Mutation Operators for Spreadsheets

Robin Abraham and Martin Erwig, *Member*, *IEEE*

Abstract—Based on 1) research into mutation testing for general-purpose programming languages and 2) spreadsheet errors that have been reported in the literature, we have developed a suite of mutation operators for spreadsheets. We present an evaluation of the mutation adequacy of definition-use adequate test suites generated by a constraint-based automatic test-case generation system we have developed in previous work. The results of the evaluation suggest additional constraints that can be incorporated into the system to target mutation adequacy. In addition to being useful in mutation testing of spreadsheets, the operators can be used in the evaluation of error-detection tools and also for seeding spreadsheets with errors for empirical studies. We describe two case studies where the suite of mutation operators helped us carry out such empirical evaluations. The main contribution of this paper is a suite of mutation operators for spreadsheets that can be used for performing empirical evaluations of spreadsheet tools to indicate ways in which the tools can be improved.

Index Terms—End-user software engineering, spreadsheets, end-user programming.

1 Introduction

Spreadsheets are among the most widely used programming systems [1]. Studies have shown that there is a high incidence of errors in spreadsheets [2], up to 90 percent in some cases [3]. Some of these errors have high impact [4] leading to companies and institutions losing millions of dollars [5]. 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. It has also been observed that users have a mostly unjustified high level of confidence about the correctness of their spreadsheets [6], [7].

To reduce the incidence of errors in spreadsheets, research into spreadsheets has focused on the following areas:

1. Recommendations for better spreadsheet design [3], [8], [9], [10], [11].
2. Auditing spreadsheets to detect and remove errors [12], [13], [14].
3. Automatic consistency checking [15], [16], [17], [18], [19], [20].
4. Error prevention techniques [21], [22].
5. Testing [23], [24], [25].

Traditional software engineering research has made considerable progress in the area of testing. Part of the EUSES [26] research collaboration's goal is to bring these benefits to the realm of spreadsheets.

For general-purpose programming languages, the idea of mutation testing was proposed in [27], [28]. In mutation testing [29], [30], [31], [32], [33], faults are inserted into the program that is being tested. Each seeded fault generates a new program, a mutant,[1] that is slightly different from the original. The objective is to design test cases that cause the original program to compute outputs that are different from those computed by the mutants. In such cases, the mutants are said to be *killed* by the test case. Since the seeded faults reflect the result of errors made by software developers, a test suite that kills mutants is assumed to be effective at detecting faults in the program that is being tested [34]. In some cases, the mutant might produce the same output as the original program, given the same input. Such mutants are said to be *equivalent* to the original program and cannot be distinguished from the original program through testing. The goal in mutation testing is to find a test suite that is mutation adequate, which means it detects all the nonequivalent mutants.

Various comparisons of the effectiveness of *test adequacy criteria*[2] are available to guide the choice of strategies for testing programs [35], [36], [37]. Two aspects guide the choice of one test adequacy criterion 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 faults 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 trade-off 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

- *R. Abraham is with Microsoft Corp., 3141, 3028 147th Place N.E., Redmond, WA 98052. E-mail: Robin.Abraham@microsoft.com.*
- *M. Erwig is with the Department of Electrical and Computer Engineering, 3047, Kelly Engineering Center, Oregon State University, Corvallis, OR 97330. E-mail: erwig@eecs.oregonstate.edu.*

Manuscript received 14 Dec. 2006; revised 11 June 2007; accepted 17 June 2008; published online 22 Aug. 2008.
Recommended for acceptance by P. Frankl.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0279-1206.
Digital Object Identifier no. 10.1109/TSE.2008.73.

1. 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.

2. A test adequacy criterion is basically a set of rules that determine if the testing is sufficient for a given program and specification.

0098-5589/09/$25.00 © 2009 IEEE Published by the IEEE Computer Society

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 users testing their spreadsheets.

The only existing framework for testing spreadsheets is "What You See Is What You Test" (WYSIWYT) [23], which has been implemented in the Forms/3 environment [38]. WYSIWYT allows users to create test cases and uses *definition-use (du) adequacy* as the criterion for measuring the level of testedness of the spreadsheet. The idea behind the du coverage criterion is to test for each definition of a variable (or a cell in the case of spreadsheets) all of its uses.

Empirical studies comparing data-flow and mutation testing have shown that mutation-adequate tests detect more faults [39]. In this paper, we propose a suite of mutation operators for spreadsheets and demonstrate their use in mutation testing of spreadsheets and in the evaluation of spreadsheet tools. In the empirical evaluation we carried out (described in Section 6), we saw some faults that would be discovered by a mutation-adequate test suite that were not discovered by the automatically generated test suites that satisfy the du-adequacy criterion.

In addition, the mutation operators also allow us to assess the effectiveness of automatic tools aimed at the detection and correction of errors in spreadsheets. Oftentimes, spreadsheets seeded with errors are used in empirical studies to evaluate the effectiveness or usability of error-prevention or -detection mechanisms [40], [41]. The seeding of errors is usually done manually on the basis of accepted classifications [7], [42]. However, there is no body of research documenting the kinds of spreadsheet errors and their frequency of occurrence in real-world spreadsheets. Systematic work still needs to be done in this area [43]. 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.

We describe related work in Section 2. We give a brief overview of mutation testing in Section 3, followed by a description of the formal notation used in the later sections of this paper in Section 4. The suite of mutation operators is described in Section 5. We describe evaluations using the mutation operators of the following spreadsheet tools:

1. Automatic test-case generator AutoTest [44] in Section 6.
2. Spreadsheet debugger GoalDebug [45], [46] in Section 7.
3. Combined reasoning [47] of the automatic consistency checker UCheck [48] and the testing framework WYSIWYT [23] in the online supplement Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2008.73.

Finally, we present conclusions and plans for future work in Section 8.

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) [49], Proteum (for C programs) [50], and μJava (for Java programs) [51]. Mutation testing has also been used and shown to be effective in other areas like aspect-oriented programming [52], Web services and Web applications [53], [54], and database queries [55]. At a more abstract level, in [56], Offutt et al. describe how mutation analysis can be viewed as a way to modify any software artifact based on its syntactic description.

As pointed out in [57], [58], 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. Some researchers have focused 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) [60]. Since there is no explicit compilation step in the case of spreadsheets, this cost is not much of a cause for concern for our system.
- 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 a 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 [61]. 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 [62], 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 using the entire set of 22 operators [63]. 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 nonequivalent mutants. These techniques have been shown to be only marginally less effective at fault detection when compared to the complete set of mutants [31], [32]. 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 focused on algorithms for automatic test generation that would result in test suites that are close to being mutation adequate [64], [65]. Support for generating test cases automatically [24], [44] would also lower the cost of mutation testing in spreadsheets.
- Some of the generated mutants might be equivalent to the original program, that is, they produce the same outputs as the original program for all 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 percent of equivalent mutants in general-purpose programming languages [66], [67].

One concern regarding the effectiveness of mutation adequacy as a criterion for the development of effective test suites is: How well do the mutants reflect faults that occur in real-world programs? The applicability of mutation to the evaluation of testing has been explored in [34], [68] and the authors have shown that generated mutants mirror real faults. However, mutants might be different from and easier to detect than 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 focused their efforts on guidelines for designing better spreadsheets so errors can be avoided to some degree [3], [8], [9], [10], [11]. Such techniques are difficult to enforce and involve costs of training the user. While this approach might be feasible within companies, it is not practical in the context of users working in isolation.

The benefits of the WYSIWYT approach (and associated systems) have been demonstrated empirically [23], [40], [41]. In WYSIWYT, users provide input values to the system and then mark the output cells with a $\surd$ if the output is correct, or a x 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. The testedness is reported to the user through a progress bar, that ranges from 0 percent to 100 percent 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).

In previous work, we have looked at ways to prevent errors in spreadsheets by automatically enforcing specifications. This approach, implemented in the Gencel system [21], [69], captures a spreadsheet and all its possible evolutions in a template developed in the Visual Template Specification Language (ViTSL) [70]. In the case of spreadsheets generated using this approach, the user does not ever have to edit the spreadsheet formulas since they are automatically generated by the system based on the template. However, the user would have to ensure the correctness of the input values. In the context of companies or other large organizations, the templates can be created (or audited to ensure correctness) 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 in the area of spreadsheet errors 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 [12], [13], [14]. However, such approaches cannot give any guarantees about the correctness of the spreadsheet once the inspection has been carried out. Empirical studies have shown that individual code inspection only catches 63 percent of the errors whereas group code inspection catches 83 percent of the errors [12]. 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 [15], [16], [17], [18], [19]. We have developed a system, called UCheck, that automatically infers the labels within the spreadsheet [20] and uses this information to carry out consistency checking [48], thereby requiring minimal effort from the user.

In previous work, we have developed a constraint-based automatic test-case generator called AutoTest [44], which can generate test cases to exercise all feasible du pairs. Testing might cause the program to exhibit failures. Even so, the user still needs to identify the faults that led to the failures and come up with program changes to correct the faults. To help users debug their spreadsheets, we have developed an approach that allows users to specify the expected computed output of cells. The system, named GoalDebug, then generates change suggestions, any one of which when applied would result in the desired output in the cell [45], [46].

The distinction between different types of spreadsheet errors is important since they require different types of error detection and prevention techniques. Different classifications of spreadsheet errors have been proposed [3], [6], [71], [72]. Panko and Halverson classified quantitative errors in terms of three main types [6]:

1. *Mechanical errors.* These errors are simple slips that may arise due to carelessness, mental overload, or distractions. Examples include mistyping a number or a reference, pointing at a wrong cell address, or selecting an incorrect range of values or cells.

2. *Omission errors.* These errors arise from the programmer leaving something out of the model by accident.
3. *Logic errors.* These errors are caused when the programmer chooses an incorrect algorithm to solve the problem. Such errors typically manifest themselves as incorrect formulas. These are the hardest to detect since they oftentimes require domain-specific knowledge.

Teo and Tan have classified errors in spreadsheets as either *quantitative* errors or *qualitative* errors [72], [73]. Quantitative errors usually manifest themselves as incorrect results in the spreadsheet. Qualitative errors, on the other hand, take the form of poor spreadsheet design and format. Qualitative errors in a spreadsheet might not be visible right away, but they lead to quantitative errors in future spreadsheet instances. Researchers have studied the occurrence of qualitative and quantitative errors and have made recommendations on how to avoid them. Mason and Keane have suggested that organizations should have a "Model Administrator" (along the lines of Database Administrators for enterprise-level databases) to regulate spreadsheet models within the organization [74]. Williams has recommended the adoption of organization standards [75]. These standards would consist of recommendations for best practices that cover spreadsheet specification, documentation, maintenance, and security that employees should follow while creating spreadsheets. The use of automated spreadsheet audit software has been recommended by Simkin [76]. Teo and Tan have shown that spreadsheet errors are hard to detect during "what-if" analyses if the spreadsheets are not well designed [73]. Based on their studies, they have recommended that courses designed for teaching spreadsheets should focus more on spreadsheet design, targeting ease of maintenance and debugging, rather than demonstrating the features available in spreadsheet systems.

3 Mutation Testing

Mutation analysis, strictly speaking, is a way to measure the quality of test suites. This aspect might not be immediately obvious from the activities involved: generation of mutants and the development of test cases targeted at killing the generated mutants. 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, if the test cases in the test suite kill all mutants, then software is well tested by the test suite, assuming the mutants represent real faults [57]. The fault seeding is done by means of *mutation operators.* These mutation operators should be powerful enough to show how effective the test suite is. However, in practice, this goal requires an expressiveness of the mutation operators that makes them expensive in terms of running time. Therefore, achieving powerful mutation operations often come at the cost of a higher 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 measure 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.* That is, a dead mutant is one that produces an output different from that of the original program on at least one test case. 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 of the *nonequivalent* 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 $\forall x \in D,\ P(x) = P_i(x)$. This equivalence is expressed as $P_i \equiv P$. Obviously, an equivalent mutant cannot be killed by testing. *Mutation adequacy* or *mutation score* is computed as

$$\frac{\text{number of killed mutants}}{\text{total number of nonequivalent mutants}} \times 100\%.$$

In choosing the mutation operators, we make the following assumptions laid out in [33]:

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 [77], [78], [79]. This result, which supports the coupling effect, is helpful for cases in which a single failure is the manifestation of multiple faults. Therefore, if we have tests that detect the isolated faults, we would be able to detect a compound fault (which is a combination of two or more simple faults detected by the test suite) with a high level of probability.

4 A Formal Model of Spreadsheet Programs

The 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.

We assume a set F that contains all possible formulas. Cell formulas ($f \in F$) are either plain values $v \in V$, references to other cells (given by addresses $a \in A$), or operations (ψ) applied to one or more argument formulas:

$$f \in F ::= v \mid a \mid \psi(f, \ldots, f).$$

Operations include binary operations, aggregations, and, in particular, a branching construct $\mathrm{IF}(f, f, f)$.

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; σ is defined as follows:

$$\begin{array}{lcl} \sigma(v) & = & \emptyset \\ \sigma(a) & = & \{a\} \\ \sigma(\psi(f_1, \ldots, f_k)) & = & \sigma(f_1) \cup \cdots \cup \sigma(f_k). \end{array}$$

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 (c) and cell addresses by $\forall (a, f) \in S,\ \sigma(a, f) = \sigma(f)$ and $\forall a \in dom(S), \sigma(a) = \sigma(S(a))$, that is, for a given spreadsheet S, $\sigma(a)$ gives the shape of the formula stored in cell a. Some of the cells in $\sigma(c)$ might themselves contain formulas. We define a related function $\sigma^*_S : S \times F \to 2^A$ that transitively chases references to determine all the input cells (that contain only values and not formulas) for the formula. The definition of σ^*_S is identical to that of σ, except for the following case:

$$\sigma^*_S(a) = \begin{cases} \{a\} & \text{if } S(a) \in V \\ \sigma^*_S(S(a)) & \text{otherwise.} \end{cases}$$

Like σ, σ^*_S can be extended to work on cells and addresses. The cells addressed by $\sigma^*_S(c)$ 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 (nontrivial) formulas, that is, $P_S = \{(a, f) \in S \mid \sigma(f) \neq \emptyset\}$. 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 replaced by a constant without changing the meaning of the spreadsheet. Correspondingly, the *input* of a spreadsheet S is given by all of its cells containing values (and locally evaluable formulas), that is, $D_S = \{(a, f) \in S \mid \sigma(f) = \emptyset\}$. Note that, with these two definitions, we have $S = P_S \cup D_S$ and $P_S \cap D_S = \emptyset$.

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 inputs, that is, $dom(I) = \sigma^*_S(f)$. In other words, the input values are given by data cells ($(a, f) \in D_S$) 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 $c = (a, f)$ 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 produces an output different from f for at least one test input in T. Likewise, a spreadsheet mutant S_i is *killed* by a test suite $\mathcal{T}_S$ if S_i produces an output different from that produced by S for at least one test input in $\mathcal{T}_S$.

5 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 [7], [42]. From this point of view, the operators themselves are a contribution to research on categories of errors in spreadsheets [3], [43], [80]. To ensure that the operators meet the second goal, we have included operators that have been developed for general-purpose programming languages.

A "standard" set of 22 mutation operators for Fortran has been proposed in [63]. A subset of these that are applicable to spreadsheets is shown in Table 1. As mentioned in Section 2, it has been shown empirically in [63] that test suites killing mutants generated by the five operators ABS, AOR, LCR, ROR, and UOI are almost 100 percent 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 adapted from the operators shown in Table 1. The original operators are mentioned in parenthesis. Other operators have been added that make sense only in the context of spreadsheets. For example, we have introduced a set of operators that mutate ranges, replace formulas in cells

TABLE 1
Subset of Mutation Operators for Fortran

Operator	Description
ABS	*ABS*olute value insertion
AOR	*A*rithmetic *O*perator *R*eplacement
CRP	*C*onstants *R*e*P*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	*SouR*ce *C*onstant replacement
SVR	*S*calar *V*ariable *R*eplacement
UOI	*U*nary *O*perator *I*nsertion

with constants, and change functions used in formulas (discussed in more detail below).

We do not have a mechanism that allows 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. Cyclic references lead to nonterminating computations and are reported by spreadsheet systems like Excel. Therefore, there is no need to detect them through test cases. 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.[3] 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 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. A type checker would be able to enforce this property at compile time in a general-purpose programming language.

TABLE 2
Correspondence of Constructs in General-Purpose Programming Languages and in Spreadsheets

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

TABLE 3
Mutation Operators for Spreadsheets

Operator	Description
ABS	*ABS*olute value insertion
AOR	*A*rithmetic *O*perator *R*eplacement
CRP	*C*onstants *R*e*P*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

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. Along similar lines, a reference could be mutated to point to another cell whose computed output is the same as that of the original cell. However, this mutant would reflect logical, and not mechanical, error.

In the current implementation of the mutation operators, we impose the third constraint (and drop the second) discussed above since we are modeling mechanical errors in automatic evaluation of spreadsheet tools (see Sections 6, 7, and the online supplement Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2008.73). Since Excel does not carry out type checking of formulas, we do not enforce the first constraint during mutation. As we have shown in [81], integrating a type checker within Excel would help prevent type errors and thereby lower the cost of testing.

We discuss below the mutation operators we have included that are unique to the domain of spreadsheets. For an RFR mutation, the candidates are the eight spatial neighbors of the cell originally being referenced and other cells that have the same output as the cell originally being referenced.

Range mutation. Aggregation formulas in spreadsheets typically operate over a range of references. Ranges might be *contiguous* or *noncontiguous*. 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 noncontiguous range that includes the references A2, A5, A8, and A11. We employ the following novel 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. *Noncontiguous Range Shrinking (NRS).* This operator removes any reference from a noncontiguous range.
3. *Contiguous Range Expansion (CRE).* This operator expands a contiguous range by altering the reference at its beginning or end.
4. *Noncontiguous Range Expansion (NRE).* This operator introduces an extra reference into a noncontiguous range.
5. *Range Reference Replace (RRR).* This operator replaces a reference in a noncontiguous range with another reference (generated using the RFR operator) not in the range.

We treat contiguous and noncontiguous 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 noncontiguous ranges can happen to any reference in the range. For range expansions, we have the following options from the cells within the spreadsheet:

1. If the cells in the range are in the same column, add a reference to a cell above or below the range.
2. If the cells in the range are in the same row, add a reference to a cell to the left or right of the range.
3. Include references generated by RFR mutations of the references within the range.

For range shrinkages, we have the following options:

1. Remove the reference at the beginning of a range.
2. Remove the reference at the end of a range.
3. Remove any reference from within a range.

The first two options are both applicable to contiguous and noncontiguous ranges. The third option is only applicable in the case of noncontiguous ranges.

Formula replacement. In addition to the reference and range operators, the Formula Replacement with Constant (FRC) operator is unique to spreadsheets. It overwrites the formula in a cell with the computed output value, given the set of inputs within the spreadsheet, 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 [82]. The FRC operator has been included in the suite of mutation operators to model this kind of error.

Function mutation. Users might sometimes 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.

TABLE 4
Sheets Used in Evaluations

Sheet	Cells		Mutants
	Fml	Total	Generated
Microgen	2	12	176
GradesNew	8	26	338
FitMachine	6	18	440
Digits	6	14	465
NetPay	6	18	108
Purchase	15	50	325
RandJury	21	58	886
Sales	16	29	338
Solution	3	12	235
Budget	6	24	158
MBTI	28	83	1145
NewClock	10	24	321
GradesBig	21	48	930
Harvest	9	26	231
Payroll	54	100	1404
Total	211	542	7500

We have developed an evaluation framework with Excel as the front end, and a back-end server developed in Haskell. Communications take place over a socket connection. The mutation operators have been implemented in Haskell and are part of the server. This framework allows us to easily carry out evaluations using the suite of mutation operators.

In the following three sections, we describe how the mutation operators were used in the evaluations of spreadsheet tools. Details regarding number of input cells, number of cells with formulas, and number of first-order mutants generated for each of the spreadsheets used are shown in Table 4. All of the spreadsheets were not used in all three evaluations. The analysis of mutation adequacy of du-adequate test suites described in Section 6 is based on the data from an empirical evaluation comparing two automatic test-case generators. The first 12 spreadsheets ("Microgen" through "NewClock") were used in this evaluation since the data for du coverage was available for these sheets. We added additional spreadsheets ("GradesBig," "Harvest," and "Payroll") to the evaluation of the spreadsheet debugger GoalDebug described in Section 7 to include more types of mutants. In the evaluation of the combined-reasoning system, described in the online supplement Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2008.73, we used two spreadsheets ("GradesNew" and "Payroll") that were used in actual user studies since we had data on the user interactions. This data was then used to model users in the evaluation of the combined-reasoning system.

6 Evaluation of Test Suites through Mutation Testing

In previous work, we have developed an automatic test-case generator, called AutoTest, to help users test their spreadsheets [44]. AutoTest generates test suites aimed at satisfying the du adequacy criterion. The idea behind the

du-coverage criterion is to test for each definition of a cell in the spreadsheet, all of its uses. In other words, the goal is to test all du pairs. In this section, we evaluate the mutation adequacy of the du-adequate test suites generated by AutoTest.

In a spreadsheet, every cell defines a value. Cells with conditionals generally give rise to two or more definitions, contained in the different branches. Likewise, one cell formula may contain different uses of a cell definition in different branches of conditionals or in the conditionals themselves. Therefore, it is incorrect to represent du pairs simply as pairs of cell addresses. Instead, we generally need paths to subformulas to identify definitions and uses.

6.1 Definition-Use Coverage

To formalize the notions of definitions and uses of cells, we employ an abstract tree representation of formulas that stores conditionals in internal nodes and conditional-free subformulas in the leaves. In this section, we briefly describe using an example how AutoTest generates du-adequate test cases given a spreadsheet formula. For a more detailed description of the system, please see [44].

A du pair is given by a definition and a use, which are both essentially represented by paths. To give a precise definition, we observe that while only expressions in leaves can be definitions, conditions in internal nodes as well as leaf expressions can be uses. Moreover, since a (sub)formula defining a cell might refer to other cells defined by conditionals, a single path is generally not sufficient to describe a definition. Instead, a definition is given by a set of paths. To generate constraints for a du pair, the constraints for a definition are combined with the constraints for a use. The attempt at solving the constraints for a du pair can have one of the following outcomes:

1. The constraint solver might succeed, in which case the solution is, for each input cell, a range of values that satisfies the constraints. For each cell, any value from the range can be used as a test input.
2. The constraint solver might fail. This situation arises when for at least one input cell the constraints cannot be solved. In this case, it is not possible to generate a test case that would execute the path under consideration. Therefore, the failure of the constraint solver indicates that the particular du pair (referred to as an *infeasible du pair*) cannot be exercised.

Consider a spreadsheet composed of the following three cells:

$$\begin{aligned}&\mathrm{A1}: 10\\&\mathrm{A2}: \mathrm{IF}(\mathrm{A1} > 15, 20, 30)\\&\mathrm{A3}: \mathrm{IF}(\mathrm{A2} > 25, \mathrm{A2}+10, \mathrm{A2}+20).\end{aligned}$$

Cell A1 is an input cell and therefore has one definition and does not use any other cells. A2 has one use of A1 (which is always executed) in the condition. Since the formula in A2 has two branches, it has two definitions with the constraints $C_1 \equiv \mathrm{A1} > 15$ and $C_2 \equiv \mathrm{A1} \leq 15$ for the true and false branch, respectively.

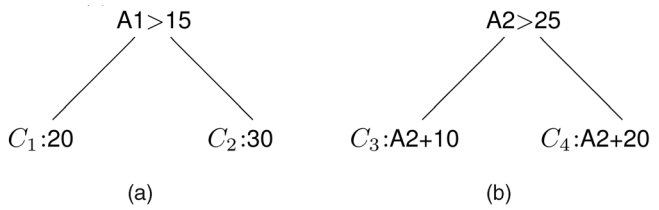


Fig. 1. Constraint trees. (a) Formula in A2. (b) Formula in A3.

The constraint tree for the formula in A2 is shown in Fig. 1a.

The formula in A3 has three uses of A2, one in the condition, and one each in the true and false branches. The use in the condition is always executed, but we need to satisfy the constraints $C_3 \equiv \mathrm{A2} > 25$ and $C_4 \equiv \mathrm{A2} \leq 25$ in order to execute the true and false branch, respectively. (The constraint tree for the formula in A3 is shown in Fig. 1b.) Therefore, to test the formula in A3 to satisfy the du-adequacy criterion, we need test cases that execute the two definitions of A2 and the two uses of A2 in the branches of A3's formula—a total of four du pairs. The constraints for the four du pairs are shown below:

$$\begin{aligned}&\{C_1 \equiv \mathrm{A2} = 20, C_3 \equiv \mathrm{A3} = \mathrm{A2}+10\},\\&\{C_1 \equiv \mathrm{A2} = 20, C_4 \equiv \mathrm{A3} = \mathrm{A2}+20\},\\&\{C_2 \equiv \mathrm{A2} = 30, C_3 \equiv \mathrm{A3} = \mathrm{A2}+10\},\\&\{C_2 \equiv \mathrm{A2} = 30, C_4 \equiv \mathrm{A3} = \mathrm{A2}+20\}.\end{aligned}$$

We consider the constraints individually in the following:

- $\{C_1 \equiv \mathrm{A2} = 20, C_3 \equiv \mathrm{A3} = \mathrm{A2}+10\}$: C_3 requires A2 to be some value greater than 25. However, satisfying C_1 results in 20 in A2. Therefore, the two constraints cannot be satisfied at the same time, and we have an infeasible du pair.
- $\{C_1 \equiv \mathrm{A2} = 20, C_4 \equiv \mathrm{A3} = \mathrm{A2}+20\}$: Satisfying C_1 results in 20 as the output of A2, which satisfies the constraint C_4. Solution of C_1 results in $\mathrm{A1} = 16$ as the test case to execute this du pair.
- $\{C_2 \equiv \mathrm{A2} = 30, C_3 \equiv \mathrm{A3} = \mathrm{A2}+10\}$: Satisfying C_2 results in an output of 30 in A2. This value satisfies the constraint C_3. Therefore, the test case that exercises this du pair is $\mathrm{A1} = 14$, which is obtained by solving C_2.
- $\{C_2 \equiv \mathrm{A2} = 30, C_4 \equiv \mathrm{A3} = \mathrm{A2}+20\}$: Satisfying C_2 results in 30 in A2. This value violates the constraint C_4. Therefore, this set of constraints cannot be solved and we have an infeasible du pair.

Even though the solution of a constraint might result in many test cases, AutoTest generates only one test case per feasible du pair. For example, any value greater than 15 in A1 would satisfy C_1, but AutoTest generates 16 as the test case. Moreover, in a more complicated spreadsheet, there might be several independent sets of constraints (representing different paths between a du pair) for a feasible du pair. In such cases, each solvable set of constraints could generate one or more test cases.

TABLE 5
Comparison of Number of Test Cases against Mutation Coverage

Spreadsheets	Mut.	Rand1			Rand2			RandAvg5			First			All		
	Gen.	Tests	Killed	Cov.	Tests	Killed	Cov.	Tests	Killed	Cov.	Tests	Killed	Cov.	Tests	Killed	Cov.
MicroGen	176	14	142	80.7	15	157	89.2	15	150	85.2	12	53	30.1	28	175	99.4
Grades	338	45	302	89.3	44	302	89.3	43.8	302	89.3	37	302	89.3	298	302	89.3
FitMachine	440	23	426	96.8	24	426	96.8	22.7	416.3	94.6	20	397	90.2	27	426	96.8
Digits	465	7	452	97.2	7	452	97.2	7	452	97.2	7	452	97.2	7	452	97.2
NetPay	108	4	108	100	4	108	100	4	90	83.3	4	108	100	4	108	100
PurchaseBudget	325	17	325	100	19	325	100	17.7	270.8	83.3	17	318	97.8	46	325	100
RandomJury	886	37	510	57.6	37	886	100	36.7	771.3	87.1	24	380	42.9	522	886	100
Sales	338	1	338	100	1	338	100	1	338	100	1	338	100	1	338	100
Solution	235	11	235	100	10	235	100	10.3	235	100	10	235	100	17	235	100
Budget	158	5	117	74.1	5	147	93	5	131.8	83.4	5	117	74.1	114	158	100
MBTI	1145	101	890	77.7	100	875	76.4	99.7	880.5	76.9	72	740	64.6	834	1011	88.3
NewClock	321	14	277	86.3	16	277	86.3	14.7	277	86.3	11	260	81	29	277	86.3

6.2 Experiment Setup

As discussed earlier in this paper, the primary goal of mutation testing is to evaluate the quality of test suites. An adequacy criterion like du adequacy is well defined in the sense that it is independent of the programming language or programming environment. On the contrary, the effectiveness of mutation testing depends on the design of the mutation operators. For example, a poor choice of mutation operators might certify that a test suite is 100 percent mutation adequate even when the program is poorly tested. The all-uses data-flow criterion for test adequacy was empirically evaluated against mutation adequacy in [39], and the authors of that paper found that mutation-adequate test suites are closer to satisfying the data-flow criterion and succeed at detecting more faults. It is not known if this result holds in the spreadsheet domain.

For the purpose of our experiment, we used spreadsheets that have been used in other evaluations described in [23], [44], [83]. We designed our experiment to answer the following research questions:

RQ1: *What is the mutation adequacy of the automatically generated test suites for each one of the spreadsheets?*

We have shown empirically in [44] that the test suites generated by AutoTest are 100 percent du adequate. RQ1 compares the effectiveness of du adequacy against mutation adequacy for each of the spreadsheets.

RQ2: *How effective are the automatically generated test suites at killing higher-order mutants?*

As a result of the coupling effect, a test suite that kills most of the first-order mutants must also be effective against higher-order mutants. In this context, we would like to determine what proportion of the higher-order mutants in comparison with the first-order mutants survive the du-path adequate test suites.

Since the mutation operators have been designed to mirror real-world faults in spreadsheets, answers to the research questions would give us a better idea of how effective the du-adequate test suite is at detecting faults. A single fault might have a very high impact. Therefore, ideally, we would like to kill as many of the nonequivalent mutants as possible.

6.3 Results

The data in Table 5 show the mutation adequacy of du-adequate test suites generated by AutoTest[4] under the following configurations:

1. *Rand1.* In this configuration, AutoTest picks a solvable constraint set at random from those available for a du pair. The randomly picked set of constraints is used to generate one test case for each du pair.
2. *Rand2.* This configuration is essentially the same as the previous case, except that the random number generator used a different seed value.
3. *RandAvg5.* The same configuration as above, except that the figures were averaged across five runs.
4. *First.* In this case, only one test case (from the first solvable set of constraints) was generated by AutoTest per feasible du pair.
5. *All.* In this configuration, we modified the system to generate test cases for all solvable constraint sets.

For each spreadsheet used in the evaluation, Table 5 contains the number of mutants generated, and the number of tests generated, number of mutants killed, and the mutation score for each configuration.

In configurations "Rand1," "Rand2," and "First," AutoTest only generates one test case be feasible du pair. We see from the data that the mutation adequacy of test suites generated using "First" is low in many cases. The configuration "All" gives an indication of the maximum possible mutation coverage that can be achieved by the current test generation approach used by AutoTest. This approach is expensive from a user's point of view since the number of test cases to be dealt with is much higher than the other approaches. Moreover, as far as du adequacy is concerned, this approach would potentially generate multiple test cases for the same du pair. The additional mutation coverage achieved under "All" is due to the large number of test cases.

Keeping the number of test cases as low as possible is important to minimize user effort. However, as can be seen from the data in Table 5, test suites with more test cases have higher mutation coverage in general. On the other hand, some test cases are more effective at killing mutants than others. For example, for the "NewClock" spreadsheet, the 14 test cases picked in "Rand1" were as mutation adequate as the 16 test cases picked by "Rand2." More importantly, the 29 test cases picked in "All" have the same mutation adequacy as the 14 test cases picked in "Rand1."

We can observe from the results in Table 5 that the test suites generated by "All" are successful at killing almost all

4. All the generated mutants were nonequivalent to the original spreadsheets.

TABLE 6
Mutation Adequacy of AutoTest Test Suites with Second-Order Mutants

Spreadsheets	Mutants generated	Mutants killed	Mutation coverage
MicroGen	7659	7659	100%
Grades	49221	49074	99.70%
FitMachine	76222	73251	96.10%
Digits	87746	87649	99.89%
NetPay	4584	4584	100%
PurchaseBudget	48880	48590	99.4%
RandomJury	373211	373211	100%
Sales	53928	53928	100%
Solution	18505	18505	100%
Budget	9819	9811	99.92%
MBTI	635259	603599	95.02%
NewClock	45193	45108	99.81%

the mutants.[5] That is, the test suites, which are 100 percent du-path adequate, are, in many cases, close to being 100 percent mutation adequate as well. Even so, the empirical results indicate that mutation adequacy might be a stronger test adequacy criterion than du adequacy. More importantly, especially when adopting the strategy of generating a single test case for every du pair, we need to pick those that help the overall test suite achieve higher mutation adequacy.

To answer RQ2, we looked at the mutation adequacy of the automatically generated test cases with second-order mutants. The coverage numbers for second-order mutants using the "All" approach is shown in Table 6. Comparing the figures in Table 5 with those in Table 6, we see that the same number of test cases achieve higher (in some cases) or similar (in other cases) mutation scores when applied to second-order mutants. This result confirms similar results observed in the case of general-purpose programming languages [77], [78], [79]. From the numbers in Table 6, we see that a considerably higher number of second-order mutants are generated compared to first-order mutants. This factor, in itself, would make mutation testing using higher-order mutants time consuming and computationally expensive to the point of being infeasible. In this context, the observation that test suites that are effective at killing first-order mutants are equally effective at killing second-order mutants as well, indicates that we only need to carry out mutation testing using first-order mutants.

7 Evaluation of a Spreadsheet Debugger

GoalDebug is a spreadsheet debugger that allows users to mark cells that contain incorrect values and specify their expected output [45]. The system uses this information to generate a list of formula and value-change suggestions, any one of which, when applied, would result in the user-specified output in the cell. Typically, many change suggestions are generated and the user would have to go over them to identify the correct change. To minimize the effort required of the user, we have incorporated a set of heuristics that rank the suggestions. As far as performance of GoalDebug is concerned, an *effective* change suggestion is one which would correct the error, and an *effective* set of ranking heuristics is one which would assign a high rank (1 ideally) to the correct change suggestions consistently.

5. The number of test cases reported for "All" in Table 5 is the sum of all tests generated when using all the constraints. This number might be lower if we only considered *effective test cases*, where an effective test case is one that kills at least one mutant that has not been killed by a previous test.

TABLE 7
Sheets Used in the Evaluation of GoalDebug

Sheet	Cells		Mutants		
	Fml	Total	Irreversible	Reversible	Total
Microgen	2	12	143	33	176
GradesNew	8	26	157	181	338
FitMachine	6	18	366	74	440
Digits	6	14	172	293	465
NetPay	6	18	61	47	108
Purchase	15	50	172	153	325
RandJury	21	58	578	308	886
Sales	16	29	0	338	338
Solution	3	12	119	116	235
Budget	6	24	46	112	158
MBTI	28	83	902	243	1145
NewClock	10	24	156	165	321
GradesBig	21	48	283	647	930
Harvest	9	26	10	221	231
Payroll	54	100	347	1057	1404
Total	211	542	3512	3988	7500

7.1 Experiment Setup

To be able to evaluate the effectiveness of the change inference and ranking heuristics of GoalDebug, we decided to use spreadsheets from the study described in the previous section, together with a few we have used in other studies. In general, not all mutation operators are applicable to all formulas. Table 7 shows the summary information about the sheets used in the evaluation. Note that some of the details from Table 4 have been reproduced in Table 7 for the convenience of the reader. For each spreadsheet, the following information is given:

1. Number of formula cells in the spreadsheet (Fml).
2. Total number of cells in the spreadsheet (Total).
3. The number of *irreversible* mutants that were generated. Irreversible mutants are the mutated formulas that evaluate to the same value as the original formula, given the input values in the original spreadsheet, and thus cannot produce failures that could be identified by the user. Therefore, GoalDebug is principally inapplicable in those cases and cannot be invoked to generate change suggestions since the computed output and expected output are the same.
4. The number of *reversible mutants* that were generated. In these cases, the mutant formulas evaluate to values that are different from the values produced by the original formulas, and GoalDebug can be invoked on those cells.
5. Total number of mutants generated from the given sheet.

For the purpose of this study, we excluded formula-deletion operator (FDL) and formula replace with constant operator (FRC) from the suite of operators. The FDL operator was excluded since it seems unlikely that a user would mark an empty cell as incorrect and specify the expected output. Along similar lines, the FRC operator was excluded because

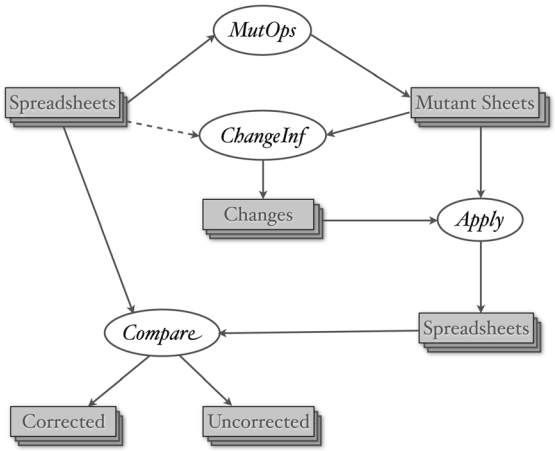


Fig. 2. Setup for evaluating change suggestions.

the mutation is not a minor change to reverse. To be able to reverse an FRC mutation, given the cell and the expected output, GoalDebug would have to come up with a formula that would compute the expected output in the marked cell. In practice, GoalDebug could accommodate these mutations by searching for candidate formulas within the spreadsheet, which if copied and pasted to the cell would result in the expected value. The formula changes could then be ranked on the basis of similarity of regions or distance from the target cell.

The experimental setup is shown in Fig. 2. We start with the set of spreadsheets that are of interest and use the mutation operators to generate a (much larger) set of mutant spreadsheets. The input cells are not mutated. We use the computed output values in the original spreadsheets to specify the expectations for the mutant cells in those cases where the computed outputs differ between the original and mutated spreadsheets. The change inference process is carried out on the mutant spreadsheets to generate a list of change suggestions. We then apply each generated change suggestion to each of the mutants to obtain a new set of spreadsheets. Now these spreadsheets can be compared with the original spreadsheet to verify if the effect of the mutation was reversed by the application of any one of the generated change suggestions. In cases in which any one of the change suggestions reversed the effect of the mutation, we also recorded the rank of the suggestion.

7.2 Preliminary Evaluation

The number of mutants that have not been corrected by the original version of GoalDebug are shown in Table 8. We carried out the evaluation using the suite of mutation operators to get an idea of what kind of extensions are required for GoalDebug to be able to suggest changes for a wide range of faults.

It should be clear from the results in Table 8 that the original version of GoalDebug was ineffective against a wide variety of spreadsheet faults. All of the uncorrected mutants are created by the nine operators shown in Table 8. GoalDebug did not have any change inference mechanism to reverse the errors seeded by the following operators: AOR, CRR, LCR, NRE, NRS, and ROR. To increase the range of errors for which GoalDebug would be helpful, it was important to extend the system to include these classes of errors. We also wanted to improve the system's coverage on errors seeded by the CRP, RFR, and RRR operators.

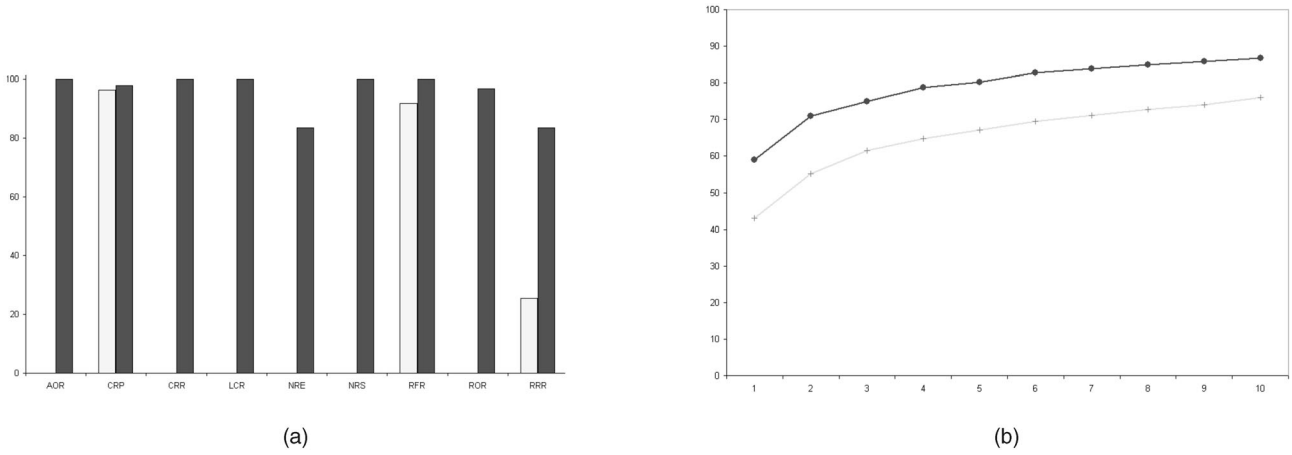
From the preliminary evaluation, we identified two areas in which GoalDebug could be improved: 1) The change inference mechanism needed to be substantially expanded to include a wider range of error situations the system could recover from. Without this extension, GoalDebug would have limited use for real-world errors. The modification of the system would result in a much higher number of change suggestions being generated under any given condition. This problem required us to 2) carry out enhancements to the ranking heuristics so that the system performs better even with the higher number of generated suggestions. The goal of refining the ranking heuristics is to ensure that the *correct* suggestions are assigned high ranks to minimize the

TABLE 8
Original Version of GoalDebug's Effectiveness at Correcting Mutations

Sheet	Operators: Uncorrected [Total]								
	AOR	CRP	CRR	LCR	NRE	NRS	RFR	ROR	RRR
Microgen	6 [6]	0 [3]	4 [4]	1 [1]	0 [0]	0 [0]	3 [16]	3 [3]	0 [0]
GradesNew	18 [18]	0 [4]	25 [25]	1 [1]	0 [0]	0 [0]	16 [123]	10 [10]	0 [0]
FitMachine	9 [9]	3 [6]	11[11]	1 [1]	0 [0]	0 [0]	5 [41]	6 [6]	0 [0]
Digits	62 [62]	0 [17]	43 [43]	0 [0]	0 [0]	0 [0]	27 [152]	19 [19]	0 [0]
NetPay	3 [3]	0 [6]	12 [12]	0 [0]	0 [0]	0 [0]	0 [22]	4 [4]	0 [0]
Purchase	14 [14]	0 [4]	29 [29]	0 [0]	0 [0]	0 [0]	3 [91]	15 [15]	0 [0]
RandJury	87 [87]	0 [47]	33 [33]	6 [6]	0 [0]	0 [0]	6 [119]	16 [16]	0 [0]
Sales	72 [72]	0 [12]	49 [49]	0 [0]	0 [0]	0 [0]	24 [205]	0 [0]	0 [0]
Solution	21 [21]	0 [2]	12 [12]	0 [0]	0 [0]	0 [0]	11 [77]	4 [4]	0 [0]
Budget	15 [15]	0 [1]	18 [18]	0 [0]	0 [0]	0 [0]	10 [75]	3 [3]	0 [0]
MBTI	43 [43]	0 [24]	35 [35]	16 [16]	0 [0]	0 [0]	3 [120]	5 [5]	0 [0]
NewClock	20 [20]	1 [11]	22 [22]	1 [1]	0 [0]	0 [0]	0 [97]	14 [14]	0 [0]
GradesBig	6 [6]	3 [5]	28 [28]	1 [1]	99 [99]	27 [27]	12 [103]	14 [14]	272 [364]
Harvest	0 [0]	0 [0]	5 [5]	0 [0]	40 [40]	18 [18]	0 [24]	0 [0]	99 [134]
Payroll	170 [170]	0 [42]	166 [166]	0 [0]	0 [0]	0 [0]	39 [641]	38 [38]	0 [0]
Total	546 [546]	7 [184]	492 [492]	27 [27]	139 [139]	45 [45]	159 [1906]	151 [151]	371 [498]
Uncorrected	100%	3.8%	100%	100%	100%	100%	8.3%	100%	74.5%

TABLE 9
GoalDebug's Effectiveness at Correcting Mutations after Enhancements

Sheet	Operators: Uncorrected [Total]								
	AOR	CRP	CRR	LCR	NRE	NRS	RFR	ROR	RRR
Microgen	0 [6]	0 [3]	0 [4]	0 [1]	0 [0]	0 [0]	0 [16]	0 [3]	0 [0]
GradesNew	0 [18]	0 [4]	0 [25]	0 [1]	0 [0]	0 [0]	0 [123]	0 [10]	0 [0]
FitMachine	0 [9]	2 [6]	0 [11]	0 [1]	0 [0]	0 [0]	0 [41]	0 [6]	0 [0]
Digits	0 [62]	0 [17]	0 [43]	0 [0]	0 [0]	0 [0]	0 [152]	0 [19]	0 [0]
NetPay	0 [3]	0 [6]	0 [12]	0 [0]	0 [0]	0 [0]	0 [22]	0 [4]	0 [0]
Purchase	0 [14]	0 [4]	0 [29]	0 [0]	0 [0]	0 [0]	0 [91]	0 [15]	0 [0]
RandJury	0 [87]	0 [47]	0 [33]	0 [6]	0 [0]	0 [0]	0 [119]	0 [16]	0 [0]
Sales	0 [72]	0 [12]	0 [49]	0 [0]	0 [0]	0 [0]	0 [205]	0 [0]	0 [0]
Solution	0 [21]	0 [2]	0 [12]	0 [0]	0 [0]	0 [0]	0 [77]	0 [4]	0 [0]
Budget	0 [15]	0 [1]	0 [18]	0 [0]	0 [0]	0 [0]	0 [75]	0 [3]	0 [0]
MBTI	0 [43]	0 [24]	0 [35]	0 [16]	0 [0]	0 [0]	0 [120]	0 [5]	0 [0]
NewClock	0 [20]	1 [11]	0 [22]	0 [1]	0 [0]	0 [0]	0 [97]	0 [14]	0 [0]
GradesBig	0 [6]	1 [5]	0 [28]	0 [1]	23 [99]	0 [27]	0 [103]	5 [14]	73 [364]
Harvest	0 [0]	0 [0]	0 [5]	0 [0]	0 [40]	0 [18]	0 [24]	0 [0]	10 [134]
Payroll	0 [170]	0 [42]	0 [166]	0 [0]	0 [0]	0 [0]	0 [641]	0 [38]	0 [0]
Total	0 [546]	4 [184]	0 [492]	0 [27]	23 [139]	0 [45]	0 [1906]	5 [151]	83 [498]
Uncorrected	0%	2.2%	0%	0%	16.5%	0%	0%	3.3%	16.7%



(a) (b)

Fig. 3. Coverage and ranking comparisons. (a) Comparison of coverage percent. (b) Cumulative coverage of ranking heuristics.

effort invested by the users while debugging faults in spreadsheets.

7.3 Evaluation of the Extended System

The effectiveness scores after extending the change inference system are shown in Table 9. Fig. 3a shows comparisons of these scores with those from the old system in Table 8. For each of the mutation operators, the percentage coverage of the old system (in lighter shade) is shown against the coverage of the new system (in darker shade). The extensions to the change inference mechanism increased the ability of GoalDebug to recover from a much wider spectrum of errors as can be seen from the plot.

The extended change inference system generates more suggestions than the original system. We had to make improvements to the ranking heuristics to cope with the increase in number of change suggestions. Therefore, to evaluate the new ranking heuristics, for each of the mutants reversed by the new change inference mechanism, we compare the rank assigned by the old version of the ranking heuristics against the rank assigned by the new version. The Wilcoxon test showed that the new ranking heuristics perform significantly better than the old ones ($p < 0.001$). Ideally, the correct change suggestion should be ranked within the top five ranks, thereby minimizing the effort the user would have to expend to locate it. The difference in ranks assigned by the two techniques is more important at high ranks than at low ones. For example, a difference of 5 between ranks 1 and 6 is more important than a difference of 5 between ranks 100 and 105. To take this effect into consideration, we also ran tests on the reciprocals of the ranks generated by the two techniques. Once again, the Wilcoxon test showed that the new ranking techniques perform significantly better than the old ones ($p < 0.001$).

Operator	p
AOR	< 0.001
CRP	< 0.001
CRR	< 0.001
LCR	0.008
NRE	0.036
NRS	0.005
RFR	< 0.001
ROR	< 0.001
RRR	< 0.001

Since the mutation operators reflect different kinds of errors that can occur in spreadsheet formulas, we also compared the performance of the ranking heuristics for each operator. The new heuristics are significantly better than the old ones for all operators as shown by the p-values reported in the table on the left.

The cumulative coverage percentages across ranks for the new heuristics (in dark red) are compared against those for the old (in light blue) in Fig. 3b. With the new heuristics in effect, the top ranked suggestion corrects the mutations in 59 percent of the cases, the top two suggestions correct the mutations in 71 percent of the cases, and so on. Out of the 3,988 reversible mutants generated, the suggestion that corrects the mutation is ranked in the top five in 80 percent of the cases with the new ranking heuristics as opposed to only 67 percent of the cases with the old version of the system.

7.4 Summary

The evaluation of GoalDebug using the suite of mutation operators helped us identify classes of errors GoalDebug was not effective against. We were able to use this information to extend the change inference mechanism of GoalDebug. Furthermore, it also enabled us to refine the ranking heuristics. Since the suite of mutation operators has been designed to reflect real-world faults, the evaluation and subsequent modifications of the system have made GoalDebug effective against a wider range of potential errors.

8 Conclusions and Future Work

In this paper, we have presented a suite of mutation operators for spreadsheets that allows users to evaluate the mutation adequacy of their test suites. In addition to testing spreadsheets, the suite of mutation operators has been successfully employed to carry out empirical studies of spreadsheet tools. We have described three such cases (one can be found in the online supplement) in which the operators helped us evaluate and improve tools we have developed.

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. The operators described in this paper are targeted at individual spreadsheets (that is, without an editing history) and hence do not reflect errors that can arise from update operations. For example, incorrect use of relative and absolute references might result in errors that occur during row/column insert/delete operations. We are currently working on operators that reflect update errors. A more complete list of mutation operators would allow a practitioner to adapt the operators to other applications more easily.

As part of the ViTSL/Gencel framework, we have developed a system that automatically infers ViTSL templates from spreadsheets [84]. A preliminary evaluation we carried out has shown that the system works well in practice. In future work, we plan to use the mutation operators to investigate how well template inference works in the presence of errors in the spreadsheet.

Appendix A

Evaluation of Effectiveness of Combined-Reasoning Approaches

None of the tools described in Section 2 of this paper protects the user from all spreadsheet errors. Each tool has its own strengths and weaknesses, and there is also a cost-benefit trade-off associated with their use. We studied the effects of combining WYSIWYT and UCheck [47] to find out whether a combination could result in an overall improved approach to error detection. As mentioned earlier in this paper, WYSIWYT is a framework for testing spreadsheets, and UCheck is an automatic consistency checker. To study the effectiveness of combining WYSIWYT and UCheck, we used the suite of mutation operators to seed errors in the spreadsheets used in the evaluation. This approach allowed us to evaluate the error detection and reporting mechanism of the two systems independently and together over a wide range of possible error situations.

This study was made possible by the suite of mutation operators. We used the operators to seed errors within the spreadsheets used in the evaluation. Each user's interactions with WYSIWYT were simulated using a model we built based on empirical data of user interactions collected from a previous study [85]. Using this approach, we could combine WYSIWYT's reasoning, based on the spreadsheet and the user's inputs, with UCheck's reasoning, which is based solely on the data and formulas within the spreadsheet.

Both WYSIWYT and UCheck use cell shading for communicating fault localization information to the user. Prior empirical research has shown that users consistently debug the darkest-shaded cells in the spreadsheet [85]. Therefore, it is important to ensure that the darkest shading corresponds to the actual location of errors within the spreadsheet. As shown in [85], there are two aspects to any error reporting mechanism—an *information base* consists of the information used to locate the faults, and a *mapping* determines how the information base is actually transformed into fault localization feedback and presented to the user. Our research goals in carrying out the evaluation were to find the most effective heuristics for selecting and combining feedback from the two systems and to identify the classes of faults that are caught by the combined system when compared to WYSIWYT and UCheck working independently.

A.1 Fault Localization

The fault localization mechanism of WYSIWYT relies on user judgments about the correctness of cell output to isolate formulas containing faults. During the course of developing the spreadsheet, users can place $\surd$ or x to indicate to the system that the computed output of a cell is right or wrong. The $\surd$ marks increase the testedness of the spreadsheet. The system combines the dependencies between the cell formulas and the location of the $\surd$ and x marks placed by the user to estimate the likelihoods of faults being located in various cells. Based on this estimate, the system shades the interior of cells light-to-dark amber.

UCheck uses orange shading for cells that are the site of unit errors and yellow shading for cells whose formulas have references to cells that are the site of unit errors. WYSIWYT,

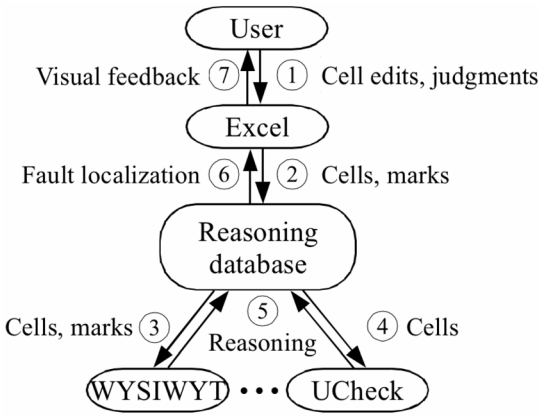


Fig. 4. Combining reasoning from UCheck and WYSIWYT.

on the other hand, has a six-point scale of shading—unshaded (or zero, indicating no error) to darkest shade (or five, indicating high likelihood of error). To integrate UCheck's three-point scale with scores on WYSIWYT's six-point scale, we assigned a score of zero to unshaded cells, one to cells that are shaded yellow, and five to cells shaded orange by UCheck since the system is reporting unit errors with 100 percent confidence in these cases. The degree of shading of a cell c is returned by the function *score(c)*.

Intuitively, the fault-localization feedback provided to the user can be considered to be more effective when the faulty cells are all shaded dark, and the nonfaulty cells are shaded (if at all) light. We compute the *visual effectiveness (VE)* of the feedback in terms of the set of formula cells with correct and incorrect formulas, and the degree of shading [86]:

$$VE = \sum_{c \in Faulty} \frac{score(c)}{|Faulty|} - \sum_{c \in Correct} \frac{score(c)}{|Correct|}.$$

A.2 Experiment Setup

In the evaluation, we compared three heuristics for combining the feedback computed independently by UCheck and WYSIWYT:

1. *ComboMax* returns the darkest shading received from the two systems for each of the cells.
2. *ComboAverage* returns the average of the scores of the shading computed by the two systems.
3. *ComboMin* returns the lightest cell shading received from the two systems.

We also considered two different mappings from the information base. For the *original mapping*, we used the scores from the actual shading generated by the two systems. For the *threshold mapping*, we ignored cells shaded yellow by UCheck and cells with very low fault likelihood (shading levels zero and one) as reported by WYSIWYT. The rest of the cells were treated as if they were shaded with the darkest hue. In both mappings, we considered the three different strategies (*ComboMax*, *ComboAverage*, and *ComboMin*) for combining feedback.

The architecture of the combined reasoning system is shown in Fig. 4. The cell edits, and other inputs from the user are captured by Excel. This information is then sent to the reasoning database. UCheck carries out unit checking based on the spreadsheet information stored in the reasoning database, and sends back information about unit errors into the database. WYSIWYT employs information about the spreadsheet cells and user judgments to calculate fault likelihood of the cells, and stores this data in the database. Based on the inputs from UCheck and WYSIWYT, the fault localization information is computed, and the results are presented to the user as visual feedback.

TABLE 10
Probabilistic User Model

Value	Formula	✓	✗
Incorrect	Incorrect	**74%**	26%
	Correct	**75%**	25%
Correct	Incorrect	50%	**50%**
	Correct	99%	**1%**

In this evaluation, we used the spreadsheets used in the study described in [85]. This choice was made since we already had empirical data on user interactions with the spreadsheets used in the study. The input values used were actual test cases developed by the subjects in the study described in [85]. The mutation operators were used to seed errors in the spreadsheets. The mutants were then checked using UCheck and WYSIWYT, and the fault localization information was recorded for further analysis.

UCheck is fully automatic, whereas user input ($\surd$ and x marks) is a key component of the WYSIWYT fault-localization mechanism. In our evaluation of the combined system, we modeled user behavior based on data from prior empirical work [85], [86]. It has been observed that users mark 85 percent of the cells while testing and debugging spreadsheets, placing $\surd$ marks more frequently than x marks. Empirical data from previous studies, summarized in Table 10, shows that users made mistakes while putting marks in cells. Such incorrect decisions made during testing and debugging are referred to as *oracle mistakes* [86].

Since WYSIWYT is a framework for testing, the designers of the system expect the users to place $\surd$ or x marks in cells based on their computed output. The data in Table 10 shows that, in 74 percent of the cases, users place a $\surd$ in cells whose formula and computed output value are both wrong. Along similar lines, in 1 percent of the cases, users place a x mark in cells whose formula and computed output are both correct. However, it has been observed that in some cases the users place the marks depending on their judgments about the correctness of the cell formulas. The users, in such cases, are actually not just testing anymore. In 75 percent of the cases, users place a $\surd$ in a cells whose value is incorrect (probably due to one or more faulty cells upstream in the computation) in spite of the formula being correct. Similarly, in 50 percent of the cases, users place an x mark in cells whose formula is incorrect in spite of the output value being correct.

The numbers in bold font in Table 10 show false positive ($\surd$ on incorrect value) and false negative (x on correct value). Even in cases where the output of a cell was incorrect, users placed $\surd$ marks (false positives) more often than x marks. However, in cases where the output of a cell was correct, users were unlikely to place an x mark on that

TABLE 11
Observed Faults versus Reasoning

Contributor	CRP	AOR	CRR	RFR	NRE	RRR	NRS	ROR	LCR
Neither	0	2	11	35	0	7	14	1	3
WYSIWYT only	3	7	13	50	0	2	13	5	0
Both (WYSIWYT ∩ UCheck)				25	38	125			
UCheck only				26	85	235			

cell (false negative). The data indicates that users made incorrect testing decisions between 5 percent and 20 percent of the time. We used these probabilities to model user input in our empirical evaluation.

A.3 Results

The results of our evaluation show that WYSIWYT and UCheck differ in their ability to uncover various types of faults. Table 11 summarizes the observed faults and the feedback provided by the systems independently and in conjunction. We see that while UCheck is very effective at detecting reference mutations, WYSIWYT detects a broader range of faults. Therefore, UCheck and WYSIWYT complement each other.

The visual effectiveness scores for the various combinations we considered are shown, sorted in descending order, in Tables 12 (original mapping) and 13 (threshold mapping). *ComboMax* scores the highest VE scores for both mappings.

UCheck and WYSIWYT support different approaches to spreadsheet development. UCheck's effectiveness is dependent on strings in cells that serve as row and column headers. WYSIWYT, on the other hand, relies on the dataflow relationships between the cells and the user judgments about the correctness of cell outputs. If a spreadsheet contains cells that provide labeling information, UCheck offers consistency checking for free, which leaves fewer number of errors the user would have to debug later. On the other hand, WYSIWYT can help identify faults through testing. However, in this case, it is likely that a much greater effort would be required to identify all faults through testing alone. As far as design of fault localization feedback is concerned, *ComboMin* would be the strategy to adopt to provide conservative feedback, whereas, *ComboMax* would be the ideal approach if higher VE is the goal.

TABLE 12
Visual Effectiveness Scores with Original Mapping

Reasoning	Faulty	−	Correct	=	VE
Combo Max	1.730	−	0.119	=	1.611
UCheck	1.526	−	0.042	=	1.484
Combo Average	1.063	−	0.064	=	0.999
WYSIWYT	0.600	−	0.085	=	0.515
Combo Min	0.396	−	0.008	=	0.388

TABLE 13
Visual Effectiveness Scores with Threshold Mapping

Reasoning	Faulty	−	Correct	=	VE
Combo Max	4.043	−	0.166	=	3.877
UCheck	3.814	−	0.106	=	3.708
Combo Average	2.339	−	0.093	=	2.246
WYSIWYT	0.864	−	0.079	=	0.785
Combo Min	0.636	−	0.019	=	0.617

A.4 Summary

In the evaluation of combining the reasoning from UCheck with that from WYSIWYT, the mutation operators allowed us to generate spreadsheets with a wide range of faults in them. Since the evaluation was performed using a model of user interactions based on empirical data from previous studies, we were also able to simulate user behavior assuming varying levels of expertise. Such an extensive analysis would prohibitively expensive if we were limited to actual user studies.

ACKNOWLEDGMENTS

This work was partially supported by the US National Science Foundation under Grant ITR-0325273 and by the EUSES Consortium http://EUSESconsortium.org).

REFERENCES

[1] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the Numbers of End Users and End User Programmers," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing,* pp. 207-214, 2005.
[2] S. Ditlea, "Spreadsheets Can Be Hazardous to Your Health," *Personal Computing,* vol. 11, no. 1, pp. 60-69, 1987.
[3] K. Rajalingham, D. Chadwick, B. Knight, and D. Edwards, "Quality Control in Spreadsheets: A Software Engineering-Based Approach to Spreadsheet Development," *Proc. 33rd Hawaii Int'l Conf. System Sciences,* pp. 1-9, 2000.
[4] G.J. Croll, "The Importance and Criticality of Spreadsheets in the City of London," *Proc. Symp. European Spreadsheet Risks Interest Group,* 2005.
[5] *European Spreadsheet Risks Interest Group,* EuSpRIG, http://www.eusprig.org/, 2008.
[6] R.R. Panko and R.P. Halverson Jr., "Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks," *Proc. 29th Hawaii Int'l Conf. System Sciences,* 1996.
[7] R.R. Panko, "Spreadsheet Errors: What We Know. What We Think We Can Do," *Proc. Symp. European Spreadsheet Risks Interest Group,* 2000.
[8] B. Ronen, M.A. Palley, and H.C. Lucas Jr., "Spreadsheet Analysis and Design," *Comm. ACM,* vol. 32, no. 1, pp. 84-93, 1989.
[9] A.G. Yoder and D.L. Cohn, "Real Spreadsheets for Real Programmers," *Proc. Int'l Conf. Computer Languages,* pp. 20-30, 1994.
[10] T. Isakowitz, S. Schocken, and H.C. Lucas Jr., "Toward a Logical/Physical Theory of Spreadsheet Modelling," *ACM Trans. Information Systems,* vol. 13, no. 1, pp. 1-37, 1995.
[11] S.G. Powell and K.R. Baker, *The Art of Modeling with Spreadsheets: Management Science, Spreadsheet Engineering, and Modeling Craft.* Wiley, 2004.
[12] R.R. Panko, "Applying Code Inspection to Spreadsheet Testing," *J. Management Information Systems,* vol. 16, no. 2, pp. 159-176, 1999.
[13] J. Sajaniemi, "Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization," *J. Visual Languages and Computing,* vol. 11, no. 1, pp. 49-82, http://www.idealibrary.com, 2000.

[14] R. Mittermeir and M. Clermont, "Finding High-Level Structures in Spreadsheet Programs," *Proc. Ninth Working Conf. Reverse Eng.*, pp. 221-232, 2002.

[15] M. Erwig and M.M. Burnett, "Adding Apples and Oranges," *Proc. Fourth Int'l Symp. Practical Aspects of Declarative Languages*, pp. 173-191, 2002.

[16] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi, "A Type System for Statically Detecting Spreadsheet Errors," *Proc. 18th IEEE Int'l Conf. Automated Software Eng.*, pp. 174-183, 2003.

[17] T. Antoniu, P.A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen, "Validating the Unit Correctness of Spreadsheet Programs," *Proc. 26th IEEE Int'l Conf. Software Eng.*, pp. 439-448, 2004.

[18] M.M. Burnett, C. Cook, J. Summet, G. Rothermel, and C. Wallace, "End-User Software Engineering with Assertions," *Proc. 25th IEEE Int'l Conf. Software Eng.*, pp. 93-103, 2003.

[19] M.J. Coblenz, A.J. Ko, and B.A. Myers, "Using Objects of Measurement to Detect Spreadsheet Errors," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, pp. 314-316, 2005.

[20] R. Abraham and M. Erwig, "Header and Unit Inference for Spreadsheets through Spatial Analyses," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, pp. 165-172, 2004.

[21] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger, "Automatic Generation and Maintenance of Correct Spreadsheets," *Proc. 27th IEEE Int'l Conf. Software Eng.*, pp. 136-145, 2005.

[22] G. Engels and M. Erwig, "ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications," *Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 124-133, 2005.

[23] G. Rothermel, M.M. Burnett, L. Li, C. DuPuis, and A. Sheretov, "A Methodology for Testing Spreadsheets," *ACM Trans. Software Eng. and Methodology*, vol. 10, no. 2, pp. 110-147, 2001.

[24] M. Fisher II, M. Cao, G. Rothermel, C. Cook, and M.M. Burnett, "Automated Test Case Generation for Spreadsheets," *Proc. 24th IEEE Int'l Conf. Software Eng.*, pp. 141-151, 2002.

[25] 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 Trans. Software Eng.*, vol. 28, no. 6, pp. 576-594, June 2002.

[26] *End Users Shaping Effective Software*, EUSES, http://EUSESconsortium.org, 2008.

[27] R.G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Trans. Software Eng.*, vol. 3, no. 4, pp. 279-290, July 1977.

[28] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34-41, 1978.

[29] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Program Mutation: A New Approach to Program Testing," *Infotech State of the Art Report, Software Testing*, vol. 2, pp. 107-126, 1979.

[30] T.A. Budd, F.G. Sayward, R.A. DeMillo, and R.J. Lipton, "The Design of a Prototype Mutation System for Program Testing," *Proc. Nat'l Computer Conf.*, pp. 623-627, 1978.

[31] A.T. Acree, "On Mutation," PhD dissertation, Georgia Inst. of Technology, 1980.

[32] T.A. Budd, "Mutation Analysis of Program Test Data," PhD dissertation, Yale Univ., 1980.

[33] 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," *Proc. Seventh ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pp. 220-233, 1980.

[34] J.H. Andrews, L.C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" *Proc. 27th IEEE Int'l Conf. Software Eng.*, pp. 402-411, 2005.

[35] E.J. Weyuker, S.N. Weiss, and D. Hamlet, "Comparison of Program Testing Strategies," *Proc. Fourth Symp. Software Testing, Analysis and Verification*, pp. 154-164, 1991.

[36] P.J. Frankl and E.J. Weyuker, "A Formal Analysis of the Fault-Detecting Ability of Testing Methods," *IEEE Trans. Software Eng.*, vol. 19, no. 3, pp. 202-213, Mar. 1993.

[37] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 367-427, Dec. 1997.

[38] 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," *J. Functional Programming*, vol. 11, no. 2, pp. 155-206, Mar. 2001.

[39] A.J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An Experimental Evaluation of Data Flow and Mutation Testing," *Software: Practice and Experience*, vol. 26, no. 2, pp. 165-176, 1996.

[40] J. Ruthruff, E. Creswick, M.M. Burnett, C. Cook, S. Prabhakararao, M. Fisher II, and M. Main, "End-User Software Visualizations for Fault Localization," *Proc. ACM Symp. Software Visualization*, pp. 123-132, 2003.

[41] 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," *Proc. IEEE Int'l Symp. Human-Centric Computing Languages and Environments*, pp. 203-210, 2003.

[42] C. Allwood, "Error Detection Processes in Statistical Problem Solving," *Cognitive Science*, vol. 8, no. 4, pp. 413-437, 1984.

[43] S.G. Powell, K.R. Baker, and B. Lawson, "A Critical Review of the Literature on Spreadsheet Errors," working paper, http://mba.tuck.dartmouth.edu/spreadsheet/product_pubs.html, 2006.

[44] R. Abraham and M. Erwig, "AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, pp. 43-50, 2006.

[45] R. Abraham and M. Erwig, "Goal-Directed Debugging of Spreadsheets," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, pp. 37-44, 2005.

[46] R. Abraham and M. Erwig, "GoalDebug: A Spreadsheet Debugger for End Users," *Proc. 29th IEEE Int'l Conf. Software Eng.*, pp. 251-260, 2007.

[47] J. Lawrence, R. Abraham, M.M. Burnett, and M. Erwig, "Sharing Reasoning about Faults in Spreadsheets: An Empirical Study," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, pp. 35-42, 2006.

[48] R. Abraham and M. Erwig, "UCheck: A Spreadsheet Unit Checker for End Users," *J. Visual Languages and Computing*, vol. 18, no. 1, pp. 71-95, 2007.

[49] K.N. King and A.J. Offutt, "Fortran Language System for Mutation-Based Software Testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685-718, July 1991.

[50] M. Delamaro and J. Maldonado, "Proteum—A Tool for the Assessment of Test Adequacy for C Programs," *Proc. Conf. Performability in Computing Systems*, pp. 79-95, 1996.

[51] Y. Ma, A.J. Offutt, and Y. Kwon, "MuJava: An Automated Class Mutation System," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97-133, 2005.

[52] P. Anbalagan and T. Xie, "Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs," *Proc. Second Workshop Mutation Analysis*, 2006.

[53] N. Mansour and M. Houri, "Testing Web Applications," *Information and Software Technology*, vol. 48, no. 1, pp. 31-42, 2006.

[54] W. Xu, A.J. Offutt, and J. Luo, "Testing Web Services by XML Perturbation," *Proc. IEEE Int'l Symp. Software Reliability Eng.*, pp. 257-266, 2005.

[55] J. Tuya, M.J. Suarez-Cabal, and C. Riva, "Mutating Database Queries," *Information and Software Technology*, vol. 49, no. 4, pp. 398-417, 2007.

[56] A.J. Offutt, P. Ammann, and L. Liu, "Mutation Testing Implements Grammar-Based Testing," *Proc. Second Workshop Mutation Analysis*, 2006.

[57] A.J. Offutt and R.H. Untch, "Mutation 2000: Uniting the Orthogonal," *Mutation 2000: Mutation Testing in the Twentieth and the Twenty-First Centuries*, pp. 45-55, 2000.

[58] M.R. Woodward, "Mutation Testing—Its Origin and Evolution," *Information and Software Technology*, vol. 35, no. 3, pp. 163-169, Mar. 1993.

[59] W.E. Wong, J.C. Maldonado, M.E. Delamaro, and S. Souza, "Use of Proteum to Accelerate Mutation Testing in C Programs," *Proc. Third ISSAT Int'l Conf. Reliability and Quality in Design*, pp. 254-258, 1997.

[60] R.H. Untch, A.J. Offutt, and M.J. Harrold, "Mutation Analysis Using Program Schemata," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 139-148, 1993.

[61] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Trans. Software Eng.*, vol. 8, no. 4, pp. 371-379, Apr. 1992.

[62] A.P. Mathur, "Performance, Effectiveness, and Reliability Issues in Software Testing," *Proc. 15th Ann. Int'l Computer Software and Applications Conf.*, pp. 604-605, 1991.

[63] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 2, pp. 99-118, 1996.
[64] R.A. DeMillo and A.J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Trans. Software Eng.,* vol. 17, no. 9, pp. 900-910, Sept. 1991.
[65] A.J. Offutt, "An Integrated Automatic Test Data Generation System," *J. Systems Integration,* vol. 1, no. 3, pp. 391-409, 1991.
[66] A.J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Software Testing, Verification and Reliability,* vol. 7, no. 3, pp. 165-192, 1997.
[67] R. Hierons, M. Harman, and S. Danicic, "Using Program Slicing to Assist in the Detection of Equivalent Mutants," *Software Testing, Verification and Reliability,* vol. 9, no. 4, pp. 233-262, 1999.
[68] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Trans. Software Eng.,* vol. 32, no. 8, pp. 608-624, Aug. 2006.
[69] M. Erwig, R. Abraham, S. Kollmansberger, and I. Cooperstein, "Gencel—A Program Generator for Correct Spreadsheets," *J. Functional Programming,* vol. 16, no. 3, pp. 293-325, 2006.
[70] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert, "Visual Specifications of Correct Spreadsheets," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing,* pp. 189-196, 2005.
[71] P. Saariluoma and J. Sajaniemi, "Transforming Verbal Descriptions into Mathematical Formulas in Spreadsheet Calculation," *Int'l J. Human-Computer Studies,* vol. 41, no. 6, pp. 915-948, 1994.
[72] T. Teo and M. Tan, "Quantitative and Qualitative Errors in Spreadsheet Development," *Proc. 30th Hawaii Int'l Conf. System Sciences,* pp. 25-38, 1997.
[73] T. Teo and M. Tan, "Spreadsheet Development and "What-if" Analysis: Quantitative versus Qualitative Errors," *Accounting Management and Information Technologies,* vol. 9, pp. 141-160, 2000.
[74] D. Mason and D. Keane, "Spreadsheet Modeling in Practice: Solution or Problem," *Interface,* pp. 82-84, 1989.
[75] T. Williams, "Spreadsheet Standards," technical report, 1987.
[76] M.G. Simkin, "How to Validate Spreadsheets," *J. Accountancy,* vol. 180, no. 11, pp. 130-138, 1987.
[77] A.J. Offutt, "Investigations of the Software Testing Coupling Effect," *ACM Trans. Software Eng. and Methodology,* vol. 1, no. 1, pp. 5-20, 1992.
[78] K.S. How Tai Wah, "A Theoretical Study of Fault Coupling," *Software Testing, Verification and Reliability,* vol. 10, no. 1, pp. 3-45, 2000.
[79] K.S. How Tai Wah, "An Analysis of the Coupling Effect: I. Single Test Data," *Science of Computer Programming,* vol. 48, nos. 2/3, pp. 119-161, 2003.
[80] M. Purser and D. Chadwick, "Does an Awareness of Differing Types of Spreadsheet Errors Aid End-Users in Identifying Spreadsheet Errors," *Proc. Symp. European Spreadsheet Risks Interest Group,* pp. 185-204, 2006.
[81] R. Abraham and M. Erwig, "Type Inference for Spreadsheets," *Proc. Eighth ACM-SIGPLAN Int'l Symp. Principles and Practice of Declarative Programming,* pp. 73-84, 2006.
[82] M. Clermont, C. Hanin, and R. Mittermeir, "A Spreadsheet Auditing Tool Evaluated in an Industrial Context," *Spreadsheet Risks, Audit and Development Methods,* vol. 3, pp. 35-46, 2002.
[83] M. Fisher II, G. Rothermel, D. Brown, M. Cao, C. Cook, and B. Burnett, "Integrating Automated Test Generation into the WYSIWYT Spreadsheet Testing Methodology," *ACM Trans. Software Eng. and Methodology,* vol. 15, no. 2, pp. 150-194, 2006.
[84] R. Abraham and M. Erwig, "Inferring Templates from Spreadsheets," *Proc. 28th IEEE Int'l Conf. Software Eng.,* pp. 182-191, 2006.
[85] J.R. Ruthruff, S. Prabhakararao, J. Reichwein, C. Cook, E. Creswick, and M.M. Burnett, "Interactive, Visual Fault Localization Support for End-User Programmers," *J. Visual Languages and Computing,* vol. 16, nos. 1/2, pp. 3-40, 2005.
[86] A. Phalgune, C. Kissinger, M. Burnett, C. Cook, L. Beckwith, and J. Ruthruff, "Garbage In, Garbage Out? An Empirical Look at Oracle Mistakes by End-User Programmers," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing,* pp. 45-52, 2005.

Robin Abraham received the BTech (with honors) degree from the Indian Institute of Technology, Kharagpur, and the MS and PhD degrees from Oregon State University. He is a program manager with the WinSE group at Microsoft. His research interests include software engineering, user software engineering, and programming language design. In the area of user software engineering, within the spreadsheet paradigm, he has worked on approaches for static checking, automatic test-case generation, mutation testing, automatic generation of spreadsheets from specifications, and debugging.

Martin Erwig received the MS degree in computer science from the University of Dortmund, Germany, in 1989 and the PhD and Habilitation degrees from the University of Hagen, Germany, in 1994 and 1999, respectively. He is an associate professor of computer science at Oregon State University. He is a member and a cofounder of the EUSES consortium, a multi-university research collaboration that is concerned with bringing the benefits of software engineering to users. He is a cofounder of a startup company that produces tools for detecting errors in spreadsheets. His research interests include functional programming, domain-specific languages, and visual languages. His current research focus is centered on questions of language design. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.