Slicing Spreadsheets:
An Integrated Methodology for Spreadsheet Testing and Debugging

James Reichwein, Gregg Rothermel, Margaret Burnett
Department of Computer Science
Oregon State University
Corvallis, OR 97331
{reichwja,grother,burnett}@cs.orst.edu

Abstract

Spreadsheet languages, which include commercial spreadsheets and various research systems, have proven to be flexible tools in many domain specific settings. Research shows, however, that spreadsheets often contain faults. We would like to provide at least some of the benefits of formal testing and debugging methodologies to spreadsheet developers. This paper presents an integrated testing and debugging methodology for spreadsheets. To accommodate the modeless and incremental development, testing and debugging activities that occur during spreadsheet creation, our methodology is tightly integrated into the spreadsheet environment. To accommodate the users of spreadsheet languages, we provide an interface to our methodology that does not require an understanding of testing and debugging theory, and that takes advantage of the immediate visual feedback that is characteristic of the spreadsheet paradigm.

1 Introduction

Spreadsheet languages, which include commercial spreadsheet systems as a subclass, have proven useful in many domain specific settings, including business management, accounting, and numerical analysis. The spreadsheet paradigm is also a subject of ongoing research in many domain specific areas. For example, there is research into using spreadsheet languages for matrix manipulation problems [33], for providing steerable simulation environments for scientists [7], for high-quality visualizations of complex data [9], and for specifying full-featured GUIs [21].

Despite the end-user appeal of spreadsheet languages and the perceived simplicity of the spreadsheet paradigm, research shows that spreadsheets often contain faults. For example, in an early spreadsheet study, 44% of "finished" spreadsheets still contained faults [4]. A more recent survey of other such studies reported faults in 38% to 77% of spreadsheets at a similar stage [25]. Of perhaps greater concern, this survey also includes studies of "production" spreadsheets actually in use for day-to-day decision-making: from 10.7% to 90% of these spreadsheets contained faults.

One possible factor in this problem is the unwarranted confidence spreadsheet developers have in the reliability of their spreadsheets [10]. Another is the difficulty of creating and debugging spreadsheets: in interviews, experienced spreadsheet users reported that debugging spreadsheets could be hard because tracing long chains of formulas is difficult and because the effects of a small fault may not be visible until they have been propagated to a final result [14, 22].

To begin to address these problems, our previous work [30] presented a testing methodology for spreadsheets. That methodology allowed the user to indicate which cells are correct for a given test case, and to view testedness information inferred from those marks. Building on that work, this paper describes our approach to integrating support for debugging and fault localization with that methodology. This integrated methodology adds the ability to mark which cells are *in*correct for a given test case, and to view fault localization information inferred from both correct and incorrect marks. Key to the effectiveness of our approach is that it is tightly integrated into the spreadsheet environment, facilitating the incremental testing and debugging activities that normally occur during spreadsheet development. Our methodology also employs immediate visual feedback to present information in a manner that requires no knowledge of the underlying testing and fault localization theories.

2 Background: Testing Spreadsheets

The underlying assumption in our previous work has been that, as the user develops a spreadsheet incrementally, he or she is also testing incrementally. We have integrated a prototype implementation of our approach to incremental, visual testing into the spreadsheet language Forms/3 [6]; the examples in this paper are presented in that language.

Testing following our methodology [30] is intended for spreadsheet developers, not software engineers. Thus, our methodology does not include specialized testing vocabulary – in fact, it includes no vocabulary at all, instead presenting test-related information visually. Users test spreadsheets by trying different input values, and validating correct cells with a checkmark. Cells start out with red borders, indicating that they are untested. As cells are checked, their border colors change along a red-blue continuum, becoming bluer as the cell's testedness increases. When all the cells are blue, the spreadsheet is considered tested.

Although users of our methodology need not realize it, they are actually using a dataflow test adequacy criterion [18, 23, 26] and creating *du-adequate* test suites. In the theory that underlies this methodology, a *definition* is a point in the source code where a variable is assigned a value, and a *use* is a point where a variable's value is used. A *definition-use pair*, or *du-pair*, is a tuple consisting of a definition of a variable and a use of that variable. A du-adequate test suite is based on the notion of an *output-influencing all-definition-use-pairs-adequate test suite* [13] and is a test suite that exercises each du-pair in such a way that it participates (dynamically) in the production of an output explicitly validated by the user.

In spreadsheet terms, cells are considered variables. A cell is used when another cell references it, and a cell is defined within its own formula. If a cell's formula contains `if` expressions, then the cell can have multiple definitions. The testedness of a cell is calculated as the number of validated du-pairs with uses in that cell, divided by the total number of du-pairs with uses in that cell. Also, in the output-influencing scheme, testedness propagates against dataflow, so that if a cell a is validated, and if one of the du-pairs that provided a's validated value has its definition in cell b, then any du-pairs that participated in providing b's value are also considered tested.

This underlying theory is hidden from the user, for whom du-pairs represent interactions between cells caused by references in cell formulas. These interactions can be visualized by the user through the display of dataflow arrows between subexpressions in cell formulas, and these arrows are colored to indicate whether the corresponding interaction has been tested.

This methodology also lets the user incrementally and simultaneously develop and test their spreadsheets. If the user adds a new formula or alters an existing formula, the underlying evaluation engine determines the du-pairs affected by this alteration and updates stored and displayed testing information. In this context, the problem of incremental testing of spreadsheets is similar to the problem of regression testing [29] and our solution emphasizes the importance of retesting code affected by modifications.

Figure 1 illustrates our prototype implementation of this methodology in use. The figure depicts a Forms/3 spreadsheet implementing a simple security check. Three key values identifying a person are placed in the cells $key1$, $key2$, and $key3$. The output cells $key1\_out$, $key2\_out$, and $key3\_out$ give a garbled version of the original keys that can be checked against a data base to determine if the person can be accepted. The spreadsheet developer initially validated the three output cells in this program. Then, to test further, the developer entered a different test case consisting of a different value for $key3$. Doing so changed the checkmark on $key3\_out$ to a question mark, indicating that previously displayed values have been validated, but the current ones have not. The formula for $key3\_3$ contains an `if` expression. So far only one branch of this expression has been tested, so the borders for the $key3\_out$ and $key3\_3$ cells are purple (gray in this paper). Cell $key3\_2$ has not been tested at all, so it is red (a light gray in this paper). Cells $key1\_out$, $key1\_1$, $key2\_out$, $key2\_1$, and $key3\_1$ have been completely tested, and have blue borders (black in this paper). The colors of displayed arrows between cells indicate the degree to which dependencies (interactions) between those cells have been validated.

3 An Integrated Methodology for Testing and Debugging Spreadsheets

During the course of a spreadsheet development session, users will locate failures in their spreadsheets: cases where cell outputs are incorrect. The interactive and incremental manner in which spreadsheets are created suggests that on discovering such failures, users may immediately attempt to locate and correct the faults that cause those failures.

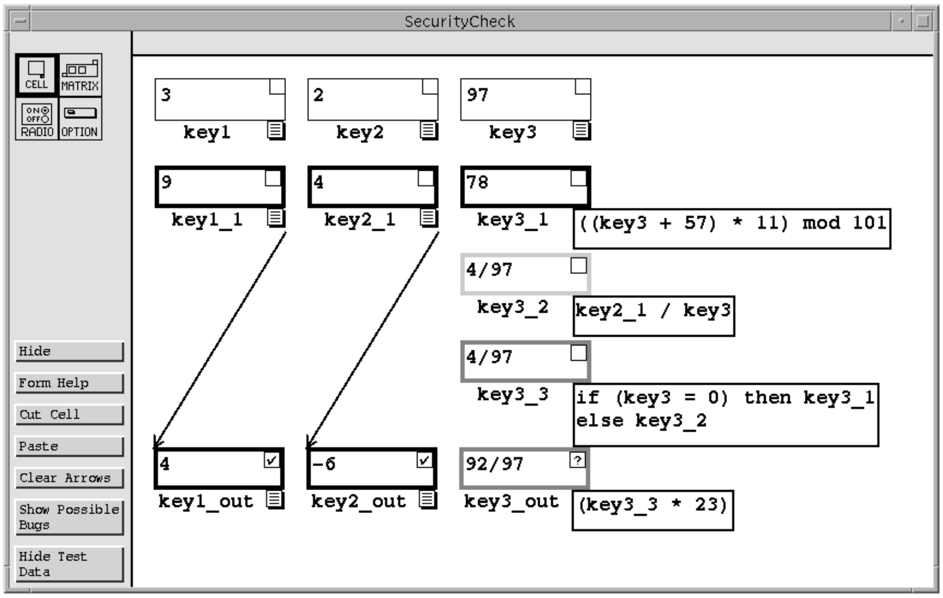

Figure 1: Forms/3 `SecurityCheck` spreadsheet with testing information displayed.

We wish to provide spreadsheet end users with automated support for this process of debugging and fault localization. There are three attributes of spreadsheet languages and their users that place constraints on methodologies providing such support:

- **Spreadsheets are modeless.** Spreadsheet creation does not require the separate code, compile, link, and execute modes typically required by traditional programming languages. Spreadsheet developers simply write formulas, enter values, and see results. Thus, in order for testing and debugging techniques to be useful for spreadsheet developers, the developers must be allowed to debug and test incrementally in parallel with spreadsheet development.

- **Spreadsheet developers are not likely to understand testing and debugging theory.** Given their end user audience, spreadsheet testing and debugging techniques cannot depend on the user understanding testing or debugging theory, nor should they rely on specialized vocabularies based on such theory.

  Furthermore, spreadsheet developers are not liable to understand the reasons if debugging feedback leads them astray. They are more likely to become frustrated, lose trust in our methodology, and ignore the feedback. Therefore, our methodology must avoid giving false indications of faults where no faults exist.

- **Spreadsheets offer immediate feedback.** When a spreadsheet developer changes a formula, the spreadsheet displays the results quickly. Users have come to expect this responsiveness from spreadsheets and may not accept functionality that significantly inhibits responsiveness. Therefore the integration of testing and debugging into the spreadsheet environment must minimize the overhead it imposes.

Our methodology has been developed with these constraints in mind.

3.1 Slicing and dicing

Our debugging methodology is based on techniques for program slicing and dicing developed originally for imperative programs. We briefly review those techniques here in turn.

Program slicing was introduced by Weiser [34] as a technique for analyzing program dependencies. A program slice is defined with respect to a slicing criterion $\langle s, v\rangle$ in which s is a program point and v is a subset of program variables. A slice consists of a subset of program statements that affect, or are affected by, the values of variables in v at s [34]. *Backward slicing* finds all the statements that affect a given variable at a given statement, whereas *forward slicing* finds all the statements that are affected by a given variable at a given statement. Weiser's slicing algorithm calculates *static* slices, based solely on information contained in source code, by iteratively solving dataflow equations. Other techniques [15, 24, 27, 31] calculate static slices by constructing and walking dependence graphs.

Korel and Laski [16] introduced *dynamic slicing*, in which information gathered during program execution is also used to compute slices. Whereas static slices find statements that may affect (or may be affected by) a given variable at a given point, dynamic slices find statements that may affect (or may be affected by) a given variable at a given point under a given execution. Dynamic slicing usually produces smaller slices than static slicing. Dynamic slices are calculated iteratively in [16]; an approach that uses program dependence graphs has also been suggested [1].

A great deal of additional work has been done on program slicing. An extensive survey of slicing is given in [32]. A more recent survey of dynamic slicing is given in [17].

Program dicing was introduced by Lyle and Weiser [20] as a fault localization technique for further reducing the number of statements that need to be examined to find faults. Whereas a slice makes use only of information on incorrect variables at failure points, a dice also makes use of information on correct variables, by subtracting the slices on correct variables away from the slice on the incorrect variable. The result is smaller than the slice on the incorrect variable; however, a dice may not always contain the fault that led to a failure.

Lyle and Weiser describe the cases in which a dice on an incorrect variable not caused by an omitted statement is guaranteed to contain the fault responsible for the incorrect value in the following theorem [20]:

> **Dicing Theorem.** A dice on an incorrect variable contains a fault (except for cases where the incorrect value is caused by omission of a statement) if all of the following assumptions hold:
>
> 1. Testing has been reliable and all incorrectly computed variables have been identified.
> 2. If the computation of a variable, v, depends on the computation of another variable, w, then whenever w has an incorrect value then v does also.
> 3. There is exactly one fault in the program.

In this theorem, the first assumption eliminates the case where an incorrect variable is misidentified as a correct variable. The second assumption removes the case where a variable is correct despite depending on an incorrect variable (e.g. when a subsequent computation happens to compensate for an earlier incorrect computation, for certain inputs.) The third assumption removes the case where two faults counteract each other and result in an accidentally correct value.

Given the assumptions required for the Dicing Theorem to hold, it is clear that dicing must be an imperfect technique in practice. Thus, Chen and Cheung [8] explore strategies for minimizing the chance that dicing will fail to expose a fault that could have produced a particular failure, including the use of dynamic rather than static slicing.

3.2 Integrated testing and debugging

We have developed an integrated and incremental testing and debugging methodology that uses a fault localization technique similar to dicing. To achieve a close integration with the spreadsheet environment, our methodology gives spreadsheet developers the ability to edit, test, or debug a spreadsheet at any point during the development process without losing previously gathered testing or debugging information. To make this possible, our methodology provides the following user operations:

- The ability to view or hide testing and fault localization information at any time.
- The ability to incrementally mark cell values correct or incorrect for a single test case.
- The ability to change test cases without losing testing or debugging information gathered during previous testing.
- The ability to make a potential bug fix or other formula edit without losing testing or debugging information.

We next discuss how our methodology provides these functionalities while satisfying the constraints imposed by spreadsheet environments. We present the material in the context of an integrated spreadsheet development, testing, and debugging session.

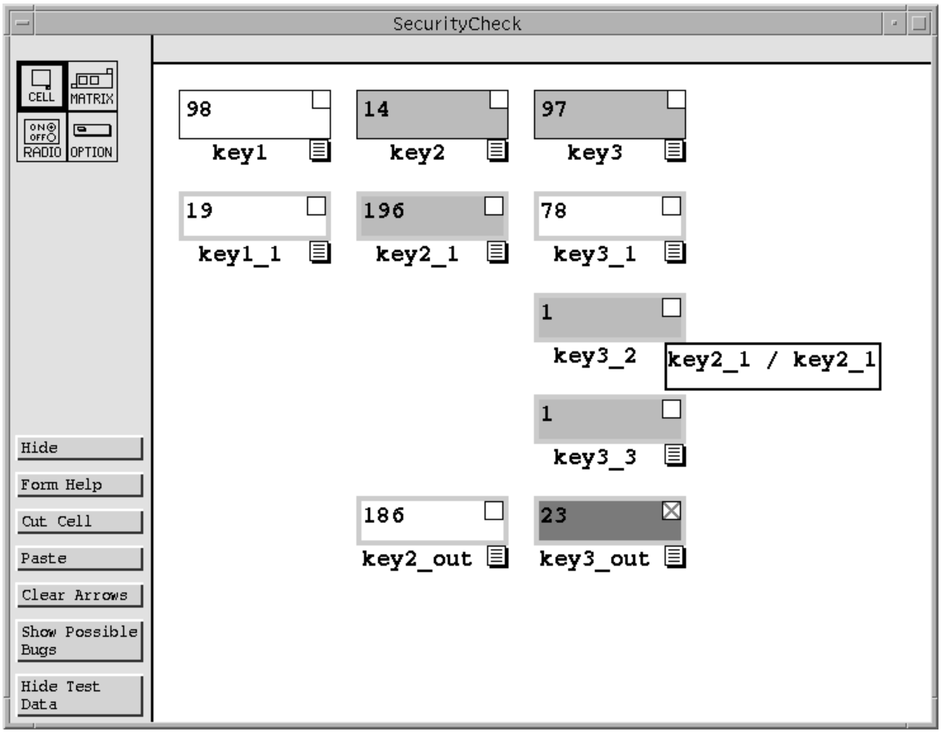

Figure 2: `SecurityCheck` spreadsheet at an early stage.

3.2.1 Spreadsheet development and simple fault localization

Suppose that, starting with an empty spreadsheet, the user begins to build the `SecurityCheck` application discussed in Section 2 and reaches the state shown in Figure 2. At this state, the user's spreadsheet contains an incorrect output: the $key3\_out$ cell, which should contain the value $(4508/97)$, contains the value 23. This is caused by an incorrect cell reference in the cell $key3\_2$. Instead of dividing the value of $key2\_1$ by the value of $key3$, the formula for $key3\_2$ divides $key2\_1$ by itself.

As soon as an incorrect output is noticed, users can place an "X" mark in a cell to indicate that it has an incorrect value. Suppose the user places such a mark in the $key3\_out$ cell.

Now, suppose the user decides to investigate the cause of this failure immediately. Having placed one or more X marks, the user can view fault localization information by pressing a "Show Possible Bugs" button. This causes cells suspected of containing faults to be highlighted in red (gray in this paper); in this case, the highlighted cells are those contained in the backward dynamic slice of $key3\_out$. Note that the border color scheme is different from that used in our previous testing methodology. Previously border colors went from red, representing untested, to blue, representing tested. Now border colors start out at black (light gray in this paper) to represent untested, move to various shades of purple to represent partially tested, and finally move to blue (black in this paper) to represent fully tested. This color scheme was chosen to avoid a conflict between red border colors representing an untested cell, and a red background color representing a potentially faulty cell.

Now six of the cells are highlighted red, including $key3\_out$. Two of these cells are constant cells. For our purposes, a constant cell is any cell whose formula does not refer to another cell. These cells are highlighted in case the incorrect value is caused by a data entry error. At this point the user, knowing that the entered data is correct, can ignore those cells and concentrate on the remaining four cells.

$Predecessors(C)$	The set of cells in S that C references in its formula.
$Successors(C)$	The set of cells in S that reference C in their formulas.
$DynamicPredecessors(C)$	The set of cells $D \in S$ such that D's value was used the last time the value of C was computed.
$DynamicSuccessors(C)$	The set of cells $D \in S$ such that D used the value of C the most recent time D's value was computed.
$BackwardSlice(C)$	The transitive closure on $Predecessors(C)$.
$ForwardSlice(C)$	The transitive closure on $Successors(C)$.
$DynamicBackwardSlice(C)$	The transitive closure on $DynamicPredecessors(C)$.
$DynamicForwardSlice(C)$	The transitive closure on $DynamicSuccessors(C)$.
$IncorrectDependentsOf(C)$	The set of cells in $DynamicForwardSlice(C)$ that have been marked incorrect.
$CorrectDependentsOf(C)$	The set of cells in $DynamicForwardSlice(C)$ that have been marked correct.

Table 1: The definitions used in determining fault likelihood. Here S is a spreadsheet, and C is any cell in S.

3.2.2 Applying additional knowledge to further refine fault localization information

Dicing in a spreadsheet environment would find the set of cells that contribute to a value marked incorrect but not to a value marked correct. The set of cells indicated by dicing could exclude a fault if one of the conditions in Dicing Theorem were violated. However, one constraint our methodology must satisfy is that the user should not be frustrated by searching through highlighted cells to find that none of them contain faults. We believe that the restrictions imposed by the Dicing Theorem are too strict to be practical in a spreadsheet environment. Therefore dicing cannot be used for our methodology.

Dicing makes a binary decision about cells: either a cell is indicated or it is not. To allow the conditions in the Dicing Theorem to be violated without causing user frustration, our technique does not make a binary decision about which cells to include or exclude. Instead, our methodology estimates the likelihood that a cell contributes to a value marked incorrect. This likelihood is presented to the spreadsheet developer by highlighting suspect cells in different shades of red. We call this likelihood the *fault likelihood* of a cell. Let I be the set of cell values marked incorrect by the spreadsheet developer. The fault likelihood of a cell C is an estimate of the likelihood that C contains a fault that contributes to an incorrect value in I.

Returning to the `SecurityCheck` example, suppose that having seen several cells highlighted in red as potentially faulty, the user does not yet wish to examine formulas, but would prefer to restrict the potential fault site further. One way to do so is to test other parts of the spreadsheet for correctness. In this case, only one of the other output cells has been created, $key2\_out$. This cell is correct, so the user can mark it with a check box. The result of doing so is shown in Figure 3. Now both $key2\_1$ and $key2$ contain a lighter shade of red than before because they contribute to a correct cell value. The shade of red in the background of a cell indicates the fault likelihood of that cell. A lighter shade of red indicates a lower likelihood, and a darker shade of red indicates a higher likelihood.

There is no way to compute an exact value for the fault likelihood of a cell: we can only estimate it based on the number of values marked correct or incorrect that depend on a cell's value. Our strategy for doing so is to maintain six properties, described below, which rely on the definitions in Table 1.

Property 1 *If* $IncorrectDependentsOf(C) \neq \phi$ *then* C *has at least a minimal fault likelihood.*

This property ensures that every cell in the backward dynamic slice of a value marked incorrect will be highlighted. This reduces the chance that the user will become frustrated searching for a fault that is not there, and in our opinion is essential to a fault localization methodology for spreadsheets. However, there are still two situations in which the highlighted cells might not include a fault responsible for a value marked incorrect. The first situation can occur when a fault is caused by the omission of a cell. The second situation can occur when a correct value is mistakenly marked incorrect. These situations, however, cannot in general be avoided by any fault localization methodology.

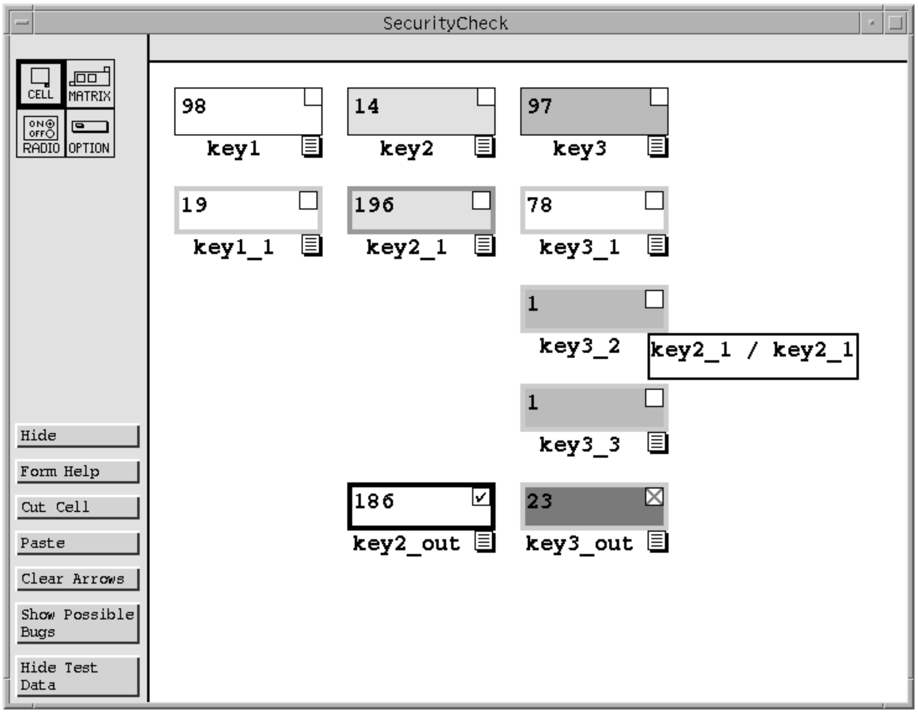

Figure 3: `SecurityCheck` spreadsheet following additional validation.

Property 1 determines the overhead imposed by the actions of marking a cell correct or incorrect. The time complexity of both operations must be $\mathbf{O}(|DynamicBackwardSlice(C)|)$, where C is the cell being marked. This is approximately the same as the cost of computing the value of C for the first time (i.e., when the predecessors also need to be computed).

In order to localize a fault to a set of cells smaller than the dynamic backward slice, we maintain several other properties to determine how fault likelihood should be estimated. These properties ensure that cells highlighted a bright shade of red have a higher likelihood of containing a fault than those marked a lighter shade of red.

Property 2 *The fault likelihood of C is proportional to $|IncorrectDependentsOf(C)|$.*

Property 3 *The fault likelihood of C is inversely proportional to $|CorrectDependentsOf(C)|$.*

Property 2 is based on the assumption that the more incorrect computations a cell contributes to, the more likely it is that the cell contains a fault. Conversely, Property 3 is based on the assumption that the more correct computations a cell contributes to, the less likely it is that the cell contains a fault.

Property 4 *If C has a value marked incorrect, the fault likelihood of C is high.*

This property is based on the assumption that a good place to start looking for a fault is the place where the effect of the fault was first recognized. We give that cell a high fault likelihood so that it stands out.

Property 5 *An incorrect mark on C blocks the effects of any correct marks on cells in $DynamicForwardSlice(C)$, preventing propagation of the correct marks' effects to the fault likelihood of cells in $DynamicBackwardSlice(C)$.*

This property is relevant when a correct cell value depends on an incorrect cell value. There are three possible explanations for such an occurrence. The first is that a formula of one of the cells between the correct cell and the incorrect cell somehow converts the incorrect value to a correct one. The second is that there is another fault between the two cells that counteracts the effect of the incorrect value. The third is that the developer made a mistake in marking one of these two cells. We choose to trust the developer's decision in this case and assume one of the first two situations. For both of these situations the incorrect value does not contribute to the correct value. Therefore the effects of the correct mark should not propagate back to cells the incorrect value depends on.

Property 6 *A correct mark on C blocks the effects of any incorrect marks on cells in DynamicForwardSlice(C), preventing propagation of the incorrect marks' effects to the fault likelihood of cells in DynamicBackwardSlice(C), except for the minimal fault likelihood required by Property 1.*

This property is relevant when a value marked incorrect depends on a value marked correct. In dicing, the dynamic backward slice of the correct value would be completely subtracted from that of the incorrect value. However we need to be more conservative and assume that a violation of the Dicing Theorem is possible. Thus, the cells in the dynamic backward slice of the correct value are given a low but nonzero fault likelihood.

3.2.3 Implementing the Properties

The above properties allow for many different ways of estimating fault likelihood. First we require the following definitions to handle the "blocks" introduced by the fifth and sixth properties. Let $NumBlockedIncorrectDependentsOf(C)$ be the number of cell values belonging to cells in $DynamicForwardSlice(C)$ that are marked incorrect but are blocked by a value marked correct along the data flow path from C to the value marked incorrect. Let $NumReachableIncorrectDependentsOf(C)$ be $|IncorrectDependentsOf(C)| - NumBlockedIncorrectDependentsOf(C)$, or in other words the number of cell values marked incorrect whose effects reach C without being blocked. Similar definitions are made for $NumBlockedCorrectDependentsOf(C)$ and $NumReachableCorrectDependentsOf(C)$.

As a starting point, we decided to divide fault likelihood into six distinct ranges: "none", "very low", "low", "medium", "high", and "very high". To estimate the fault likelihood for a cell C, we first assign a range to the values of $NumReachableIncorrectDependentsOf(C)$ and $NumReachableCorrectDependentsOf(C)$ using Table 2. These ranges can be associated with numeric values from 0, representing "none", to 5, representing "very high". To combine these ranges to determine the resulting fault likelihood, we use the function:

Range	$NumReachableIncorrectDependentsOf(C)$ or $NumReachableCorrectDependentsOf(C)$
none	0
low	1–2
medium	3–4
high	5–9
very high	10+

Table 2: $NumReachableIncorrectDependentsOf(C)$ and $NumReachableCorrectDependentsOf(C)$ yield six distinct fault likelihood ranges as shown.

$$\text{fault likelihood}(C) = \max(1, RID - \left\lfloor \frac{RC}{2} \right\rfloor)$$

where $RID = NumReachableIncorrectDependentsOf(C)$ and $RC = NumReachableCorrectDependentsOf(C)$.

We make three exceptions. If $IncorrectDependentsOf(C) = \phi$, then the fault likelihood of C is "none". If C has been marked incorrect, its fault likelihood is assigned a value of "very high", in keeping with Property 4. If $NumReachableIncorrectDependentsOf(C) = 0$, but $NumBlockedIncorrectDependentsOf(C) > 0$, then the fault likelihood of C is assigned a value of "very low". This is to maintain Property 1, and ensures that every cell in the dynamic backward slice of a value marked incorrect is highlighted.

Returning to Figure 3, $key3\_out$ is a bright red color because it has a "very high" fault likelihood. Cells $key3\_3$ and $key3\_2$ each have one reachable incorrect dependent, so they have fault likelihoods of "low". Cell $key2\_1$ has a "very low" fault likelihood because it has one reachable incorrect dependent and another reachable correct dependent.

3.2.4 Applying additional test cases

Suppose the user developing the `Security Check` spreadsheet still wants to further narrow down the set of possible locations of the fault. One option is for the user to apply additional test cases. Figure 4 shows the result of entering a new test case into $key1$, $key2$, and $key3$. Now both $key2\_out$ and $key3\_out$ are correct, so the user checks both cells. The information about the previous test case is not lost, so now $key3\_3$ has a reach-

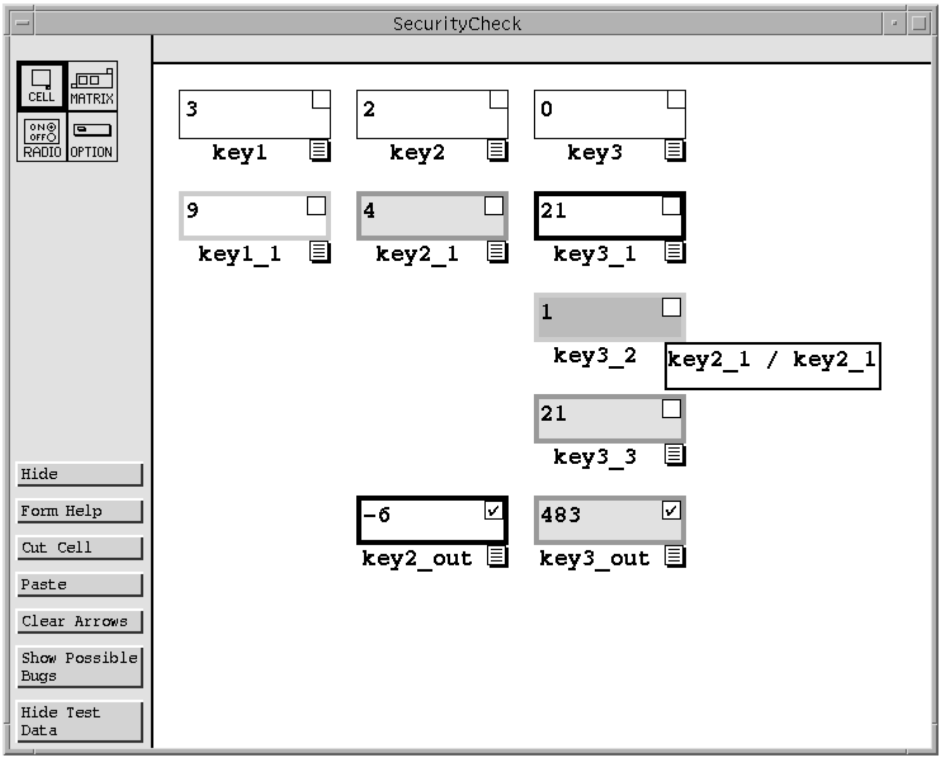

Figure 4: `SecurityCheck` spreadsheet following application of additional test cases.

able correct dependent for this test case and a reachable incorrect dependent for the previous test case. This gives $key3\_3$ a fault likelihood of "very low". However, in this test case $key3\_2$ is no longer in the dynamic backward slice of $key3\_out$. This is because $key3\_3$ is designed to not use $key3\_2$ if its result would be a divide by zero. There is still one reachable incorrect dependent from the previous test case for $key3\_2$, so its fault likelihood stays "low". Now the faulty cell, $key3\_2$, has the brightest red color on the form, suggesting that it is most likely to contain a fault.

In order for testing and debugging information to be preserved between test cases, our methodology must respond correctly to formula edits. Any formula edit that changes a constant cell to another constant is considered a change in test case. When this occurs, all of the marked cells dependent on the changed cell must have their marks removed and replaced with question marks. However, the effects of those marks on cell colors and testing and debugging information should not be removed by changing a test case. In effect, the mark is "hidden" behind the question mark.

This process of preserving testing and debugging information across edits adds no additional time complexity to the process of editing a cell, because the spreadsheet environment must already visit the dependents of a changed cell in order to mark them as requiring recomputation.[1]

3.2.5 Maintaining testing and fault information after changes to formulas

Now suppose the developer of the `Security Check` application decides to fix the fault. This involves editing the formula for $key3\_2$ from "$key2\_1/key2\_1$", to "$key2\_1/key3$". Figure 5 shows the result of this action. As expected, $key3\_2$ now contains a divide by zero error; this is why $key3\_3$ uses $key3\_1$ instead. However, now that the formula has changed, the marks placed on $key3\_out$ are out of date. Not only must they be removed, but their effects on testing and debugging information must be undone. In effect, the affected cells can

[1]For more information on marking and evaluation strategies applicable to spreadsheet languages, see[5].

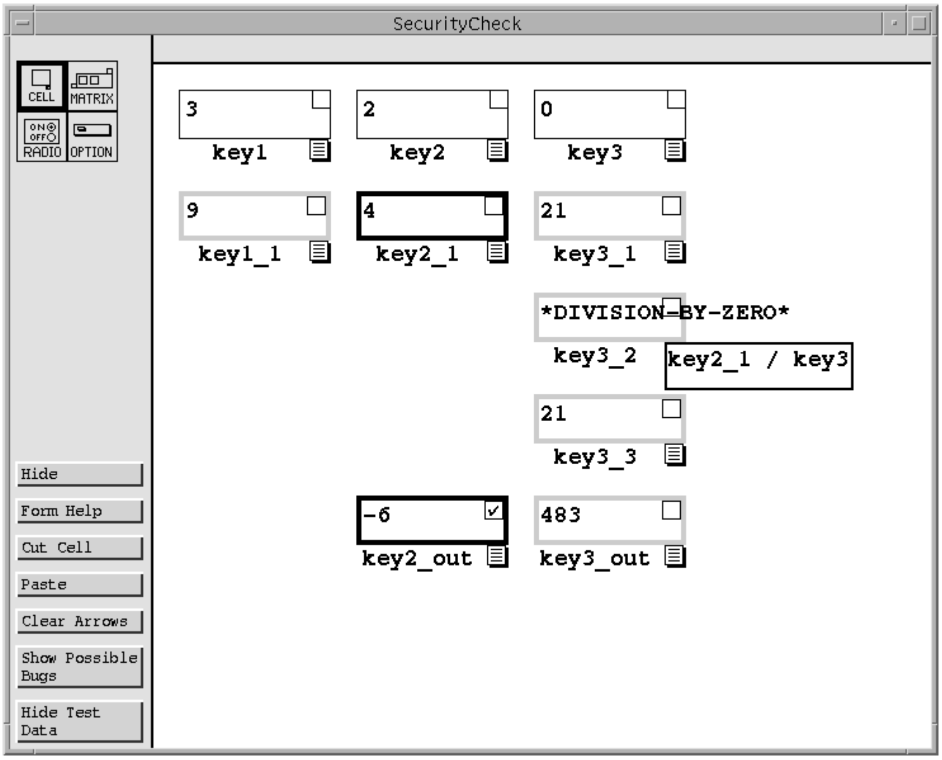

Figure 5: Corrected `SecurityCheck` spreadsheet.

no longer be considered tested because they rely on a new and untested formula. This also encourages the user to perform regression testing on the cells affected by the change.

Whenever a formula edit is made that does not change a constant to a constant, testing and debugging information must be removed. This information must be removed for all prior test cases, so the effects of every mark ever placed on a cell in the static forward slice of the edited cell must be removed.

Furthermore, our methodology relies on static and dynamic slicing information. This information must be kept up to date whenever any edit is made.

Whenever a formula edit occurs, the spreadsheet environment must mark all affected cells to be recomputed. This requires storing some form of static or dynamic successor information. An algorithm for a formula edit that maintains static successor information for a cell C must visit the cells in $Predecessors(C)$ and update their static successor information. This includes the cells in $Predecessors(C)$ both before the formula change and after the change. The static predecessor information can be gathered from the cell's formula. Let SP represent the the maximum number of static predecessors before and after the formula edit. The worst case time complexity for such an algorithm is $\mathbf{O}(SP + |ForwardSlice(C)|)$. Such an algorithm provides all the information needed for static slicing; the slicing algorithm can perform a graph walk using the cell's formula during a backward slice, and the static successor information for a forward slice.

Dynamic successor and predecessor information can be maintained during the process of recalculating cells. An algorithm to recompute a cell C and maintain dynamic successor and predecessor information must first visit the cells in $DynamicPredecessors(C)$ to update their dynamic successor information. Then C must be recomputed. This involves visiting every cell required for C's new value. During this process, the dynamic predecessor information for C can be collected, and the dynamic successor information for the cells used to compute C can be updated. Thus the only additional cost needed to maintain dynamic slicing information is the time needed to update the dynamic successor information for the cells previously used by C. Let DP be the maximum number of dynamic predecessors of C, both before and after C is recomputed. The time complexity for an algorithm to compute C's value is $\mathbf{O}(DP + COMP)$, where $COMP$ is the time required to compute the cell's value once the values of its predecessors are known. Note that in some spreadsheet languages cell formulas can use functions whose time complexity is not bounded by the number of operands. However in order to evaluate the effect our methodology has on responsiveness, we assume the worst case scenario in which the time complexity of recomputing a cell is dominated by the cost of maintaining dynamic successor and predecessor information.

Our methodology also adds overhead by removing testing information when editing a formula. As discussed earlier, a formula edit to provide a different a test case adds no additional overhead to an algorithm for accepting an edit to a cell. However, editing a non-constant formula does. An algorithm that handles such an edit to a cell C while maintaining static slicing information and testing and debugging information must not only visit the cells in $ForwardSlice(C)$, but also the cells in the backward static slices of every cell that had been marked during the current or a previous test case. Let M be the number of cells marked in $\mathit{ForwardSlice}(C)$. Let B be the maximum length of a backward static slice of a marked cell in $\mathit{ForwardSlice}(C)$. The worst case time complexity for editing a formula while maintaining fault localization information is $\mathbf{O}(SP + M \cdot B + |\mathit{DynamicForwardSlice}(C)|)$.

To analyze the effects on responsiveness of maintaining testing and debugging information, we must consider two separate activities. The first is the formula edit itself. The second is the calculation phase where at minimum every cell on the screen is recalculated. From the user's point of view, the system is responsive only if both the edit and the recalculation of on-screen cells happen quickly. Therefore we merge the two and define a *responsive edit* to be an operation where a cell is edited and at least the on-screen cells are updated. For our methodology, the worst case time complexity for a responsive edit is $\mathbf{O}(SP + M \cdot B + DP \cdot |\mathit{DynamicForwardSlice}(C)|)$. There are three overhead factors that must be considered, SP, DP, and $M \cdot B$.

Spreadsheet environments maintain successor information in order to mark or recompute cells that need to be recomputed. Therefore, either SP or DP will already be a factor in the complexity for a spreadsheet environment that does not implement our methodology. However, one of these factors may be added as overhead from our methodology. The factor DP is likely to have a larger effect than SP, because every cell that is recalculated to update the screen must update the dynamic successor information of its predecessors. In spreadsheet languages that do not support ranges, such as Forms/3, DP or SP are likely to be constant bound. This is because the number of predecessors depends on the number of references in a formula. Since long formulas are often awkward, users are more likely to break them up into distinct cells than to create large formulas that reference a large number of cells. However, if ranges are added, a short formula is capable of referencing a large number of cells. Ranges also add additional complexity for visualizing a slice, as drawing an arrow to each cell in the range will result in a jumble of arrows. This can be dealt with by handling a range as a single entity. For example, the data flow arrows in Excel point to a box around a range rather than to every cell in the range.

The largest factor introduced by our methodology is likely to be the $M \cdot B$ factor introduced when changing a formula. Since B is the size of a static slice, it could be large. However we have no way of knowing how large M will be, as this depends on how many marks are placed by the user.

It remains to be seen if the factors of $M \cdot B$, SP or DP can be large enough to negatively affect the responsiveness of our methodology. However, there are approaches that can mitigate the effects of these factors. One approach would be a multithreaded approach to updating cell values and testing and debugging information. A background thread can be used to update information while the user is free to continue interacting with the spreadsheet. This allows the spreadsheet environment to stay interactive. However as the user makes changes the state of the spreadsheet could become increasingly inconsistent. This can be partially mitigated by placing priority on updating information for on-screen cells. This approach would be useful only if the benefits to be gained by our technique outweighed the difficulty of implementing it.

4 Related Work

Previous work related to testing and debugging spreadsheets has been limited to auditing tools. Most of the work on such tools has taken place in commercial applications. For example, Microsoft Excel 97 allows users to place restrictions on the value of a cell, to place comments on cells, and to draw arrows that track dataflow dependencies. The dataflow arrows implement a form of static slicing that lets users view backward and forward slices by expanding the arrows out one dependence level at a time.

The only research work we are aware of addressing auditing tools is by Davis [12]. Davis proposes two tools, a dataflow arrow tool and an automatically derived dataflow graph. The arrow tool is similar to the arrows in Excel; however, from any one cell it is possible only to request arrows that show one dependence level. The dataflow graph tool draws a graph using different symbols to categorize cells as input, output, decision variables, parameters, or formulas. This graph is similar to the graph suggested by Ronen, Palley and Lucas [28], however it is generated automatically by the system instead of being drawn by the user.

Our testing and debugging methodology utilizes dataflow arrows similar to those used by Excel and Davis's tools. Unlike Excel arrows, however, our dataflow arrows allow users to display an entire slice at once, or to limit the depth of the slice. The user can also choose between forward and backward slicing, as well as dynamic and static slicing. At the testing level, users can also display dataflow arrows at a finer granularity (between subexpressions); this ability will soon be adapted into our debugging functionality.

There has also been research into using interactive, visual techniques to aid in debugging, particularly as it relates to program comprehension (eg: [2, 19]). This approach integrates debugging functionality with reviewing execution histories, both graphically and textually, to better understand program behavior and thus find faults.

5 Conclusions and Future Work

Due to the popularity of commercial spreadsheets, spreadsheet languages are being used to produce software that influences important decisions. Furthermore, due to recent advances from the research community that expand its capabilities, the use of this paradigm is likely to continue to grow. We believe that the fact that such a widely-used and growing class of software often has faults should not be taken lightly.

To address this issue, we have developed a methodology that brings some of the benefits of formal testing and debugging methodologies to this class of software. Our methodology is tightly integrated into the spreadsheet environment, facilitating the incremental testing and debugging activities that occur during spreadsheet development. Our methodology also employs the visual feedback that is characteristic of spreadsheet environments, while presenting information in a manner that requires no knowledge of the underlying testing and fault localization theories.

Future work is planned along two dimensions. First, although our previous user studies have suggested that visual feedback about testing coverage helps users correct faults [11], we have not yet conducted a user study involving the methodology presented here. We are currently designing such a study.

Second, we expect user studies to reveal possibilities for new or refined debugging techniques. For example, we may wish to investigate the use of slicing and fault localization at the level of subexpressions: this may yield more precise results, but at additional cost. Another possible direction is to refine the user interface for our methodology. Our current interface limits the user to only the information that can be displayed on one screen. A more scalable approach, such as the one used in [3] for C programs, would allow users to scan through slicing and fault localization information for a large spreadsheet program without having to scroll within or switch between multiple worksheets.

Although our current research is with spreadsheet languages, we believe this methodology could be extended to other end user languages. The notion of fault likelihood used in this paper could be used in other environments in which slicing is available. However, availability of slicing is not sufficient for our methodology. In addition the slicing must be highly accurate, to avoid giving false indications that frustrate the user. Furthermore the environment must also support interactive editing of source code, editing of test cases, and validation of output, and must provide immediate feedback as to the effects of those actions. This combination of requirements suggests that our methodology is best suited for the highly interactive visual environments that have recently begun to emerge for end user programming.

The studies presented in [14, 22] showed that end users find that the act of debugging, and particularly the act of locating faults in long computation chains, is very difficult in spreadsheet programs. Our methodology attempts to alleviate this difficulty by providing the user with slicing and fault localization information. The goal of this work is to provide effective testing and debugging methodologies that help reduce the number of faults in spreadsheet programs and may also be helpful in other end user programming environments.

Acknowledgements

We thank the Visual Programming Research Group for their work on the Forms/3 implementation and for their feedback on the methodology. This work has been supported by NSF under ASC 93-08649, by NSF Young Investigator Award CCR-9457473, by Faculty Early CAREER Award CCR-9703108, and by ESS Award CCR-9806821 to Oregon State University.

References

[1] H. Agrawal and J.R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[2] J. Atwood, M. Burnett, R. Walpole, Wilcox E., and S. Yang. Steering programs via time travel. In *IEEE Symposium on Visual Languages*, September 1996.

[3] T. Ball and S.G. Eick. Visualizing program slices. In *Proceedings. IEEE Symposium on Visual Languages*, pages 288–95, October 1994.

[4] P. Brown and J. Gould. Experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems*, 5(3):258–272, July 1987.

[5] M. Burnett, J. Atwood, and Z. Welch. Implementing level 4 liveness in declarative visual programming languages. In *1998 IEEE Symposium on Visual Languages*, September 1998.

[6] M. Burnett and H. Gottfried. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction*, 5(1):1–33, March 1998.

[7] M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, and X. Yang. Toward visual programming languages for steering in scientific visualization: a taxonomy. *IEEE Computer Science and Engineering*, 1(4), 1994.

[8] T. Y. Chen and Y. Y. Cheung. On program dicing. *Software Maintenance: Research and Practice*, 9(1):33–46, January–February 1997.

[9] E. H. Chi, P. Barry, J. Riedl, and J. Konstan. A spreadsheet approach to information visualization. In *IEEE Symposium on Information Visualization*, October 1997.

[10] C. Cook, M. Burnett, and D. Boom. A bug's eye view of immediate visual feedback in direct-manipulation programming systems. In *Proceedings of Empirical Studies of Programmers*, October 1997.

[11] C. Cook, K. Rothermel, M. Burnett, T. Adams, G. Rothermel, A. Sheretov, F. Cort, and J. Reichwein. Does immediate visual feedback about testing aid debugging in spreadsheet languages. Technical Report TR 99-60-07, Oregon State University, March 1999.

[12] J.S. Davis. Tools for spreadsheet auditing. *International Journal of Human-Computer Studies*, 45(4):429–442, 1996.

[13] E. Duesterwald, R. Gupta, and M. L. Soffa. Rigorous data flow testing through output influences. In *Proceedings of the 2nd Irvine Software Symposium*, March 1992.

[14] D. G. Hendry and T. R. G. Green. Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model. *International Journal of Human-Computer Studies*, 40(6):1033–1065, June 1994.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[16] B. Korel and J. Laski. Dynamic slicing of computer programs. *The Journal of Systems and Software*, 13(3):187–195, November 1990.

[17] B. Korel and J. Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40(11–12):647–659, December 1998.

[18] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, May 1993.

[19] H. Lieberman and C. Fry. Zstep 95: A reversible, animated source code stepper. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1997.

[20] J.R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference, Computers and Applications*, pages 877–883, 1987.

[21] B. Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *ACM CHI '91*, pages 243–249, April 1991.

[22] B. A. Nardi and J. R. Miller. Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *International Journal of Man-Machine Studies*, 34(2):161–184, February 1991.

[23] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, 10(6), November 1984.

[24] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–84, April 1984.

[25] R. Panko and R. Halverson. Spreadsheets on trial: A survey of research on spreadsheet risks. In *Twenty-Ninth Hawaii International Conference on System Sciences*, January 1996.

[26] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[27] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, December 1994.

[28] B. Ronen, M.A. Palley, and H.C. Lucas. Spreadsheet analysis and design. *Communications of the ACM*, 32(1):84–93, January 1989.

[29] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), August 1996.

[30] G. Rothermel, L. Li, C. Dupuis, and M. Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *Proceedings of the 20th International Conference on Software Engineering*, pages 198–207, April 1998.

[31] S. Sinha, M.J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[32] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[33] G. Viehstaedt and A. Ambler. Visual representation and manipulation of matrices. *Journal of Visual Languages and Computing*, 3(3):273–298, September 1992.

[34] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.