Test-Driven Goal-Directed Debugging in Spreadsheets

Robin Abraham
Microsoft Corporation
Robin.Abraham@microsoft.com

Martin Erwig
Oregon State University
erwig@eecs.oregonstate.edu

Abstract

We present an error-detection and -correction approach for spreadsheets that automatically generates questions about input/output pairs and, depending on the feedback given by the user, proposes changes to the spreadsheet that would correct detected errors. This approach combines and integrates previous work on automatic test-case generation and goal-directed debugging. We have implemented this method as an extension to MS Excel. We carried out an evaluation of the system using spreadsheets seeded with faults using mutation operators. The evaluation shows among other things that up to 93% of the first-order mutants and 98% of the second-order mutants were detected by the system using the automatically generated test cases.

1. Introduction

Activities related to testing and debugging of code take up a majority of programmer time and effort as shown by a study conducted in the U.S. by NIST. The study found that software engineers typically spend 70-80% of their time testing and debugging code; on an average, errors take 17.4 hours to find and fix [27].

Other studies have previously shown that spreadsheets are among the most widely used programming tools [26] and that spreadsheets contain many errors [23] that have considerably negative impact [11, 13].

In this context, a major part of spreadsheet research has focused on helping end users answer the following three questions.

(1) *Are there faults in the spreadsheet?*
The "What You See Is What You Test" (WYSIWYT) testing framework has been developed to help end users test their spreadsheets to expose faults [24]. Approaches aimed at automatic consistency checking of spreadsheet formulas have also been developed to help detect faults. Most of these systems require the users to annotate the spreadsheet with additional information, which is then used for consistency checking of formulas [6, 8–10]. In previous work, we have developed a system, called UCheck, that automatically infers labels within spreadsheets and uses this information to carry out consistency checking [3], thereby minimizing the effort required of the user.

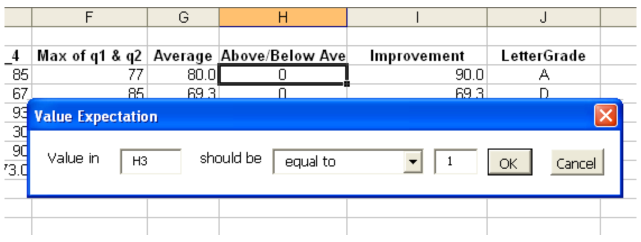
(2) *Where are the faults located?*
Many of the the systems that aim to help users detect faults also have fault localization mechanisms. In WYSIWYT, users can mark cell outputs as correct or incorrect using ✓ and ✗ marks. The system uses this input to generate feedback about the likelihood of faults in cells through cell shading—cells with higher fault likelihood are shaded darker than those with lower fault likelihood. The systems described in [3, 6, 8] also use similar shading schemes to draw the user's attention to cells with higher fault likelihood.

(3) *How can the faults be corrected?*
Even though techniques like spreadsheet audit and code inspection [18, 20, 25] help identify both the presence and location of faults in some cases, very few systems actually help the users correct the identified faults. In previous work, we have developed a new approach, called "goal-directed debugging", to help end users debug their spreadsheets [2]. This approach automatically generates change suggestions based on the user's expectations about the output of a cell. The automatic consistency checker, UCheck, has also been extended to generate change suggestions based on the inconsistencies detected by the system [5].

In the evaluation described in [2], we generated *first-order mutants*[1] from a set of spreadsheets using mutation operators [4], and studied how effective GoalDebug was at correcting the seeded faults. The evaluation helped us improve the system so as to make it effective at correcting almost all the seeded faults that were detected. However, about 47% of the seeded faults were not even detected since we only used one set of input values. Moreover, the impact of having more than one fault in the spreadsheet was not studied. In this paper, we describe the integration of GoalDebug with an automatic test case generation mechanism together with an evaluation of the extended system. The evaluation showed that one of the configurations of the new system detects 92.93% of the first-order mutants and 98.36% of the second-order mutants. The new version of the system was able to detect more mutants because of the additional test cases used. Since only one set of inputs were used, even though the old version of GoalDebug corrected 97.11% of the detected mutants, this figure only translates to 51.64% of the seeded faults. In comparison, the new version of the system corrects 88.45% of the first-order mutants and 97.81% of the second-order mutants.

We briefly describe GoalDebug, and the results from pre-

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



Value Expectation

Value in H3 should be equal to 1

Figure 1. Specifying value expectation

vious evaluations in the next section. The modifications done to the system to integrate the automatic test case generator AutoTest are briefly described in Section 3. The evaluation we carried out of the test-driven debugging framework is described in Section 4. We then describe related work in Section 5. We present conclusions and directions for future work in Section 6.

2. Goal-Directed Debugging

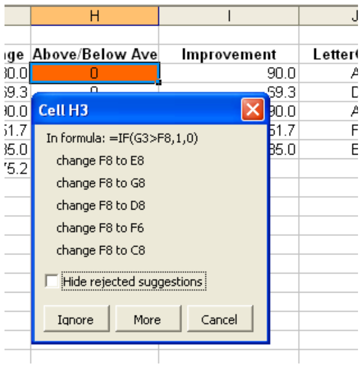
GoalDebug is an implementation of the goal-directed debugging approach for Microsoft Excel. Whenever a spreadsheet user observes a failure in their spreadsheets, GoalDebug allows them to mark the cell with the failure as incorrect, and specify their expectation for the output of that cell as illustrated in Figure 1. The system converts the user expectation to constraints and propagates these upstream in the data flow. The propagated constraints are then used to generate change suggestions, any one of which, if applied, would result in the output in the marked cell to meet the user expectation.

In general, the set of generated change suggestions is large. To minimize the effort required of the user, the system uses a set of heuristics to rank the generated change suggestions from most likely to least likely. The cell with the highest ranked change suggestion is shaded orange.

After the ranking has been carried out, the user can right-click in any cell and view the top five change suggestions that have been generated for that cell as shown in Figure 2. At this stage, the user can perform any one of the following actions:

1. Reject one or more of the presented suggestions. In such cases, the rejected suggestions are converted to constraints that are then used to refine the change-inference process.
2. Ask for more suggestions. In this case, the system displays more suggestions to the user.
3. Pick any one suggestion from the list of suggestions. The system then brings up a confirmation dialog which allows the users to make any changes to the suggestion before the changes are performed on the actual spreadsheet.

Since the cell with the top-ranked suggestion is shaded orange by the system, both the change-inference mechanism and ranking heuristics play important roles in fault localization. Moreover, the change-inference mechanism determines the classes of faults GoalDebug is useful against. Ideally, the system should be able to suggest changes that correct detected faults so that the user does not have to manually edit formulas.



Cell H3

In formula: =IF(G3>F8,1,0)

change F8 to E8

change F8 to G8

change F8 to D8

change F8 to F6

change F8 to C8

☐ Hide rejected suggestions

Figure 2. Change suggestions

In the evaluation described in [2], we ran mutation operators on the spreadsheets and generated first-order mutants by introducing faults in the cells that contain formulas. Using the values in the *data cells*[2] as input, we then compared the output from the generated mutants with the output from the original spreadsheets. For the cells in which the outputs from the original spreadsheet differed from the outputs from the mutant spreadsheet, we specified the outputs from the original spreadsheet as the expected values and ran GoalDebug to generate change suggestions. We had to discard those mutants in which the cell outputs were the same as those of the original sheet for the given single set of input values. We applied the generated change suggestions to the mutated spreadsheets to determine the number of cases in which the mutation is reversed by the change suggestions generated by GoalDebug and also recorded the rank of the correct change suggestions.

The evaluation led us to make improvements to the change-inference mechanism and the ranking heuristics. As a result of the improvements, GoalDebug works very well for first-order mutants. Specifically, the evaluation showed that the system generated suggestions that would correct 99.97% of the first-order mutants. Moreover, the evaluation also showed that in 59% of the cases, the top ranked suggestion corrects the mutation, and in 71% of the cases, the correct suggestion is ranked within the top two.

3. Test-Driven Debugging

In previous work with GoalDebug, we were primarily concerned with improving the effectiveness of the system with regard to the following two important aspects.

1. Generate change suggestions to recover from a wide variety of faults: This aspect is important since it increases the range of faults GoalDebug would be useful

[2]These are the cells that do not contain formulas.

against.

2. Ability to assign a high rank to the correct change suggestion: If the correct change suggestion is assigned the top rank, it will show up at the beginning of the list presented to the user. Therefore, a high rank can lower the amount of effort the user has to spend to locate the correct change suggestion.

In around 47% of the mutant spreadsheets in the evaluation described in [2], the seeded faults did not result in failures for the single set of inputs used. These mutants had to be excluded from the study since the output computed by the cell formula matched the expected output for the set of inputs used. This result indicates a very strong need to integrate a mechanism for exposing more faults with GoalDebug.

The primary user input to GoalDebug are the value expectations entered by the user in cells where failures are observed. In general, many test cases are required to expose faults in the cell formulas. As more and more faults get exposed by test cases, the user would be able to provide additional value expectations to the system. The value expectations, in turn, result in more constraints which GoalDebug can then exploit to refine change inference.

However, to an end-user programmer who does not have any formal training in software engineering, testing presents two challenges.

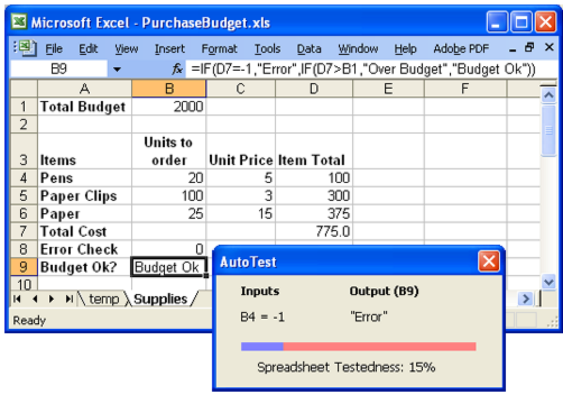
1. *When is a spreadsheet well tested?* To help with this aspect of testing, researchers have come up with test adequacy criteria, which allow a tester to decide when to stop testing.
2. *What test inputs will improve coverage?* When an end user is faced with the task of inventing new test inputs, it is not clear if a new test input increases coverage. Moreover, the task of inventing new test cases can be rather tedious.

In the next section, we briefly describe an automatic test-case generator we have previously developed. In Section 3.2 we describe how the test-case generator can be integrated with GoalDebug into a new test-driven debugging framework.

3.1 Generation of Test Inputs

The AutoTest system generates test suites that satisfy the *definition-use (du)* test adequacy criterion [1]. The idea behind the definition-use coverage criterion is to test for each variable (or cell in the case of spreadsheets) all of its uses (through references).

AutoTest generates test inputs employing a constraint-based approach. The system presents test inputs and the corresponding computed output from a cell formula to the user. The user can then specify if the computed output is right or wrong. AutoTest also gives the user feedback about the "level of testedness", that is, the du coverage of the spreadsheet, as shown in Figure 3.



B9 =IF(D7=-1,"Error",IF(D7>B1,"Over Budget","Budget Ok"))

	A	B	C	D	E	F
1	Total Budget	2000				
2						
3	Items	Units to order	Unit Price	Item Total		
4	Pens	20	5	100		
5	Paper Clips	100	3	300		
6	Paper	25	15	375		
7	Total Cost			775.0		
8	Error Check	0				
9	Budget Ok?	Budget Ok				
10						

Inputs	Output (B9)
B4 = -1	"Error"

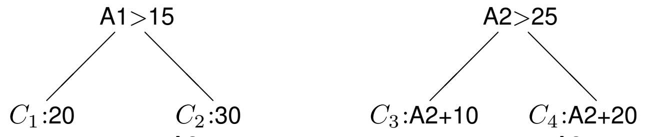
Spreadsheet Testedness: 15%

Figure 3. Testing with AutoTest

In the following we use a simple example to show how AutoTest works. A more detailed and formal description of the system can be found in [1]. Consider a spreadsheet that contains the following three cells.

A1: 10
A2: IF(A1>15,20,30)
A3: IF(A2>25,A2+10,A2+20)

Cell A1 is an input cell (since it does not contain a formula) and therefore has one definition. Figure 4(a) is the *constraint tree*[3] representation of the formula in cell A2. The formula in A2 has one use of A1 (which is always executed) in the condition. The two branches of the formula give A2 two definitions, which can be executed by satisfying the constraints $C_1 \equiv \text{A1} > 15$ and $C_2 \equiv \text{A1} \leq 15$ for the true and false branch, respectively.



(a) Formula in A2 (b) Formula in A3

Figure 4. Constraint trees

Similarly, the formula in A3 (constraint tree shown in Figure 4(b)) has three uses of A2—one (which is always executed) in the condition, and one in each of the true and false branches. The uses in the true and false branches can be executed by satisfying the constraints $C_3 \equiv \text{A2} > 25$ and $C_4 \equiv \text{A2} \leq 25$, respectively.

To test the formula in A3 enough to satisfy the du-adequacy criterion, we need to execute the two definitions of A2 and the two uses of A2 in the two branches of the formula in A3—a total of 4 du pairs. The constraints that must be satisfied to execute the 4 du pairs are shown below together with the cell definitions that will be executed when

[3] A constraint tree is the internal representation of formulas used by AutoTest in which conditions are stored in internal nodes and condition-free subformulas are stored in the leaves.

the constraints are fulfilled.

$$\{C_1\text{:A2} = 20, C_3\text{:A3} = \text{A2+10}\}$$
$$\{C_1\text{:A2} = 20, C_4\text{:A3} = \text{A2+20}\}$$
$$\{C_2\text{:A2} = 30, C_3\text{:A3} = \text{A2+10}\}$$
$$\{C_2\text{:A2} = 30, C_4\text{:A3} = \text{A2+20}\}$$

In the above, we use the notation C_3:A3 = A2+10 to indicate that satisfying constraint C_3 causes A3 to execute branch A2+10.

In some cases, it might not be possible to solve the constraints for a du pair. For example, the constraints $\{C_1\text{:A2} = 20, C_3\text{:A3} = \text{A2+10}\}$ cannot be solved since C_3 requires A2 to be some value greater than 25. However, satisfying C_1 results in 20 in A2. In such cases, it is not possible to generate a test case that executes the du pair under consideration. Such du pairs are said to be *infeasible* and cannot be exercised.

In all other cases where the constraints can be solved, the solution gives test inputs that exercise the du pair under consideration. For example, $\{\text{A1} = 16, \text{A2} = 20\}$ is a solution for the constraints $\{C_1\text{:A2} = 20, C_4\text{:A3} = \text{A2+20}\}$. Therefore, changing the value in A1 to 16 exercises this du pair.

3.2 Debugging With Change Suggestions

To improve fault detection, we integrated GoalDebug with AutoTest. We envisage the combination of the two systems working as follows.

1. AutoTest generates the test inputs for a formula.
2. The user examines the computed output from the cell formulas to determine if it is correct. If the output is correct, the user "approves" it and the input values and computed output are stored as a test case.
3. When failures are observed, the user marks the cell as incorrect and specifies the expected output. The input values and expected output are stored as a test case, and the user's expectation is converted to a constraint and stored by GoalDebug. The constraints are used by GoalDebug to generate change suggestions for spreadsheet formulas.
4. For each cell in the spreadsheet, the system maintains sets of test cases, constraints, and change suggestions. The user can view these and apply any of the suggested changes they deem correct.

The debugging/change-inference component of the new system has a couple of significant enhancements over the old one. The previous versions of GoalDebug only allowed the user to specify one value expectation for each cell. This expectation was assumed to be on the basis of the values in the input cells. Within the new framework, users can specify multiple value expectations for each cell. Each value expectation is associated with the inputs of a particular test case.

In the previous versions, whenever a user marks the output of a cell as incorrect and specifies the expected output, it leads to change suggestions being generated for the marked cell and all other cells, which contain formulas, upstream from the marked cell. This approach was adopted since fault localization was based on user feedback provided on the basis of a single set of test inputs. In the absence of additional information about the correctness of the formulas upstream from the marked cell, the system generates change suggestions for all of them and ranks those suggestions lower than the changes generated for the marked cell.

In the new framework, it is assumed that the use of additional test cases improves the capability of the system to expose more faults. Therefore, the expectation specified for a cell formula is not used directly to generate change suggestions for formulas upstream from the marked cell. Instead, it is only used to refine the change suggestions generated by expectations specified directly within those cells by the user. This modification prunes the set of generated change suggestions, and is based on the assumption that additional test cases increase confidence in the correctness of the unmarked cells.

4. Evaluation

After integrating GoalDebug with the testing framework of AutoTest as described in the previous section, we evaluated the system to answer the following research questions.

RQ1: *Is goal-directed debugging effective at correcting faults exposed by additional test cases?*
The evaluation described in [2] showed that GoalDebug is very effective against first-order mutants that manifest as failures for the single set of input values originally present in the spreadsheet. However, quite a few of the mutants went undetected since the computed output matched the expected output for the particular set of input values used. We expect additional test cases to expose more faults. We need to investigate how effective GoalDebug is at correcting these faults.

RQ2: *Is change inference effective against higher-order mutants?*
Higher-order mutants are interesting since they more closely reflect real-world situations. Moreover, faults in a few cells usually manifest as failures in many more cells, thereby making change inference more challenging.

4.1 Setup

An overview of the evaluation setup is shown in Figure 5. For the purpose of the evaluation, we use spreadsheets that have been used in previous empirical studies [1, 2, 4]. The spreadsheets have been picked to include as many different kinds of formulas, and formulas with branching, as possible, in the evaluation. Information about the spreadsheets is given in Table 2. For each spreadsheet used in the evaluation, the table contains information about the number of cells with formulas (Fml) and the total number of cells (Total). The spreadsheets have been picked to include as many different kinds of formulas as possible in the evaluation.

We generate mutant spreadsheets by seeding faults in the

Table 1. Mutation operators for spreadsheets

Operator	Description
ABS	*ABS*olute value insertion
AOR	*A*rithmetic *O*perator *R*eplacement
CRP	*C*onstants *ReP*lacement
CRR	*C*onstants for *R*eference *R*eplacement
LCR	*L*ogical *C*onnector *R*eplacement
ROR	*R*elational *O*perator *R*eplacement
RCR	*R*eference for *C*onstant *R*eplacement
FDL	*F*ormula *DeL*etion
FRC	*F*ormula *R*eplacement with *C*onstant
RFR	*ReF*erence *R*eplacement
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

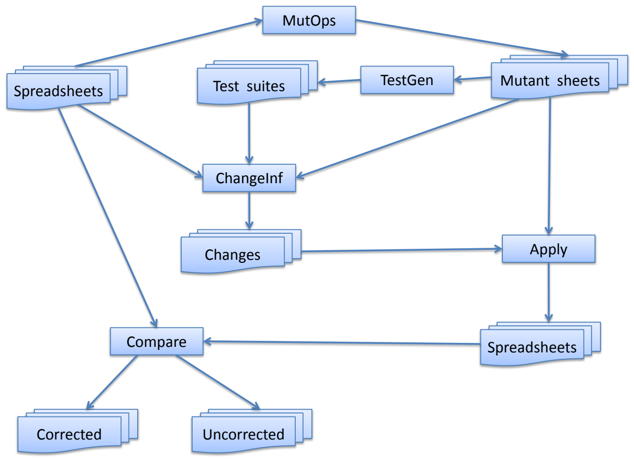


Figure 5. Evaluation setup

Table 2. Sheet details

Sheet	Cells		Sheet	Cells	
	Fml	Total		Fml	Total
Microgen	2	12	GradesNew	8	26
FitMachine	6	18	Digits	6	14
NetPay	6	18	Purchase	15	50
RandJury	21	58	Sales	16	29
Solution	3	12	Budget	6	24
MBTI	28	83	NewClock	10	24
GradesBig	21	48	Harvest	9	26
Payroll	54	100			

original spreadsheets using the mutation operators given in Table 1. The mutation operators have been designed to reflect errors reported in spreadsheet literature, and they are also based off of mutation operators developed for general-purpose programming languages. We did not use the FDL (formula-deletion) and FRC (formula replace with constant) since it is unrealistic to expect the system magically to suggest a formula for an empty cell or a cell with constant.

Test inputs that satisfy the du-adequacy criterion are then generated for each of the mutant spreadsheets using AutoTest. The outputs to these test input values from the original spreadsheets is assumed to be the expected/correct output. In cases where the output from the original spreadsheets differ from those computed by the mutant spreadsheets, the values from the original spreadsheets are treated as the user expectations. These are then used by the change-inference mechanism of GoalDebug to generate change suggestions. The generated change suggestions are applied to the mutant spreadsheets. These sheets are then compared with the original spreadsheets to determine if the change suggestions have been successful at correcting the seeded faults.

We evaluated the performance of the framework under the following three configurations:

1. GoalDebug only: This configuration was used in the evaluation described in [2]. Basically, it involves testing each of the spreadsheet formulas with a single set of input values, and triggering change suggestions from the cells in which failures are observed.
2. Test-driven debugging (F): In this configuration, the spreadsheet is tested with a test suite that satisfies 100% du-adequacy. The test suite is generated from the first solvable constraint set for each feasible du pair.
3. Test-driven debugging (R): In this configuration, AutoTest picks a solvable constraint set at random from those available for each feasible du pair. As in the previous configuration, the spreadsheet is tested with a test suite that satisfies 100% du adequacy in this case as well.

For each of the configurations, we collected the following information.

1. The number of generated *irreversible mutants* (Irrev.).

These mutant formulas evaluate to the same value as the original formula for the test inputs and thus cannot produce failures that could be identified by the user. These faults are therefore undetected, and GoalDebug is inapplicable in these cases since the computed output agrees with the expected output.

2. The number of generated *reversible mutants* (Rev.). These mutant formulas manifest as failures for at least one set of test inputs. GoalDebug can therefore be invoked on those cells.
3. The number of reversible mutants that were *corrected* (Corr.). The seeded fault in these formulas were corrected by one of the change suggestions generated by GoalDebug. This group of mutants were the ones GoalDebug was effective against.

4.2 Results

The data collected for various configurations of GoalDebug for first-order mutants is shown in Table 3. In the "GoalDebug only" configuration, only 53.17% of the first-order mutants were detected. 97.11% of the detected mutants were corrected by GoalDebug. The percentage of detected mutants went up to 80.55% under the "Test-driven debugging (F)" configuration. 94.37% of these detected mutants were corrected by the system. Under the "Test-driven debugging (R)" configuration, 92.97% of the first-order mutants that were detected by the system, out of which, 95.14% were corrected by GoalDebug.

It is not surprising that additional test cases expose more mutants. Under the "Test-driven debugging (F)" configuration, the du-adequate test suite is generated by solving the first constraint for each feasible du pair, whereas, under the "Test-driven debugging (R)" configuration, the du-adequate test suite is generated from a constraint set picked at random from those available for a du pair. We have selected the random strategy because we have observed from the empirical evaluation of the mutation adequacy of du-adequate test suites described in [4] that random selection of constraints detects more mutants than selecting the first solvable constraint for test case generation.

The performance of the test-driven debugging framework under different configurations for spreadsheets seeded with two faults each is shown in Table 4. As can be seen from the table, the number of mutants generated is considerably higher in this case. In the "Test-driven debugging (F)" configuration, the system detects 82.41% of the generated mutants, out of which 99.35% are corrected by the generated change suggestions. Along similar lines to the case of first-order mutants, the system in the "Test-driven debugging (R)" configuration detects 98.36% of the generated mutants, out of which 99.44% are corrected by the changes generated by the system.

4.3 Discussion

RQ1: As can be seen from the data collected from the evaluation, the additional test cases dramatically improve the number of faults detected and corrected by the system. Further analyses of the spreadsheets and surviving mutants showed that the benefit of having additional test cases is more obvious in spreadsheets whose formulas have branching. This result is as expected since more targeted testing is required to expose faults in branches of formulas.

RQ2: In addition to the high percentage of second-order mutants detected by the new system, we also see that 97.81% of these detected faults are corrected by the change suggestions generated by the system. Therefore, the change-inference mechanism of GoalDebug is effective against second-order mutants.

The evaluation has shown that integration of the testing framework with GoalDebug has huge potential benefits, both, at exposing more faults, and at generating effective change suggestions to correct them.

5. Related Work

The WYSIWYT framework has an automatic test case generator called "Help Me Test" (HMT) to help end users develop effective test cases [14]. The framework is geared towards fault detection through testing. Empirical studies have shown that users are able to detect and, in many cases, correct seeded faults in spreadsheets using these systems [22]. However, the WYSIWYT framework itself does not have any mechanism to help users correct detected faults. It would be beneficial to integrate the approach described in this paper within the WYSIWYT framework. When the users observe faults within the spreadsheets, they can place ✗ marks in the cells and specify the expected output. In these cases, the expected output could be converted to constraints for the given set of inputs. In cells where the output matches the users' expectations, they can place ✓ marks. In these cases, the computed output in the cell could be converted to a constraint for the given set of inputs. The ✓ and ✗ marks would help the fault localization mechanism of WYSIWYT, and the generated constraints would be inputs to GoalDebug for change inference.

The WHYLINE system uses static and dynamic analyses of programs developed in the Alice environment to help users isolate faults [17]. The system allows users to ask "Why...?" and "Why didn't...?" questions to express their expectations about program behavior. The approach is similar to GoalDebug except that it does not generate change suggestions. Empirical studies have shown that WHYLINE helps users debug errors up to 8 times faster in the Alice environment.

Other approaches and techniques, like code inspection [20], auditing [18, 25], and adoption of good spreadsheet design practices [16,23,28], from traditional software engineering have also been studied to minimize the occurrence of faults in spreadsheets. None of these approaches guarantee correctness. Moreover, they do not help the user debug faults either.

The idea of mutation testing for general-purpose programming languages was proposed in [12, 15]. The suite of

Table 3. Fault detection under various GoalDebug configurations (single fault)

Sheet	Total Mutants	GoalDebug only			Test-driven debugging (F)			Test-driven debugging (R)		
		Irrev.	Rev.	Corr.	Irrev.	Rev.	Corr.	Irrev.	Rev.	Corr.
Microgen	176	143	33	33	123	53	42	19	157	144
GradesNew	338	157	181	181	36	302	295	36	302	295
FitMachine	440	366	74	72	43	397	297	14	426	350
Digits	465	172	293	293	13	452	379	13	452	379
NetPay	108	61	47	47	0	108	97	0	108	97
Purchase	325	172	153	153	7	318	289	0	325	302
RandJury	886	578	308	308	506	380	362	0	886	864
Sales	338	0	338	338	0	338	338	0	338	338
Solution	235	119	116	116	0	235	227	0	235	227
Budget	158	46	112	112	41	117	114	11	147	140
MBTI	1145	902	243	243	405	740	717	270	875	844
NewClock	321	156	165	164	61	260	251	44	277	263
GradesBig	930	283	647	545	195	735	697	95	835	791
Harvest	231	10	221	211	10	221	211	10	221	211
Payroll	1404	347	1057	1057	19	1385	1385	15	1389	1389
Total	7500	3512	3988	3873	1459	6041	5701	527	6973	6634

mutation operators used in the evaluation described in this paper was proposed by us in [4]. They have been designed to reflect errors reported in spreadsheet literature [7, 21] and also include mutation operators originally proposed for general-purpose programming languages [19].

6. Conclusions and Future Work

We have demonstrated that automatic test-case generation can be successfully integrated into a spreadsheet debugger to significantly increase the number of faults that can be detected. Whereas the old version of the system could only detect 53.17% of the seeded faults, the new version detects up to 92.97% of the first-order mutants (and 98.36% of second-order mutants). The improvement in fault-detection capability has had a corresponding improvement on fault-correction capabilities of the system. The old version of the system could only correct 51.64% of the seeded faults, whereas, the new version of the system corrects 88.45% of the first-order mutants and 97.81% of the second-order mutants.

Test cases have two roles in the framework we have described in this paper. Firstly, they have the "traditional" role of helping to identify faults within the spreadsheet. Secondly, the test cases—especially the expected output component of the test cases—provide GoalDebug additional constraints that help refine the change suggestions. Therefore it is important to determine which test cases provide maximum benefit to help with GoalDebug so that this information can be used for test selection. In future work, we plan to study this aspect as a test-selection criterion.

In the evaluation described in this paper, the user feedback has been simulated and is assumed to be accurate. In future work, we would like to investigate the impact of errors in user feedback on goal-directed debugging. User errors can occur at various stages in the process and their impact needs to be explored. For example, users might make mistakes when specifying test cases, or while specifying value expectations on cell output. Moreover, expectations specified upstream or at sinks might be more accurate than those specified at intermediate points, especially those that are further downstream. Since data about these error frequencies is not available, we would need to carry out user studies to evaluate their impact. User studies would also be helpful in determining if actual end users can use the different components of the framework effectively.

References

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

[2] R. Abraham and M. Erwig. GoalDebug: A Spreadsheet Debugger for End Users. *29th IEEE Int. Conf. on Software Engineering*, pp. 251–260, 2007.

[3] R. Abraham and M. Erwig. UCheck: A Spreadsheet Unit Checker for End Users. *Journal of Visual Languages and Computing*, 18(1):71–95, 2007.

[4] R. Abraham and M. Erwig. Mutation Operators for Spreadsheets. *IEEE Trans. on Software Engineering*, 2008. To appear.

[5] R. Abraham, M. Erwig, and S. Andrew. A Type System Based on End-User Vocabulary. *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pp. 215–222, 2007.

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

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

Table 4. Fault detection under various GoalDebug configurations (two faults)

Sheet	Total Mutants	Test-driven debugging (F) Irrev.	Rev.	Corr.	Test-driven debugging (R) Irrev.	Rev.	Corr.
Microgen	7303	3626	3677	3251	15	7288	7145
GradesNew	48350	0	48350	48157	0	48350	48157
FitMachine	75949	36508	39441	36249	3138	72811	70221
Digits	85849	5612	80237	79513	72	85777	79513
NetPay	4504	0	4504	4479	0	4504	4479
Purchase	48575	544	48031	47501	320	48255	47881
RandJury	370078	240744	129334	129110	166	369912	369821
Sales	53147	171	52976	52976	171	52976	52976
Solution	18403	8	18395	18117	8	18395	18117
Budget	9327	590	8737	8711	16	9311	9291
MBTI	631681	119388	512293	511013	2655	629026	628854
NewClock	44665	4644	40021	39914	183	44482	44336
GradesBig	383161	53190	329971	322451	27400	355761	351128
Harvest	15932	715	15217	15009	715	15217	15009
Payroll	948221	17200	931021	931021	10030	938191	938191
Total	2745145	482940	2262205	2247472	44889	2700256	2685119

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

[9] M. M. Burnett, C. Cook, J. Summet, G. Rothermel, and C. Wallace. End-User Software Engineering with Assertions. *25thIEEE Int. Conf. on Software Engineering*, pp. 93–103, 2003.

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

[11] G. J. Croll. The Importance and Criticality of Spreadsheets in the City of London. *Symp. of European Spreadsheet Risks Interest Group (EuSpRIG)*, 2005.

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help For the Practicing Programmer. *IEEE Computer*, 11(4):34–41, 1978.

[13] EuSpRIG. European Spreadsheet Risks Interest Group. `http://www.eusprig.org/`.

[14] 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. on Software Engineering and Methodology*, 15:150–194, 2006.

[15] R. G. Hamlet. Testing Programs With the Aid of a Compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

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

[17] A. J. Ko and B. A. Myers. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. *Int. Conf. on Human Factors in Computing Systems*, pp. 151–158, 2004.

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

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

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

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

[22] 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. *IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, pp. 203–210, 2003.

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

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

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

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

[27] G. Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology*, RTI Project Number 7007.011, 2002.

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