore, an experimenter who wants to seed spreadsheets with errors to carry out a study can use operators from our suite. Moreover, depending on the goals of the empirical study, the designer of the experiment could use some operators and not others. This approach would make the study less biased as opposed to the scenario in which the experimenter seeds the faults manually.

9 Conclusions and Future Work

In this paper, we have presented a suite of mutation operators for spreadsheets, and a system that allows users to create test suites for spreadsheets and carry out mutation testing using their test suites. We have also discussed three strategies that allow us to exploit knowledge about cp-similar regions within the spreadsheets, both in generating test sheets and also in minimizing time taken to run mutation tests. In addition to helping

[7]A correct change suggestion is one that would reverse the mutation being considered.

with mutation testing of spreadsheets, the mutation operators can be used for evaluating spreadsheet tools and also for seeding spreadsheets with errors for empirical studies.

The list of spreadsheet mutation operators proposed in this paper is by no means complete. We suspect the list will evolve as we try to mirror more errors from real-world spreadsheets. A more complete list of mutation operators would allow a practitioner to adapt the operators to other application more easily.

After a spreadsheet has undergone changes, some test cases might become out of date and need to be removed from the test suites, some test cases might need to be refined, and some new ones might need to be included to test new formulas in the spreadsheet. The current version of μTest does not have any mechanism to carry out the necessary regression testing analyses to inform the user of the impact of changes on the level of testedness of the spreadsheet. Since the WYSIWYT implementation already supports regression analyses, it would be beneficial to integrate the two approaches once WYSIWYT has been implemented in Excel.

Since the high cost of testing and debugging could discourage users, support for automatic test case generation is very important to help end users use this tool. We plan to explore mutation-based approaches [37] and see how they can be integrated with the existing "Help Me Test" (HMT) [26] implementation that is part of WYSIWYT.

References

[1] R. Abraham and M. Erwig. Header and Unit Inference for Spreadsheets Through Spatial Analyses. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 165–172, 2004.

[2] R. Abraham and M. Erwig. Goal-Directed Debugging of Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 37–44, 2005.

[3] R. Abraham and M. Erwig. How to Communicate Unit Error Messages in Spreadsheets. In *1st Workshop on End-User Software Engineering*, pages 52–56, 2005.

[4] R. Abraham and M. Erwig. Inferring Templates from Spreadsheets. In *28th IEEE Int. Conf. on Software Engineering*, pages 182–191, 2006.

[5] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual Specifications of Correct Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 189–196, 2005.

[6] A. T. Acree. *On Mutation.* PhD thesis, Georgia Institute of Technology, Atlanta GA, 1980.

[7] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A Type System for Statically Detecting Spreadsheet Errors. In *18th IEEE Int. Conf. on Automated Software Engineering*, pages 174–183, 2003.

[8] C. Allwood. Error Detection Processes in Statistical Problem Solving. *Cognitive Science*, 8(4):413–437, 1984.

[9] J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation An Appropriate Tool For Testing Experiments? In *27th Int. Conf. on Software engineering*, pages 402–411, 2005.

[10] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the Unit Correctness of Spreadsheet Programs. In *26th IEEE Int. Conf. on Software Engineering*, pages 439–448, 2004.

[11] T. A. Budd. PhD thesis, Yale University, New Haven CT, 1980.

[12] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. In *ACM Symp. on Principles of Programming Languages*, pages 220–233, 1980.

[13] M. M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming*, 11:155–206, March 2001.

[14] M. M. Burnett, C. Cook, J. Summet, G. Rothermel, and C. Wallace. End-User Software Engineering with Assertions. In *25th IEEE Int. Conf. on Software Engineering*, pages 93–103, 2003.

[15] M. M. Burnett, A. Sheretov, B. Ren, and G. Rothermel. Testing Homogeneous Spreadsheet Grids with the "What You See Is What You Test" Methodology. *IEEE Transactions on Software Engineering*, 29(6):576–594, 2002.

[16] M. J. Coblenz, A. J. Ko, and B. A. Myers. Using Objects of Measurement to Detect Spreadsheet Errors. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 314–316, 2005.

[17] M. Delamaro and J. Maldonado. Proteum–A Tool for the Assessment of Test Adequacy for C Programs. In *Proc. of the Conf. on Performability in Computing Systems*, pages 79–95, 1996.

[18] R. A. DeMillo, D. S. Guindi, K. N. King, and W. M. McCracken. An Extended Overview of the Mothra Software Testing Environment. In *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, 1988.

[19] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17:900–910, 1991.

[20] S. Ditlea. Spreadsheets Can be Hazardous to Your Health. *Personal Computing*, 11(1):60–69, 1987.

[21] G. Engels and M. Erwig. ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications. In *20th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 124–133, 2005.

[22] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic Generation and Maintenance of Correct Spreadsheets. In *27th IEEE Int. Conf. on Software Engineering*, pages 136–145, 2005.

[23] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Gencel — A Program Generator for Correct Spreadsheets. *Journal of Functional Programming*, 16(3):293–325, May 2006.

[24] M. Erwig and M. M. Burnett. Adding Apples and Oranges. In *4th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 2257, pages 173–191, 2002.

[25] EUSES. End Users Shaping Effective Software. `http://EUSESconsortium.org`.

[26] M. Fisher II, M. Cao, G. Rothermel, C. Cook, and M. M. Burnett. Automated Test Case Generation for Spreadsheets. In *24th IEEE Int. Conf. on Software Engineering*, pages 141–151, 2002.

[27] M. Fisher II and G. Rothermel. The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanism. In *1st Workshop on End-User Software Engineering*, pages 47–51, 2005.

[28] K. Godfrey. Computing Error at Fidelity's Magellan Fund. In *Forum on Risks to the Public in Computers and Related Systems*, January 1995.

[29] R. Hierons, M. Harman, and S. Danicic. Using Program Slicing to Assist in the Detection of Equivalent Mutants. *Software Testing, Verification and Reliability*, 9:233–262, 1999.

[30] K. S. How Tai Wah. A Theoretical Study of Fault Coupling. *Software Testing, Verification and Reliability*, 10(1):3–45, 2000.

[31] K. S. How Tai Wah. An Analysis of the Coupling Effect I: Single Test Data. *Science of Computer Programming*, 48(2-3):119–161, 2003.

[32] W. E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Trans. on Software Engineering*, 8:371–379, 1992.

[33] T. Isakowitz, S. Schocken, and H. C. Lucas, Jr. Toward a Logical/Physical Theory of Spreadsheet Modelling. *ACM Transactions on Information Systems*, 13(1):1–37, 1995.

[34] Y. Ma, A. J. Offutt, and Y. Kwon. MuJava: An Automated Class Mutation System. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[35] A. P. Mathur. Performance, Effectiveness, and Reliability Issues in Software Testing. In *Fifteenth Annual Int. Computer Software and Applications Conference*, pages 604–605, 1991.

[36] R. Mittermeir and M. Clermont. Finding High-Level Structures in Spreadsheet Programs. In *9th Working Conference on Reverse Engineering*, pages 221–232, 2002.

[37] A. J. Offutt. An Integrated Automatic Test Data Generation System. *Journal of Systems Integration*, 1(3):391–409, 1991.

[38] A. J. Offutt. Investigations Of The Software Testing Coupling Effect. *ACM Trans. on Software Engineering and Methodology*, 1(1):5–20, 1992.

[39] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination Of Sufficient Mutant Operators. *ACM Trans. on Software Engineering and Methodology*, 5(2):99–118, 1996.

[40] A. J. Offutt and J. Pan. Detecting Equivalent Mutants and the Feasible Path Problem. *Software Testing, Verification, and Reliability*, 7:165–192, 1997.

[41] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An Experimental Evaluation of Data Flow and Mutation Testing. *Software Practice and Experience*, 26(2):165–176, 1996.

[42] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the Orthogonal. In *Proc. Mutation*, pages 45–55, 2000.

[43] R. R. Panko. Applying Code Inspection to Spreadsheet Testing. *Journal of Management Information Systems*, 16(2):159–176, 1999.

[44] R. R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. In *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000.

[45] R. R. Panko and R. P. Halverson, Jr. Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks. In *29th Hawaii Int. Conf. on System Sciences*, 1996.

[46] S. L. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, Cambridge, UK, 2003.

[47] S. G. Powell and K. R. Baker. *The Art of Modeling with Spreadsheets: Management Science, Spreadsheet Engineering, and Modeling Craft.* Wiley, 2004.

[48] S. Prabhakarao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett. Strategies and Behaviors of End-User Programmers with Interactive Fault Localization. In *IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, pages 203–210, 2003.

[49] K. Rajalingham, D. Chadwick, B. Knight, and D. Edwards. Quality Control in Spreadsheets: A Software Engineering-Based Approach to Spreadsheet Development. In *33rd Hawaii Int. Conf. on System Sciences*, pages 1–9, 2000.

[50] B. Ronen, M. A. Palley, and H. C. Lucas, Jr. Spreadsheet Analysis and Design. *Communications of the ACM*, 32(1):84–93, 1989.

[51] G. Rothermel, M. M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A Methodology for Testing Spreadsheets. *ACM Transactions on Software Engineering and Methodology*, pages 110–147, 2001.

[52] J. Ruthruff, E. Creswick, M. M. Burnett, C. Cook, S. Prabhakararao, M. Fisher II, and M. Main. End-User Software Visualizations for Fault Localization. In *ACM Symp. on Software Visualization*, pages 123–132, 2003.

[53] J. Sajaniemi. Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization. *Journal of Visual Languages and Computing*, 11:49–82, 2000.

[54] C. Scaffidi, M. Shaw, and B. Myers. Estimating the Numbers of End Users and End User Programmers. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.

[55] A. Scott. Shurgard Stock Dives After Auditor Quits Over Company's Accounting. *The Seattle Times*, November 2003.

[56] R. E. Smith. University of Toledo loses $2.4M in projected revenue. *Toledo Blade*, May 2004.

[57] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation Analysis Using Program Schemata. In *Proc. of the Int. Symp. on Software Testing and Analysis*, pages 139–148, 1993.

[58] E. J. Weyuker, S. N. Weiss, and D. Hamlet. Comparison of Program Testing Strategies. In *Fourth Symposium on Software Testing, Analysis and Verification*, pages 154–164, 1991.

[59] W. E. Wong, J. C. Maldonado, M. E. Delamaro, and S. Souza. Use of Proteum to Accelerate Mutation testing in C Programs. In *ISSAT International Conference on Reliability and Quality in Design*, pages 254–258, 1997.

[60] A. G. Yoder and D. L. Cohn. Real Spreadsheets for Real Programmers. In *Int. Conf. on Computer Languages*, pages 20–30, 1994.