AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets*

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

Abstract

In this paper we present a system that helps users test their spreadsheets using automatically generated test cases. The system generates the test cases by backward propagation and solution of constraints on cell values. These constraints are obtained from the formula of the cell that is being tested when we try to execute all feasible DU associations within the formula. AutoTest generates test cases that execute all feasible DU pairs. If infeasible DU associations are present in the spreadsheet, the system is capable of detecting and reporting all of these to the user. We also present a comparative evaluation of our approach against the "Help Me Test" mechanism in Forms/3 and show that our approach is faster and produces test suites that give better DU coverage.

1 Introduction

Studies have shown that there is a high incidence of errors in spreadsheets [10], up to 90% in some cases [20]. These errors oftentimes lead to companies and institutions losing millions of dollars [23, 24, 14]. A recent study has also shown that spreadsheets are among the most widely used programming systems [22]. To carry out testing in commercially available spreadsheet systems like Microsoft Excel, users are forced to proceed in an ad hoc manner because tool support for testing is not available.[1] In particular, the lack of tools leaves users without information about how much of their spreadsheet has been tested. In this situation, users come away with a very high level of confidence about the correctness of their spreadsheets even when, in reality, their spreadsheets have many non-trivial errors in them [18]. This situation is highly problematic because users have spreadsheets with potentially lots of errors in them, but they are not aware of it.

Given the dilemma that testing does not guarantee the correctness of a program, how does a practitioner go about testing a program? Researchers have come up with test adequacy criteria which allow the tester to decide when to stop testing. Test adequacy criteria have different levels of confidence about absence of faults in the program being tested. One often employed criterion is *DU adequacy*, which requires that each possible path from any definition to all its uses is covered by a test case. The relative effectiveness of this and other criteria at fault detection have been compared in [26, 16]. In addition to monitoring the coverage of the test cases, a user is faced with the problem of inventing new test cases, which is generally tedious and prone to errors. In addition, it might not always be immediately clear to the user whether a new test case really improves the coverage. This is where an automatic tool for generating test cases comes into play: The user only has to inspect suggested test cases and approve or reject them. Generation and monitoring of coverage is reliably and automatically handled by the system.

Regarding DU coverage, we can observe that in the case of a single spreadsheet cell, different paths that require different test cases can principally result only from IF expressions in that cell's formula. In general, through nested IF expressions, each cell gives rise to a tree of subexpressions that need different test cases to be executed. The method that underlies our spreadsheet testing tool AutoTest is based on representing expressions as trees in which internal nodes carry conditions of IF expressions, and leaves of the tree carry arbitrary expressions. The edges of the tree are labeled *T* or *F* leading to the expressions of the "then" and "else" branches. From such an expression tree we can generate in several steps constraints that are solved to yield test cases to cover all expressions in the tree.

In the next section, we describe related work. In Section 3 we describe the scenario of an end user working with a spreadsheet, faced with the problem of testing it. In Section 4, we describe formally what it means to test a spreadsheet and what constitutes spreadsheet test cases. The notion of *DU coverage* as a test adequacy criteria is presented in Section 5, and in Section 6 we describe how AutoTest generates DU adequate test cases. A comparative evaluation of AutoTest against the "Help Me Test" (HMT) [13] test case generation system of Forms/3 [7] is described in Section 7. We present conclusions and plans for future work in Section 8.

2 Related Work

In earlier work we have developed the systems described in [11, 12] that allow the end users to create specifications of their spreadsheets and then use the specifications to generate spreadsheets that are provably free from refer-

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

[1]The WYSIWYT methodology (to be explained later) has been implemented for the Forms/3 spreadsheet system and is currently being ported to Excel.

ence, range, and type errors. The system described in [3] enables users to extract the specifications (also called *templates*) from their spreadsheets so they can adopt and work within the safety of these specification-based approaches.

The systems described in [5, 6] allow the user to carry out consistency checking of spreadsheet formulas on the basis of user annotations, or on the basis of automatically inferred headers [1], and flag the inconsistent formulas as potential faults. Consistency checking can also be carried out using assertions on the range of values allowed in spreadsheet cells [8].

Approaches as the ones described usually require additional effort from the user. For example, in order to be able to use the specification-based approach to the generation of safe spreadsheets [12], the user has to learn the specification language [4]. Sometimes these systems have only limited expressiveness. On the other hand, static analysis techniques cannot find all faults. The systems described above that do consistency checking of the spreadsheets do so without any information about the specifications from which the spreadsheet was created. As a result of this shortcoming we could have spreadsheets that would pass the consistency check and still not be correct with respect to the specifications.

As an alternative to static analysis and program generation techniques, *testing* has been used as a means for identifying faults in spreadsheets and thus improving the correctness of programs by removing the faults. Much effort in the area of testing has focused on automating it because of the high costs involved in testing. Effort invested in automating testing pays off in the long run when the user needs to test programs after modifications. This aspect makes a strong case in favor of systematically building test suites, based on some coverage criterion, that can be run in as little time as possible, resulting in thorough testing of the program. The "What You See Is What You Test" (WYSIWYT) methodology for testing spreadsheets [21] allows users to test their spreadsheets to achieve DU adequacy. WYSIWYT has been developed for the Forms/3 spreadsheet language [7] and gives the user feedback of the overall level of testedness of the spreadsheet by means of a progress bar. "Help Me Test" (HMT) is a component of the Forms/3 engine that does automatic test case generation to help the users minimize the cost of testing their spreadsheets [13]. Automatic test case generation has also been studied for general-purpose programming languages [9, 15]. The WYSIWYT methodology has been evaluated within the Forms/3 environment and found to be quite helpful in detecting faults in end-user spreadsheets [19].

Detecting faults is only the first step in correcting a spreadsheet. Fixing incorrect formulas is generally required to remove faults. The spreadsheet debugger described in [2] exploits the end users' understanding of their problem domain and expectations on values computed by the spreadsheet. The system allows the users to mark cells with incorrect output and specify their expected output. The system then generates a list of change suggestions that would result in the expected output being computed in the marked cell. The user can simply pick from the list of automatically generated change suggestions, thereby minimizing the number of formula edits they have to perform manually.

3 A Scenario

Nancy is the office manager of a small-sized firm and has developed the spreadsheet shown in Figure 1 to keep track of the office supplies.[2] The amount in B1 (2000 in this case) is the budget allowed for the purchase of office supplies. Rows 4, 5, and 6 store information about the different items that need to be purchased. B4 has the number of pens that need to be ordered, C4 has the cost per pen, and the formula in D4 computes the product of the numbers in B4 and C4 to calculate the proposed expenditure on the purchase of pens. Similarly, rows 5 and 6 keep track of the proposed expenses for paper clips and paper, respectively. The formula in B8 checks to ensure that none of the numbers in B4, B5, or B6 is less than zero. If one or more of the numbers are less than 0, the cell output is 1 to flag the error. Otherwise, the cell output is 0. Cell D7 contains the formula IF(B8=1,-1,D4+D5+D6), which computes the total cost across the three items if the error flag in B8 is set to 0. If the error flag in B8 is set to 1 the formula results in -1. The formula in B9 checks if the total proposed expenditure is within the maximum allowed budget for office supplies.

Microsoft Excel - PurchaseBudget.xls

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						

Figure 1. Office supplies spreadsheet

After creating the spreadsheet, Nancy goes through the formula cells, one at a time, to ensure that the formulas look correct to the best of her knowledge.[3] She then uses historical data from the previous month as input to verify if the spreadsheet output matches the actual expenses incurred. Once this verification is done, Nancy is confident about the correctness of her spreadsheet and starts using it for planning the office expenses. Overconfidence in the correctness

[2]The office budget spreadsheet shown in Figure 1 was among the spreadsheets used in the evaluation described in Section 7.

[3]Code inspection of spreadsheet formulas done by individuals working alone has been shown to detect 63% of errors, and group code inspection has up to 83% success rate at detecting errors [17].

of her spreadsheet might even keep her from doing the cursory "testing" the next time she modifies the spreadsheet.

From a software engineering perspective, the single set of test inputs Nancy used would not qualify as *adequate* testing of the spreadsheet. Even if she uses historical data from a few more months, she might not necessarily gain coverage since the inputs might only cause the execution of the same parts of the spreadsheet program. Given the nonexistent support for testing in Microsoft Excel, Nancy would have to come up with the test cases on her own without knowing if the new tests were actually resulting in more thorough testing of her spreadsheet. Moreover, with no feedback on meeting any test adequacy criteria, she also would have no idea of when she can consider her spreadsheet well tested.

Using the AutoTest system, Nancy can simply right-click on the cell whose formula she wants to test and pick the option "Test formula" from the popup menu. Assuming Nancy asks AutoTest to test the formula in cell B9, the system generates a set of *candidate test cases* for the formula in the cell and presents it to Nancy as shown in Figure 2. A candidate test case is defined as the set of inputs generated by the system, together with the corresponding output computed by the formula that is to be tested.

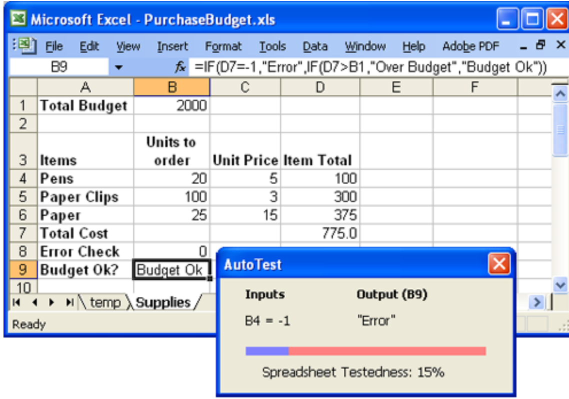


Figure 2. Automatically generated test cases for B9

AutoTest allows the user to do any one of the following three things to a candidate test case.

1. Users can *validate* generated test cases, thereby indicating that the computed output matches the expected output for the formula given the generated input values. Once a user validates a test case, it is moved to the test suite, and it is displayed on the interface in green-colored font.
2. Users can *flag* generated test cases to indicate that the computed output value is incorrect given the generated inputs. This action implies the formula is faulty since it is computing the wrong result. A flagged test case is displayed in red-colored font on the interface, and the cell with the formula that is being tested is also shaded red. The user can inspect the formula within the cell and make changes to correct it since testing detected a failure. Once the formula has been modified, the user can revisit the corrected test case to ensure the computed output matches the expected output for the cell, and then validate the test case.
3. They can also ignore generated test cases if they are unable to decide if the computed output is right or wrong. The users can come back to them at any later point during the course of testing.

For every candidate test case that Nancy approves, the updated progress bar shows how well tested the spreadsheet program is. Internally, the system uses DU adequacy (described in Section 5) to compute the level of testedness. AutoTest saves Nancy the effort of coming up with test cases by automatically generating test cases aimed at achieving 100% DU adequacy. The automatic generation of effective *test suites* lowers the cost of testing by reducing and directing the effort invested by the user. Moreover, the progress bar is an accurate indicator of the testedness of the spreadsheet and lets the user know when the spreadsheet has been thoroughly tested.

4 Spreadsheet Programs and Test Cases

A spreadsheet is a partial function $S : A \to F$ mapping cell addresses to formulas (and values). An element $(a, f) \in S$ is called a *cell*. Cell addresses are taken from the set $A = \mathbb{N} \times \mathbb{N}$, and 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.

$$f \in F \quad ::= \quad v \mid a \mid \psi(f, \dots, f)$$

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

The function $\sigma : F \to 2^A$ that computes for a formula the addresses of the cells it references is defined as follows.

$$\begin{aligned} \sigma(v) &= \varnothing \\ \sigma(a) &= \{a\} \\ \sigma(\psi(f_1, \dots, f_k)) &= \sigma(f_1) \cup \ldots \cup \sigma(f_k) \end{aligned}$$

A set of addresses $s \subseteq A$ is called a *shape*. We call $\sigma(f)$ the *shape* of f. The function σ can be naturally extended to work on cells and cell addresses by $\sigma(a, f) = \sigma(f)$ and $\sigma(a) = \sigma(S(a))$, that is, for a given spreadsheet S, $\sigma(a)$ gives the shape of the formula stored in cell a.

Related is the function $\sigma_S^* : S \times F \to 2^A$ that transitively chases references to determine all the input cells for a formula. The definition of σ_S^* is identical to that of σ, except for the following case:

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

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 observe that each spreadsheet contains a program together with the corresponding input. More precisely, the *program part* of a spreadsheet S is given by all of its cells that contain (non-trivial) formulas, that is, $P_S = \{(a,f) \in S \mid \sigma(f) \neq \varnothing\}$. This definition ignores formulas like $2+3$ and does not regard them as part of the spreadsheet program, because they always evaluate to the same result and can be effectively replaced by a constant. Correspondingly, the *input* of a spreadsheet S is given by all of its cells containing values (and locally evaluable formulas), that is, $I_S = \{(a,f) \in S \mid \sigma(f) = \varnothing\}$. Note that with these two definitions we have $S = P_S \cup I_S$ and $P_S \cap I_S = \varnothing$. Without loss of generalization we can assume from now on that all input cells are of the form (a,v).

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

Now we can define that a formula f *passes* a test $t = (I,v)$ if $[\![f]\!]_I = v$. Otherwise, f *fails* the test t. Likewise, we say that a cell (a,f) passes (fails) t if f passes (fails) t.

5 Definition-Use Coverage

The idea behind the DU coverage criterion is to test for each definition of a variable (or cell in the case of spreadsheets) all of its uses. In other words, test all DU pairs. In a spreadsheet every cell defines a value. In fact, cells with conditionals generally give rise two or more definitions, contained in the different branches. Likewise, one cell may contain different uses of a cell definition in different branches of conditionals. Therefore, definitions and uses cannot simply be represented by cell addresses. Instead, we generally need paths to subformulas to identify definitions and uses.

5.1 Expression and Constraint Trees

To formalize the notions of definitions and uses we employ an abstract tree representation of formulas that stores conditions of conditionals in internal nodes and conditional-free formulas in leaves. We can construct such a representation through two simple transformations of formulas. First, we lift all conditionals out of subformulas (that are not conditionals) so that the formula has the form of a nested conditional. This transformation can be achieved by repeatedly applying the following semantics-preserving rewrite rule to conditionals that are subformulas.

$$\psi(\ldots,\mathsf{IF}(c,f_1,f_2),\ldots) \rightsquigarrow \mathsf{IF}(c,\psi(\ldots,f_1,\ldots),\psi(\ldots,f_2,\ldots))$$

Note that the rewrite rule is only applied when $\psi \neq \mathsf{IF}$.

In a second step, we transform a lifted formula into its corresponding *expression tree* (see also Figure 3(a)) using the function $\mathcal{T}$, which creates for each conditional an internal node labeled with the condition and two subtrees for the two branches. The edges to the branches are labeled T and F to indicate which subtree corresponds to the "then" and "else" branch of the conditional.

$$\begin{array}{lcl} \mathcal{T}(\mathsf{IF}(c,f_1,f_2)) & = & c \text{ with edges } T \to \mathcal{T}(f_1),\ F \to \mathcal{T}(f_2) \\ \mathcal{T}(f) & = & f \end{array}$$

The second case leaves all non-conditional formulas unchanged.

Each condition c stored in an internal node of an expression tree can be transformed into two constraints γ^T and γ^F that guarantee that c evaluates to true or false, respectively. These constraints will replace the edge labels T and F in the expression tree. Constraints have the following form

$$\begin{array}{lcl} \gamma & ::= & f\,\omega\,v \mid \gamma \wedge \gamma \mid \gamma \vee \gamma \\ \omega & ::= & < \mid \leq \mid = \mid \geq \mid > \end{array}$$

For example, a condition $\mathsf{B3} > 4$ will be transformed into the two constraints $\mathsf{B3} > 4$ and $\mathsf{B3} \leq 4$, which will replace the labels T and F, respectively, in the expression tree.

A traversal of the whole expression tree that transforms conditions in internal nodes into constraints that are attached to the outgoing edges produces a *factored constraint tree*, shown in Figure 3(b). The constraints along each path from the root to a condition c in an internal node or an expression e in a leaf characterize the conditions under which the original formula would evaluate the c and e, respectively.

In a final traversal[4] we can collect all the constraints along each path and attach the resulting conditions to the leaf expressions, which results in a *constraint tree*, shown in Figure 3(c). For a condition or expression to be executed, the constraint attached to it has to be satisfied. For example, for expression e_1 to be executed, we need both, the constraints γ_1^T and γ_2^T, to be satisfied. That is why the leaf for e_1 has been annotated with $\gamma_1^T \wedge \gamma_2^T$ in Figure 3(c).

5.2 DU Pairs

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

[4] In an implementation, both traversals can be combined into one.

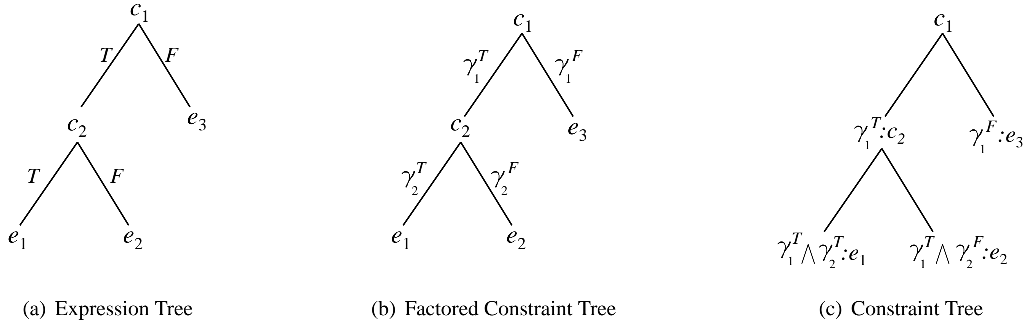


Figure 3. Stages of test-case generation

defining a cell may refer to other cells defined by conditionals, a single path is generally not sufficient to describe a definition. Instead, a definition is given be a set of paths.

These observations lead to the following definitions. Let $\mathcal{C}(a)$ be the constraint tree obtained from the expression $\mathcal{T}(S(a))$ as described above. We define the *uses* of a as the set $U_S(a)$, which contains the nodes of all trees $\mathcal{C}(a')$ for which $a \in \sigma(S(a'))$. Correspondingly, the *immediate definitions* of a are given by the the leaves of $\mathcal{C}(a)$. We refer to this set as $D_S^0(a)$. To obtain the complete set of definitions for a we have to combine each expression e in $D_S^0(a)$ with all definitions for any cell referenced by e, which leads to the following inductive definition for $D_S(a)$, the set of *definitions* of a. $D_S(a)$ is initially defined to be $\{\{\gamma{:}e\} \mid \gamma{:}e \in D_S^0(a)\}$. Then we repeatedly replace a set of paths $P = \{\gamma_1{:}e_1, \ldots, \gamma_k{:}e_k\} \in D_S(a)$ for which $a' \in \sigma(e_i)$ by the set $P \times D_S^0(a')$ until no such a' exists anymore.

Now the set of all *DU pairs* for address a is given by $\{(a,d,u) \mid d \in D_S(a) \wedge u \in U_S(a)\}$.

For each DU pair we can try to generate a test by solving the constraints stored in the paths (as described in the next section). Whenever the constraint solving fails, a test cannot be generated and an *infeasible* DU pair has been identified.

A test suite that consists of a test for every feasible DU pair is said to be *DU-pair adequate*.

6 Generating DU-Adequate Test Cases

The generation of a test case for a DU pair $(a, \{\gamma_1{:}e_1, \ldots, \gamma_k{:}e_k\}, \gamma_{k+1}{:}e_{k+1})$ requires solving the constraint $\gamma_1 \wedge \ldots \wedge \gamma_k \wedge \gamma_{k+1}$. In a first step, we group the constraints by involved addresses so that we obtain a constraint of the form

$$\gamma^{a_1} \wedge \gamma^{a_2} \wedge \ldots \wedge \gamma^{a_n}$$

where each γ^{a_i} is of the form

$$\gamma_1^{a_i} \vee \gamma_2^{a_i} \vee \ldots \vee \gamma_{k_i}^{a_i}$$

and each $\gamma_j^{a_i}$ is of the form

$$a_i\ \omega\ v_i^1 \wedge a_i\ \omega\ v_i^2 \wedge \ldots \wedge a_i\ \omega\ v_i^{m_{ij}}$$

That is, for each address a_i we obtain an alternative of constraints, each of which determines through a conjunction of value comparisons possible input values for the cell at address a_i. Note that by construction each $\gamma_j^{a_i}$ contains at most one address, namely a_i.

The attempt at solving each constraint γ^{a_i} can have one out of two possible outcomes.

1. The constraint solver might succeed, in which case the solution is, for each address, a range of values that satisfy the constraints. For each address, 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 a_i none of the alternative constraints $\gamma_j^{a_i}$ is solvable. In this case it is not possible to generate any test case that would be able to execute the path. Therefore, failure of the constraint solving process indicates that the particular path for a definition or use cannot be exercised.

If all constraints have been successfully solved, a test case can be created by taking values from computed ranges for each address and by evaluating the formula to be tested (of which e_{k+1} is a subformula) using these values (see Section 4). A test case has the following form.

$$(\{(a_1, v_1), \ldots, (a_n, v_n)\}, v)$$

If the constraint solver fails while trying to solve the constraints for a DU pair, that DU association cannot be exercised given the constraints. In other words, unsolvable constraints on input data cells allow us to automatically detect *infeasible DU pairs* in the spreadsheet program. In general, it might not be possible to execute all of the DU associations in spreadsheets. The problem of identifying infeasible DU pairs in programs written in general-purpose programming languages is undecidable [25]. Detection of infeasible DU pairs is easier in the case of spreadsheets languages like Excel since they do not have loop constructs or recursion.

To illustrate how our algorithm works, we revisit the scenario described in Section 3 and explain how AutoTest generates test cases for the spreadsheet shown in Figure 1. In

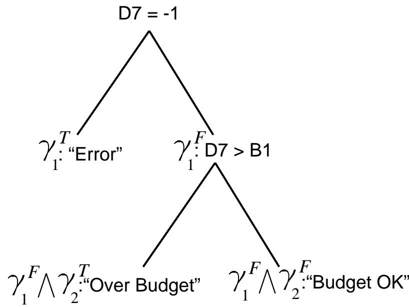
particular, we show how test cases are generated for the formula in B9. To test the formula in B9 we need to test all definitions of D7 and B1 and their uses in the formula in B9.

$$\text{IF}(\text{D7} = -1, \text{“Error”}, \text{IF}(\text{D7} > \text{B1}, \text{“Over Budget”}, \text{“BudgetOK”}))$$

The formulas that affect the definitions of D7 are:

$$\text{D7} = \text{IF}(\text{B8} = 1, -1, \text{D4} + \text{D5} + \text{D6})$$
$$\text{B8} = \text{IF}(\text{OR}(\text{B4} < 0, \text{B5} < 0, \text{B6} < 0), 1, 0)$$

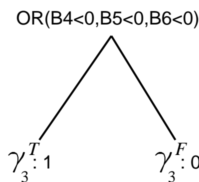
The constraint tree for B9 shows the uses of D7 and B1 in B9. Only the constraint $\gamma_1^F \equiv \text{D7} \neq -1$ is needed in the following.



The use of D7 in the condition $\text{D7} = -1$ is always executed, so we do not need to generate any constraints for it. However, to reach the use of D7 and B1 in the condition $\text{D7} > \text{B1}$ we need to satisfy the constraint γ_1^F. (Since B1 is an input cell, satisfying the constraint γ_1^F fully tests all DU associations of B1 in B9.)

The constraint tree for the formula in B8 is shown below. The two leaves represent two definitions for which we have the constraints:

$$\gamma_3^T \equiv \text{B4} < 0 \lor \text{B5} < 0 \lor \text{B6} < 0$$
$$\gamma_3^F \equiv \text{B4} \geq 0 \land \text{B5} \geq 0 \land \text{B6} \geq 0$$



Since B4, B5, and B6 are input cells, the definitions in B8 are determined by the two constraints inferred for the formula in the cell alone.

Cell D7 also has two definitions since the expression tree of the formula in the cell has two leaves. In this case we have the following constraints for the definitions.

$$\gamma_4^T \equiv \text{B8} = 1$$
$$\gamma_4^F \equiv \text{B8} \neq 1$$

B8 = 1

γ_4^T: -1 $\quad$ γ_4^F: D4 + D5 + D6

Since the formulas in D4, D5, and D6 do not contain conditionals, no constraints are generated for their definitions. Combining the definitions of B8 with those of D7, we obtain the following four definitions for D7.

$$\{\gamma_4^T : -1, \gamma_3^T : 1\}, \{\gamma_4^T : -1, \gamma_3^F : 0\},$$
$$\{\gamma_4^F : \text{D4+D5+D6}, \gamma_3^T : 1\}, \{\gamma_4^F : \text{D4+D5+D6}, \gamma_3^F : 0\}$$

The four definitions combined with the two uses in B9 give rise to 8 DU pairs.

As mentioned earlier, both definitions of D7 already hit the use in the condition $\text{D7} = -1$. Therefore, for generating test cases for the four DU pairs resulting from all the definitions of D7 and this use, we can solve the sets of constraints shown above. Since the sets $\{\gamma_4^T : -1, \gamma_3^F : 0\}$ and $\{\gamma_4^F : \text{D4+D5+D6}, \gamma_3^T : 1\}$ cannot be satisfied,[5] we are left with $\{\gamma_4^T : -1, \gamma_3^T : 1\}$ and $\{\gamma_4^F : \text{D4+D5+D6}, \gamma_3^F : 0\}$. The first set of constraint can be satisfied by setting value in B4 to -1,[6] and the second set of constraints is already satisfied by the values in the spreadsheet.

To test the use of D7 in the condition $\text{D7} > \text{B1}$, we combine the definitions of D7 with those of this use to get the following sets of constraints.

$$\{\gamma_4^T : -1, \gamma_3^T : 1, \gamma_1^F : \text{D7} > \text{B1}\},$$
$$\{\gamma_4^T : -1, \gamma_3^F : 0, \gamma_1^F : \text{D7} > \text{B1}\},$$
$$\{\gamma_4^F : \text{D4+D5+D6}, \gamma_3^T : 1, \gamma_1^F : \text{D7} > \text{B1}\},$$
$$\{\gamma_4^F : \text{D4+D5+D6}, \gamma_3^F : 0, \gamma_1^F : \text{D7} > \text{B1}\}$$

Two sets of constraints cannot be solved in this case, for the same reason discussed above. Moreover, satisfying $\gamma_4^T : -1$ and $\gamma_3^T : 1$ leads to the output -1 in D7, which will not satisfy $\gamma_1^F : \text{D7} > \text{B1}$. Therefore, the only set of constraints that can be solved is $\{\gamma_4^F : \text{D4+D5+D6}, \gamma_3^F : 0, \gamma_1^F : \text{D7} > \text{B1}\}$, and this is already satisfied by the current values in the spreadsheet.

Note that, formally, the actual test case is the set of address-value pairs that satisfy the constraints for a DU pair. Only the generated values are shown in the AutoTest interface—the values for the other input cells that affect the formula output that are already in the spreadsheet are implicitly part of the test case and not shown in the interface.

7 Evaluation

For AutoTest to be useful as a tool for testing spreadsheets, it has to be both *effective* and *efficient*. Effectiveness is judged with respect to a test-adequacy criterion, DU adequacy in this case. A more effective tool in this respect would be one which is capable of generating test cases that exercises more of the feasible DU pairs. Efficiency is measured in terms of the time taken by the system to generate the test cases that meet the adequacy criterion. This factor is especially important in the case of spreadsheet systems with their support for immediate visual feedback.

Since we are comparing AutoTest against HMT, we follow the evaluation of HMT described in [13] and take into consideration two dependent variables.

1. *Ultimate effectiveness*, defined as the percentage of the total number of feasible DU associations

[5] Satisfying γ_3^T leads to 1 in B8 which, in turn, results in γ_4^T being satisfied. On the other hand, satisfying γ_3^F leads to 0 in B8, which results in γ_4^F being satisfied.

[6] Actually, the first constraint can also be satisfied by setting values in B5 or B6 to -1. Since the three test cases exercise the same DU pair, the system only generates the first one.

2. Response time for test generation

7.1 Effectiveness and Efficiency

Since HMT uses randomization, depending on the technique used, the measures of the dependent variables may change from one run to another. Therefore the ultimate effectiveness score for HMT is averaged over 35 runs and the median response time over 35 runs has been presented in [13]. Our system, on the other hand, always produces the same output given the same starting spreadsheet configuration. We reproduce the results from [13] in Tables 1 and 2 and compare with the numbers obtained by running AutoTest on the same spreadsheets.[7]

AutoTest is also efficient in the sense that it only generates one test case per feasible DU pair for the formula that is being tested. Therefore it generates the minimum number of test cases to be able to execute all feasible DU pairs. Such optimal test suites would save the user time and effort during generation (since the user has to approve a candidate test case before it can be added to the suite of test cases for the formula), test runs, and maintenance of test suites. The ultimate effectiveness scores reported in Table 1 and the response times reported in Table 2 for AutoTest are based on the optimal test suites that are generated by the system. The size of the test suites generated by HMT for achieving the level of coverage shown in Table 1 are not available.

7.2 Discussion

As can be seen from the ultimate effectiveness scores shown in Table 1, AutoTest generates test cases that cause the execution of all feasible DU associations for the spreadsheets used in the evaluation. HMT running the Chaining algorithm has comparable ultimate effectiveness scores.

The response times for test generation for each of the spreadsheets used in the evaluation are shown in Table 2. Note that the figures show the time taken to generate a set of test cases that come as close as possible to a 100% ultimate effectiveness score. From the numbers in Table 2, we see that AutoTest far outperforms the algorithms used by HMT for the generation of test cases.

We can conclude therefore that our AutoTest system, whose approach to generate test cases is based on an derivation, propagation, and solving of constraints, is efficient and accurate. Moreover, AutoTest can also detect infeasible DU associations automatically

8 Conclusions and Future Work

We have presented a system, AutoTest, that supports users of Microsoft Excel in the systematic testing of their spreadsheets. AutoTest automatically generates a minimal set of tests for each formula cell that guarantees a DU adequate test coverage. The system runs efficiently and also produces, as a by-product, information about infeasible DU associations in the spreadsheet.

The test generation approach, which is based on constraint generation, propagation, and solving, is conceptually simple, which is important for at least two reasons. First, it allows its reuse in other systems. For example, we believe that it would be straightforward to integrate this new technique into the WYSIWYT tool. Second, it facilitates extensions to be investigated in future. For example, we plan to extend AutoTest to allow whole regions to be tested by the test cases for a single region-representative formula. We can base this extension effectively on the region inference mechanisms reported in [3].

[7] Note that the first test case "Budget" is the spreadsheet we have used in the scenario in Section 3.

Acknowledgments

We thank Marc Fisher for providing us access to the spreadsheets and data from the studies described in [13].

References

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

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

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

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

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

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

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

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

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

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

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

Spreadsheet	HMT				AutoTest
	Random (without ranges)	Random (with ranges)	Chaining (without ranges)	Chaining (with ranges)	
Budget	99.6%	100.0%	100.0%	100%	100%
Digits	59.4%	97.9%	100.0%	100%	100%
FitMachine	50.5%	50.5%	97.9%	97.9%	100%
Grades	67.1%	99.8%	99.7%	99.9%	100%
MBTI	25.6%	100.0%	99.9%	99.6%	100%
MicroGen	71.4%	99.2%	100.0%	100%	100%
NetPay	40.0%	100.0%	100.0%	100%	100%
NewClock	57.1%	100.0%	99.0%	99.4%	100%
RandomJury	78.8%	83.2%	94.3%	92.7%	100%
Solution	57.7%	78.8%	100.0%	100%	100%

Table 1. Ultimate effectiveness scores of the different techniques per spreadsheet.

Spreadsheet	HMT				AutoTest
	Random (without ranges)	Random (with ranges)	Chaining (without ranges)	Chaining (with ranges)	
Budget	3.8	3.9	9.9	10.6	< 0.01
Digits	2.2	10.1	34.5	28.4	< 0.01
FitMachine	3.8	3.9	13.5	14.3	< 0.01
Grades	7.7	18.6	14.2	14.4	0.5
MBTI	37.5	40.0	31.2	30.7	1.0
MicroGen	2.7	8.1	6.4	6.5	< 0.01
NetPay	1.2	1.2	1.7	1.7	< 0.01
NewClock	2.3	2.5	10.4	9.3	< 0.01
RandomJury	80.2	28.7	182.3	173.2	2.0
Solution	1.3	1.4	18.9	17.9	< 0.01

Table 2. Response times (in seconds) of the different techniques per spreadsheet.

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

[13] M. Fisher, 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*, 2006. To appear.

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

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

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

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

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

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

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

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

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

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

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

[25] E. J. Weyuker. More Experience with Data Flow Testing. *IEEE Trans. on Software Engineering*, 19(9):912–919, 1993.

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