**AN ABSTRACT OF THE THESIS OF**

James Reichwein for the degree of Master of Science in Computer Science presented on June 16, 2000.

Title: An Integrated Testing and Fault Localization Methodology for Spreadsheet Languages

Abstract approved: Signature redacted for privacy.

Margaret M Burnett

Spreadsheet languages, which include commercial spreadsheets and various research systems, have proven to be flexible tools in many settings. Research shows, however, that spreadsheets often contain faults. This thesis presents an integrated testing and fault localization methodology for spreadsheets. This methodology allows spreadsheet developers to engage in modeless development, testing and debugging activities. Furthermore, we provide an interface to our methodology that does not require an understanding of testing and debugging theory.

To accomplish this, we introduce the notion of *fault likelihood*: the likelihood that a given cell contains a fault that contributes to an known failure in the spreadsheet. To estimate fault likelihood we present five properties that we feel should govern its behavior. We then discuss our implementation of this methodology and illustrate its use.

# An Integrated Testing and Fault Localization Methodology for Spreadsheet Languages

by

James Reichwein

A Thesis

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed June 2000
Commencement June 2001

## ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Margaret Burnett, for her endless patience, good nature, and advice throughout my graduate work at Oregon State University. I would also like to thank Dr. Gregg Rothermel and Dr. Curtis Cook for their help and advice.

My thanks also go to the members of the Forms/3 Visual Programming Research Group for their work on the Forms/3 implementation and for their feedback on my work. Thanks especially to Andrei Sheretov for writing the incredibly useful, yet incredibly annoying, CRG data structure.

# TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

# LIST OF FIGURES

## LIST OF TABLES

# AN INTEGRATED TESTING AND FAULT LOCALIZATION METHODOLOGY FOR SPREADSHEET LANGUAGES

# Chapter 1 INTRODUCTION

Spreadsheet languages, which include commercial spreadsheet systems as a subclass, have proven useful in many settings, including business management, accounting, and numerical analysis. The spreadsheet paradigm is also a subject of ongoing research, including spreadsheet languages for matrix manipulation problems [41], for providing steerable simulation environments for scientists [6], 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 [28]. 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 [11]. 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, 23].

To begin to address these problems, previous work [33] presented a WYSIWYT (What You See Is What You Test) 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. An empirical study [34, 35] of this methodology has shown that participants using the WYSIWYT methodology were significantly more effective and efficient at testing than participants in a control group. Furthermore, participants using the WYSIWYT methodology were significantly less overconfident than participants in a control group. Additional work by Burnett, Sheretov, and Rothermel [7] addressed scalability issues with the WYSIWYT methodology when used with large grids. This involved storing testing information for regions of cells with similar formulas rather than for every cell in the grid.

This thesis[1] presents work done to integrate debugging support with the WYSIWYT methodology. This integrated methodology adds the ability to indicate which cells are incorrect for a given test case, and to view fault localization information inferred from both correct and incorrect marks. As in [7], debugging information is stored on the level of cell regions, allowing this new methodology to work with large grids. Key to the effectiveness of our approach is that it is tightly integrated into the spreadsheet environment, facilitating the

[1] Portions of the work described in this thesis have previously appeared in [30].

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.

# Chapter 2

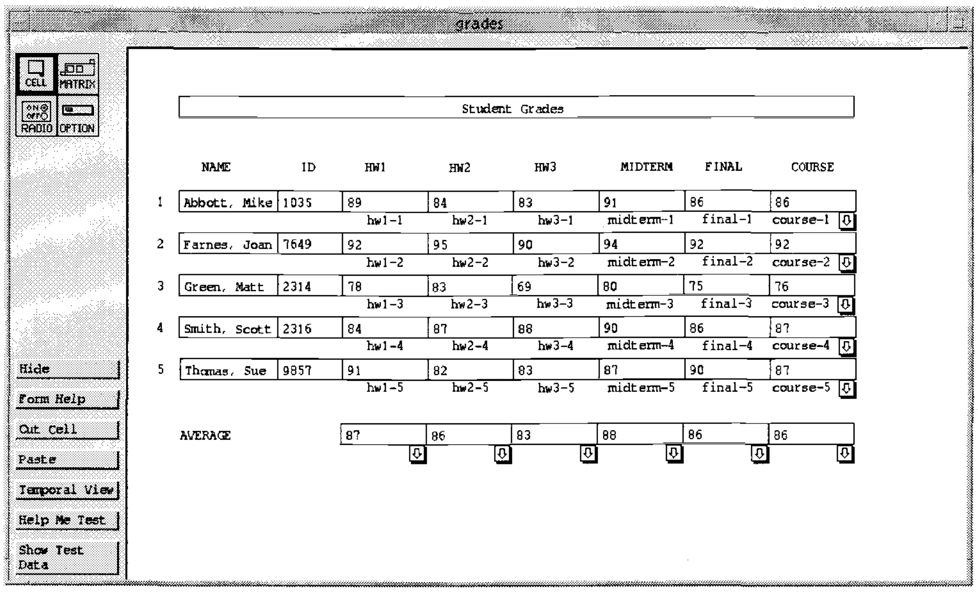
# BACKGROUND

## 2.1 WYSIWYT: A testing methodology for spreadsheets

The research in this thesis builds upon the What You See Is What You Test (WYSIWYT) methodology[33]. The underlying assumption in that methodology has been that, as the user develops a spreadsheet incrementally, he or she is also testing incrementally. A prototype implementation of this approach to incremental, visual testing has been implemented in the spreadsheet language Forms/3 [5]; the examples in this thesis are presented in that language.

Figure 2.1, shows a Forms/3 spreadsheet used to calculate grades. Like commercial spreadsheets, the value of a cell in Forms/3 is defined solely by its formula. However, unlike commercial spreadsheets, cells are not locked into a fixed grid, but can be laid out in whatever way the user desires. A side effect of this is that cells can no longer be referenced using grid locations such as "A1", therefore the methodology allows the user to give cells explicit names which are displayed underneath the cell's border. Additionally, Forms/3 provides formula tabs on the lower right hand corner of cells to allow more than one formula to be viewed at a time.

The WYSIWYT methodology is intended for spreadsheet developers, not software engineers. Thus, the methodology does not include specialized testing vocabulary – in fact, it includes no vocabulary at all, instead presenting testing-



Student Grades

| | NAME | ID | HW1 | HW2 | HW3 | MIDTERM | FINAL | COURSE |
|---|---|---|---|---|---|---|---|---|
| 1 | Abbott, Mike | 1035 | 89 | 84 | 83 | 91 | 86 | 86 |
| | | | hw1-1 | hw2-1 | hw3-1 | midterm-1 | final-1 | course-1 |
| 2 | Farnes, Joan | 7649 | 92 | 95 | 90 | 94 | 92 | 92 |
| | | | hw1-2 | hw2-2 | hw3-2 | midterm-2 | final-2 | course-2 |
| 3 | Green, Matt | 2314 | 78 | 83 | 69 | 80 | 75 | 76 |
| | | | hw1-3 | hw2-3 | hw3-3 | midterm-3 | final-3 | course-3 |
| 4 | Smith, Scott | 2316 | 84 | 87 | 88 | 90 | 86 | 87 |
| | | | hw1-4 | hw2-4 | hw3-4 | midterm-4 | final-4 | course-4 |
| 5 | Thomas, Sue | 9857 | 91 | 82 | 83 | 87 | 90 | 87 |
| | | | hw1-5 | hw2-5 | hw3-5 | midterm-5 | final-5 | course-5 |
| | AVERAGE | | 87 | 86 | 83 | 88 | 86 | 86 |

FIGURE 2.1: A Forms/3 spreadsheet used to calculate grades.

related information visually. Users test spreadsheets by trying different input values, and validating correct cells with a check mark. 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.

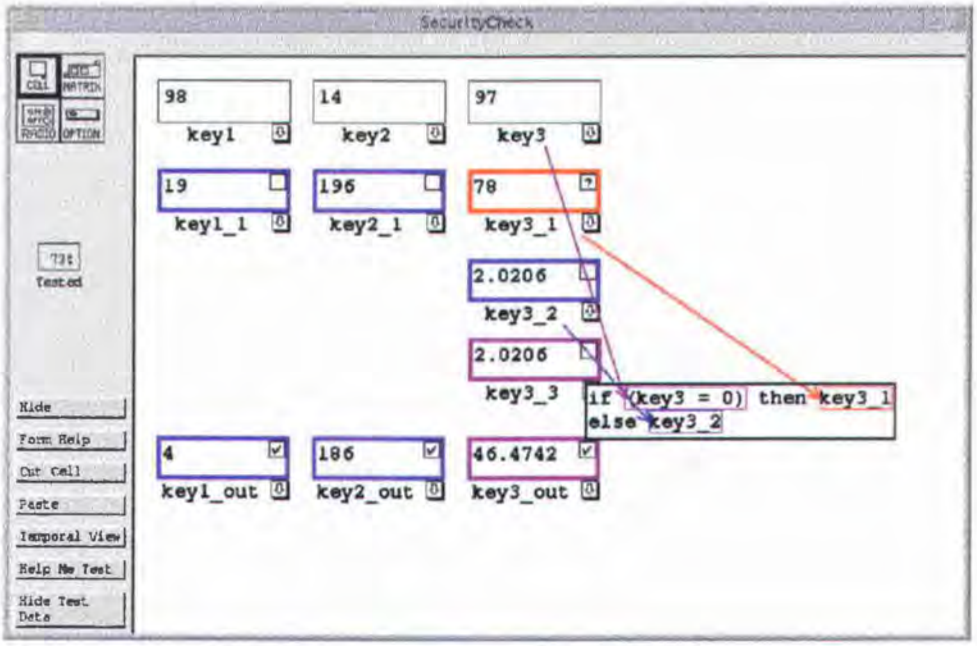
Although users of the WYSIWYT methodology need not realize it, they are actually using a dataflow test adequacy criterion [18, 25, 29] and creating *du-adequate* test suites. In the theory that underlies this methodology, a *definition* is a point in the source code where a variable is assigned a value, 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* [12] 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 [32] and the WYSIWYT methodology emphasizes the importance of retesting code affected by modifications.

Figure 2.2 illustrates the 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`,



```
if (key3 = 0) then key3_1
else key3_2
```

FIGURE 2.2: `SecurityCheck` spreadsheet with testing information displayed.

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 has validated the three output cells in this program. 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. Cell `key3_1` has not been tested at all, so it is red. Cells `key1_out`, `key1_1`, `key2_out`, `key2_1`, and `key3_2` have been completely tested, and have blue borders. The colors of displayed arrows between cells indicate the degree to which dependencies (interactions) between those cells have been validated.

Using this prototype we have recently conducted an empirical study to evaluate the usefulness of the WYSIWYT methodology. This study was performed using participants drawn from undergraduate and graduate Computer Science courses. The participants were divided into two groups. One group tested spreadsheets using Forms/3 with the WYSIWYT methodology enabled, and the other tested the same spreadsheets using Forms/3 without the WYSIWYT features. The test suites developed by these groups were then analyzed based on their effectiveness–measured by DU-coverage–and efficiency–measured by the number of redundant test cases and the speed of testing. Participants were also given questionnaires to determine whether they showed any overconfidence in the effectiveness of their test suites. The results of this study showed that participants using the WYSIWYT methodology tested with significantly more effectiveness and efficiency, and with significantly less overconfidence [34, 35].

## 2.2 Fault localization

In standard terminology [2], a *failure* is an incorrect computational result. A *fault* is a location in the program that results in a failure. Fault localization is, therefore, the process of finding the fault or faults that cause a failure. Several techniques have been proposed to automate this process. One such approach is *algorithmic debugging* [36, 24], which traverses an execution tree while asking the user to verify the correctness of each node. Myers [22] mentions several other fault localization techniques.

The fault localization technique described in this thesis is based on *Program Slicing* [42]. 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 $s$ [42]. *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, 26, 31, 39] 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 [40]. A more recent survey of dynamic slicing is given in [17].

A slice performed on an incorrect output variable can be used as a means for fault localization. Lyle and Weiser presented an alternate method for fault localization using slicing called *program dicing* [19]. Whereas a slice makes use only of information on incorrect variables at failure points, a dice also makes use of information on correct variables. Dicing works by subtracting the slices on correct variables away from the slice on the incorrect variable, resulting in a smaller set of cells than in 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 [19]:

> **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. For example, it is unlikely that every correct variable depends on a fault; if a dicing algorithm subtracts the intersection of all slices of correct variables from that of the incorrect variable, the result is less likely to miss a fault. In addition, Chen and Cheung explore dynamic dicing, which uses dynamic slices instead of static slices, in order to avoid including statements that were not executed by the test suite. In similar work, Pan and Spafford [27], introduce 20 heuristics for combining dynamic and static slices to aid the programmer in fault localization.

# Chapter 3

# FAULT LOCALIZATION IN FORMS/3

We have designed a fault localization methodology for Forms/3 that is targeted for spreadsheet developers. With this audience in mind, we have observed the following constraints in designing our methodology.

1. **Our methodology must offer modeless operation.** 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 get immediate feedback from their actions. 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.

2. **Our methodology must not rely on terminology from testing or 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.

3. **Our methodology must avoid false indications.** While computer specialists may be able to understand the limitations of their tools, spreadsheet developers are not liable to understand the reasons if debugging feed-

back leads them astray. If our methodology highlights a set of cells that does not include a fault, these users are likely to become frustrated, and to become disillusioned with our methodology. Therefore, our methodology must avoid giving false indications of faults where no faults exist.

In this chapter, we present the high level design of our methodology and describe how it meets the above constraints. In Chapter 4, we will go into detail about the data structures and algorithms used to implement our methodology.

## 3.1 Integrating Testing and Fault Localization with the Spreadsheet Paradigm.

The previous WYSIWYT methodology met our first design constraint by allowing the user to place check marks during any stage of spreadsheet creation, and by handling formula edits to correctly keep track of testedness information in the light of new formulas. Furthermore, the user can undo check marks if they change their decision about the correctness of a value, or if they wish to see exactly what changes in colored cell borders were caused by the check mark. The WYSIWYT methodology met our second constraint through the use of colored cell borders to avoid using terminology from testing theory. Similarly, the integrated testing and fault localization methodology presented in this thesis meets our first two constraints by supporting the following user actions.

- **Marking Cell Values as Correct or Incorrect.**

  In addition to the check marks the WYSIWYT methodology used, the user can now place X marks on cells. The check mark indicates that a cell's value is correct, and an X mark indicates that a cell's value is incorrect. The user is also able to undo X and check marks.

To avoid testing or fault localization terminology, colors are used to give testing and fault localization information to the user. Testing information is displayed by the color of the cell border, while fault localization information is indicated by the color of the background of a cell. How these colors are chosen is discussed later in this chapter.

- **Changing the Current Test Case**

  In software engineering terminology, a test case is a single set of inputs to a program. We define the inputs of a spreadsheet to be all of the constant cells in that spreadsheet. We define a constant cell as any cell whose formula does not refer to other cells and does not contain an "if" expression.

  For our methodology to be modeless, it must keep testing information consistent with changes in test cases. To accomplish this, a change in an input value causes all check and X marks on affected cell values to be removed. This is because the values on those cells need to be re-checked for the new test case. Note that the cumulative effects on testing and fault localization information stored for these cells is not removed, rather it is continually added to as the user continues the testing process. From the user's perspective, changing a test case removes the marks, but does not change the colors that result from the marks. The only time testing and fault localization information is removed is when the user changes the spreadsheet's logic.

- **Changing the Spreadsheet's Logic**

  We consider the changing of a non-constant cell's formula to be the same as a change in source code. As such any testing and fault localization information affected by the change is no longer applicable to the new program and must be removed. As with changing a constant cell, this requires that all check and X marks on affected cell values be removed. However, it also requires that the effects these marks have on testing and fault localization information be undone. From the user's perspective, both the marks and colors are removed.

  These changes affect only the testing and fault localization information affected by the edited cell. This is important in meeting our constraint that our methodology be modeless. When a change in logic is made, the user does not have to retest the entire spreadsheet, only the affected portions.

## 3.2 Fault Likelihood

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 the 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 $I$.

## 3.3 Representing Fault Likelihood as a Color

The WYSIWYT methodology uses a red-blue continuum for borders against the white background used for spreadsheets. This choice has several important consequences. First, red and blue are easily distinguished [10]. Second, because of the physiology of the human eye, red stands out while blue recedes [38], which furthers our goal of attracting the user's attention to untested cell. Third, because blue and red differ in two of the three RGB components of screen color, the red-blue continuum should be usable by some red-deficient or blue-deficient users [20][1].

[1] Both our prototypes of the WYSIWYT methodology and of the methodology described by this thesis support using grayscale colors for users who have difficulty distinguishing colors.

Additionally, the WYSIWYT methodology uses a quadratic formula that separates the colors used to represent 0%, 100%, and intermediate percent values of testedness. However, under the WYSIWYT methodology the user is unlikely to be able to discern different levels of partial testedness.

In designing a fault localization methodology, we chose red to represent the fault likelihood of a cell, as red stands out well and is often used to indicate that something is wrong, or that something requires immediate attention (e.g. a warning light in a car or a stop sign). This choice, however, meant that we could not use the red-blue continuum of the WYSIWYT methodology without giving red two inconsistent meanings. To avoid this, we decided to change the colors used on cell borders to indicate testedness. Instead of ranging from red for untested, to blue for tested, the cell borders now range from black for untested, to blue for fully tested. Black was chosen as it is the default color for borders around cells, and therefore a black border can be interpreted as a cell containing no additional testedness. Another option would be to introduce new colors for the testedness border. However, we preferred to keep the number of colors used to a minimum, as too many colors can overwhelm the user [38]. Shades of purple are used to represent partially tested cells, and again we are careful to make sure that all of the shades of purple used to represent partial testedness can be easily distinguished from the blue used to represent complete testedness, and the black used to represent no testedness.

The fault likelihood of a cell is shown by using a shade of red for the background color of the cell. The shades vary from white, representing no fault likelihood, to pink, representing minimal fault likelihood, to bright red, representing maximum fault likelihood. We are careful to make sure that it is easy to distinguish the lightest shade of pink from white.

Another concern is how easy it is for the user to distinguish the different colors representing fault likelihood. To address this, we decided to limit the number of colors used to five distinct shades of pink or red. We did this for two reasons. First, we have no good way of mapping fault likelihood to a numeric scale. For example, it makes no sense to say that something has 75% fault likelihood. This makes it difficult to find a mathematical function to map fault likelihood to a range of colors, as we do with testedness. Second, we chose a small number of colors, to avoid having the user need to distinguish between very close shades of color, as such shades may not be distinguishable on some monitors, or by some users. The five colors we chose vary in their red components, as well as their blue and green components, which should allow some users with red-deficient vision to distinguish the different shades.

## 3.4 Estimating Fault 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 the properties described below, which rely on the definitions in Table 3.1.

**Property 1** *If* $\mathit{IncorrectDependentMarks}(C) \neq \phi$ *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. Hence every cell that could possibly be contributing to a failure is highlighted, reducing the chance that our methodology will highlight a set of cells that does not contain a fault. However, there are still two situations in which the highlighted cells might not include a fault

| | |
|---|---|
| *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 *Dynamic-Predecessors*$(C)$. |
| *DynamicForwardSlice*$(C)$ | The transitive closure on *Dynamic-Successors*$(C)$. |
| *IncorrectDependentCells*$(C)$ | The set of cells in *DynamicForwardSlice*$(C)$ that have been marked incorrect for the *current* test case. |
| *CorrectDependentCells*$(C)$ | The set of cells in *DynamicForwardSlice*$(C)$ that have been marked correct for the *current* test case. |
| *IncorrectDependentMarks*$(C)$ | The set of marks that have been in *DynamicForwardSlice*$(C)$ for *any* test case. |
| *IncorrectDependentMarks*$(C)$ | The set of marks that have been in *DynamicForwardSlice*$(C)$ for *any* test case. |

TABLE 3.1: The definitions used in determining fault likelihood. Here $S$ is a spreadsheet, and $C$ is any cell in $S$.

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.

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 attempt to highlight cells with a higher likelihood of containing a fault a brighter shade of red than those with a lower likelihood of containing a fault.

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

**Property 3** *The fault likelihood of C is inversely proportional to |CorrectDependentMarks(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 there is no path along dynamic data flow from a cell B to a correct mark on cell D that does not travel through a cell in IncorrectDependentCells(B), then the mark on cell D is said to be blocked with respect to cell B. The mark on cell D has no effect on cell B.*[2]

[2] In [30] we presented six properties of fault likelihood, but later decided to remove the fourth property as it added an inconsistent meaning to the background colors of cells. Properties 4 and 5 in this this thesis correspond to properties 5 and 6 in [30].

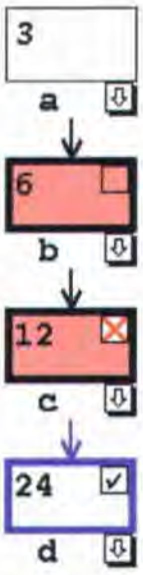


FIGURE 3.1: An example of a check mark being blocked by an X mark, as described in Property 4.

This property is relevant when a correct cell value depends on an incorrect cell value, as shown in Figure 3.1. 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.

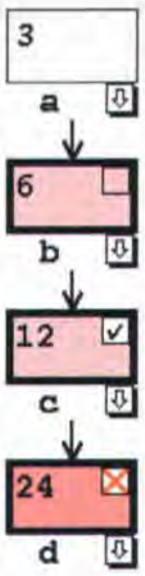


FIGURE 3.2: An example of an X mark being blocked by a check mark, as described in Property 5.

**Property 5** *If there is no path along dynamic data flow from a cell B to an incorrect mark on cell D that does not travel through a cell in CorrectDependentCells(B), then the mark on cell D is said to be blocked with respect to B. The mark on cell D has no effect on cell B, except for the minimum fault likelihood required by Property 1.*

This property is relevant when a value marked incorrect depends on a value marked correct, as shown in Figure 3.2. In dicing, the dynamic backward slice of the correct value would be completely subtracted from that of the incorrect value. However Property 1 requires that we 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.

## 3.5 Example

The following example illustrates the integrated testing and fault localization methodology described in this thesis.

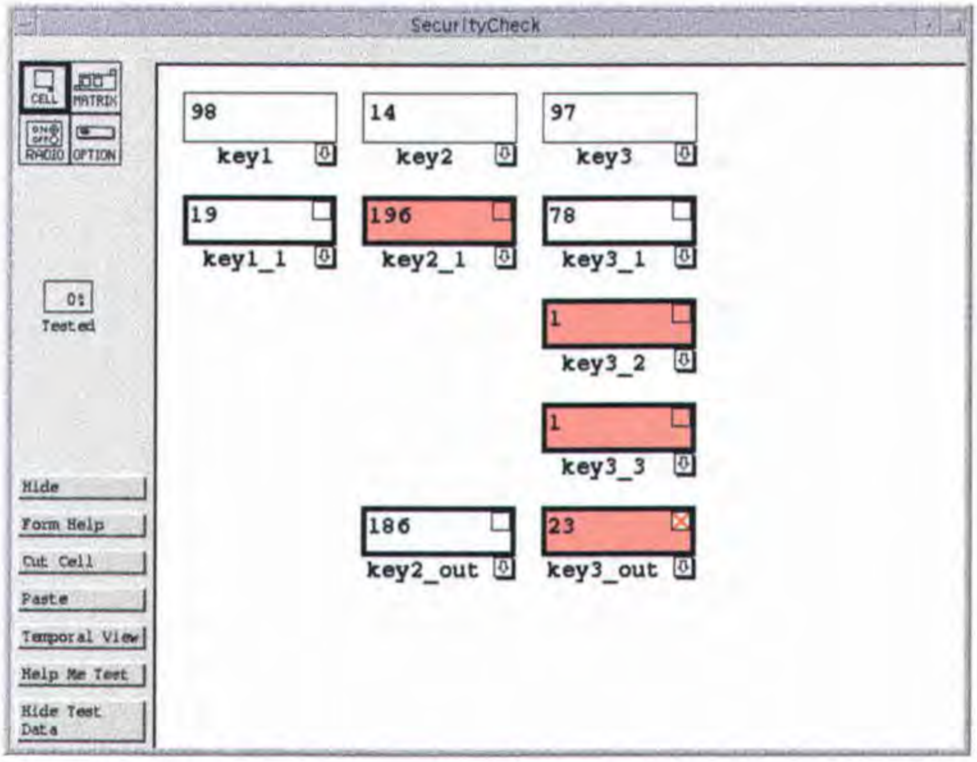


FIGURE 3.3: `SecurityCheck` spreadsheet at an early stage of development. The user has noticed a failure in cell `key3_out` and has marked it incorrect.

### *3.5.1 Marking Cells Correct or Incorrect*

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 3.3. 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.

As soon as an incorrect output is noticed, the user can place an "X" mark in a cell to indicate that it has an incorrect value. In Figure 3.3 the user has placed 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, as shown in Figure 3.3. In keeping with Property 1, the highlighted cells are those contained in the backward dynamic slice of `key3_out`.

Now suppose the user notices that the cell `key2_out` is correct. The user can now mark it with a check mark. The result of doing so is shown in Figure 3.4. Now cell `key2_1` contains a lighter shade of red than before. This is in keeping with Property 2 because it now contributes to a correct cell value.

### *3.5.2 Choosing a Different Test Case*

Suppose the user developing the `SecurityCheck` 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 3.5 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 reachable correct dependent for this test case and a reachable incorrect dependent for the previous test case. This

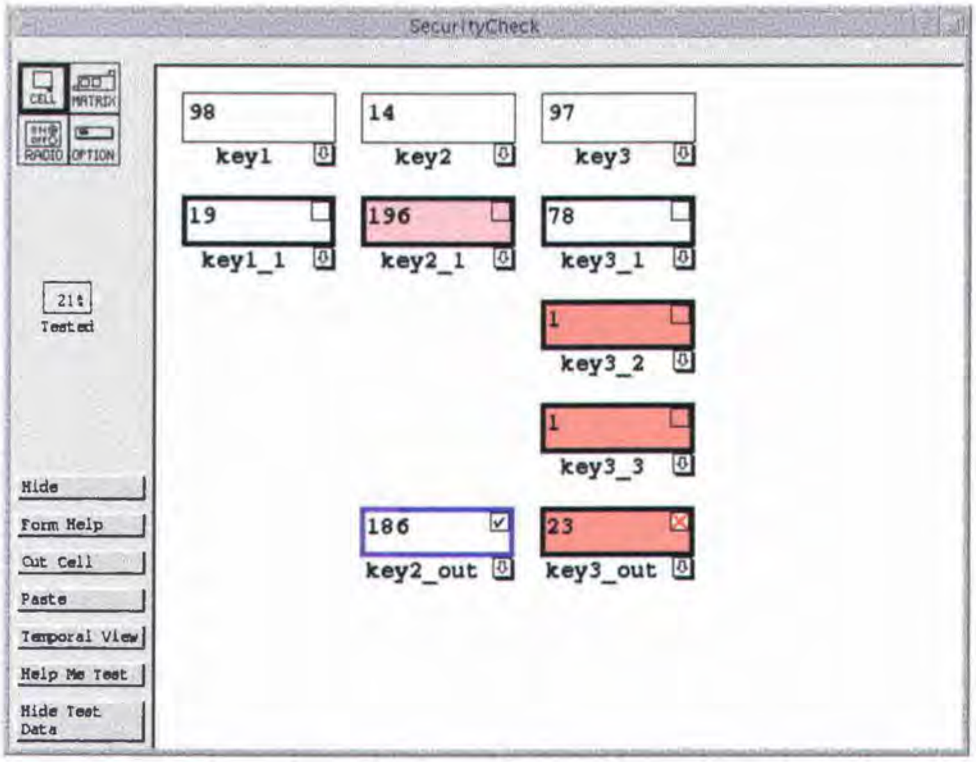


FIGURE 3.4: `SecurityCheck` spreadsheet following additional validation.

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.

### *3.5.3 Fixing the Fault*

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`",

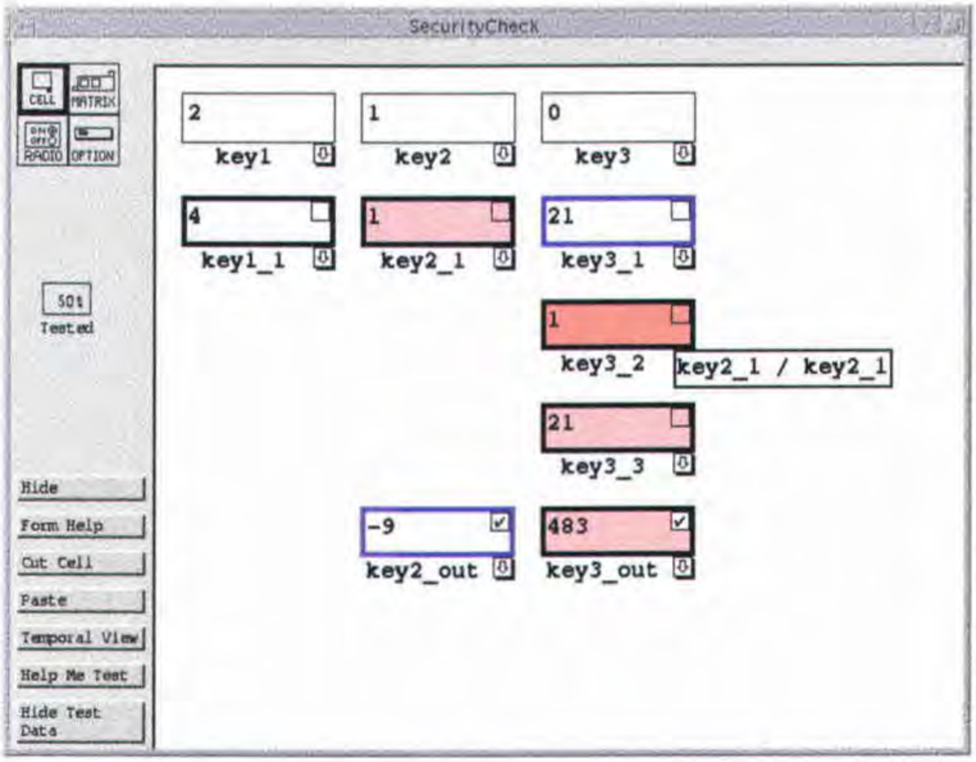


FIGURE 3.5: `SecurityCheck` spreadsheet following application of additional test cases.

to "`key2_1 / key3`". Figure 3.6 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 X mark previously placed on `key3_out` in Figure 3.3 becomes out of date. Not only has it been removed, but its effects on testing and debugging information were also undone. This feedback is intended to encourage the user to perform regression testing on the cells affected by the change. However, notice that the value of cell `key3_out` did not change when the fault was fixed. This is because cell `key3_2` is not in the backward dynamic slice of cell `key3_out`. Since this value is still correct for the current test case, the mark that was placed there is not removed.

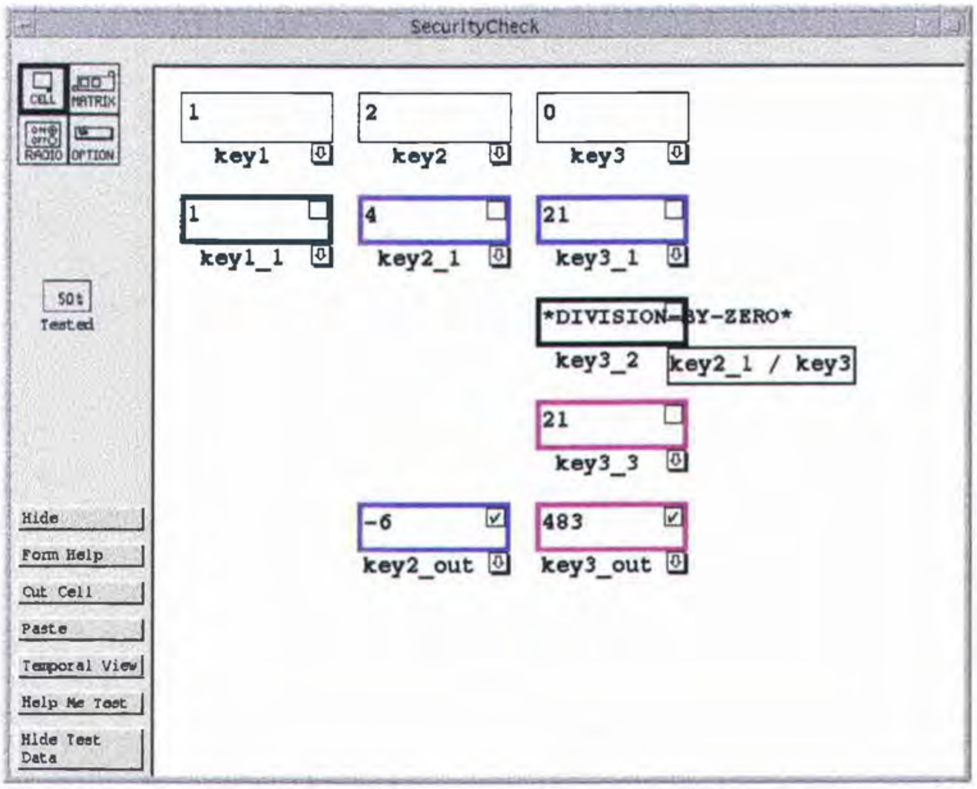


FIGURE 3.6: Corrected SecurityCheck spreadsheet.

# Chapter 4

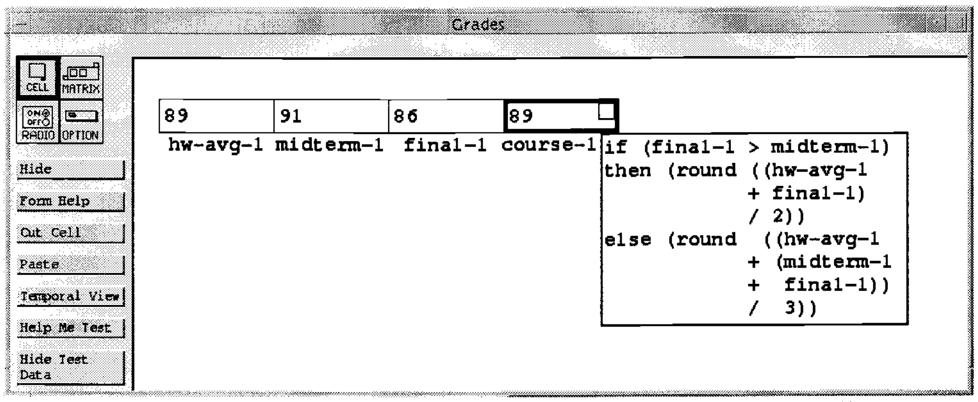
# IMPLEMENTATION AND COMPLEXITY ANALYSIS

We have implemented a prototype of the fault localization methodology described in Chapter 3. This chapter describes the data structures and algorithms used to implement this prototype and presents an analysis of the run time complexity of these algorithms.

## 4.1 Data Structures

### *4.1.1 Cell Relation Graph*

The Cell Relation Graph (CRG) is an abstract model for spreadsheets [33], and forms the basis for the implementation of the fault localization methodology described in Chapter 3. Figure 4.1 shows the CRG for a portion of the `Grades` spreadsheet shown in Chapter 2. A CRG consists of *formula graphs*, shown as dotted rectangles, which are composed of subexpression nodes, shown as solid circles and rectangles. Edges within a formula graph represent the flow of control within cell formulas, and are represented by solid lines. *Entry* and *exit* nodes, labeled with "E" and "X" respectively, represent where the evaluation of the formula begins and ends. *Predicate nodes* are shown as solid rectangles, and represent conditional expressions. Edges coming from predicate nodes are labeled with the value to which the conditional expression must evaluate for



| 89 | 91 | 86 | 89 |
|---|---|---|---|
| hw-avg-1 | midterm-1 | final-1 | course-1 |

```
if (final-1 > midterm-1)
then (round ((hw-avg-1
              + final-1)
              / 2))
else (round  ((hw-avg-1
              + (midterm-1
              +  final-1))
              /  3))
```

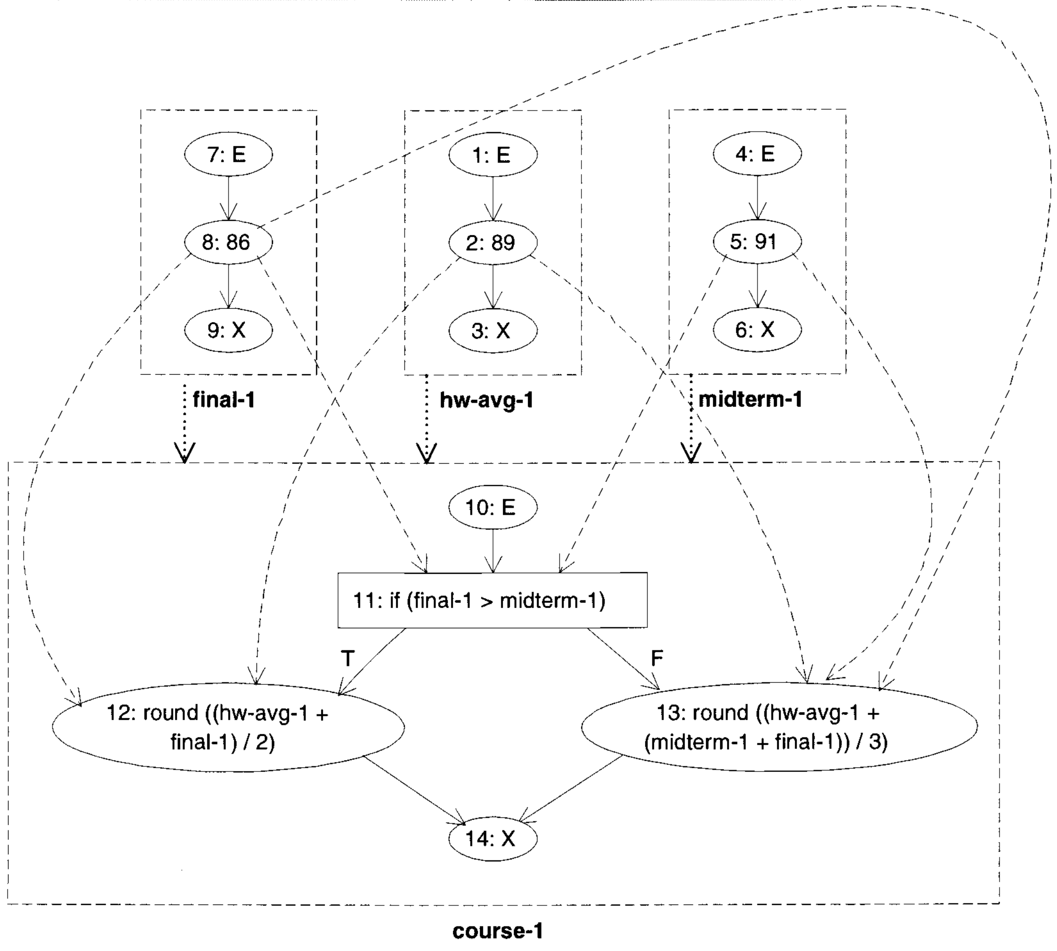

7: E
8: 86
9: X
final-1
1: E
2: 89
3: X
hw-avg-1
4: E
5: 91
6: X
midterm-1
10: E
11: if (final-1 > midterm-1)
T
F
12: round ((hw-avg-1 + final-1) / 2)
13: round ((hw-avg-1 + (midterm-1 + final-1)) / 3)
14: X
course-1

FIGURE 4.1: The CRG for a portions of a grades spreadsheet.

control to follow that edge. All other nodes are *computation* nodes, and are represented as circles. Data flow dependencies between subexpression nodes are represented by *du-pairs*, which are shown by dashed lines. Data flow dependencies between formula graphs are shown by *cell-dependence edges*, shown by dotted lines.

In collaboration with others in the Forms/3 group, we have implemented the Cell Relation Graph as an internal data structure within Forms/3 [37]. In this section we give an overview of the interface provided by the CRG data structure. Following sections describe how we added fault localization information to this data structure.

Below we describe the classes that we use to represent the CRG within Forms/3. Note that we are describing all algorithms and data structures in this paper using a pseudocode language based on C++ and Java. A commonly used facility in our implementation is the iteration over a collection of items. In C++ and Java, the syntax for this operation can vary depending on how the collection of objects is stored. To avoid this low level detail, we adopt the syntax `list<Type>` to represent a list of objects of type `TYPE`. The only requirement on these lists that we impose is that they support iteration through $n$ items in O($n$) time.

```
class CRGNode
{
    public void NewFormula(Formula Formula);
    public void Recomputed();
    public void MarkPlaced(Mark mark, bool do);

    public void updateGUI();

    public list<CRGNodes> GetAffectingCRGNodes();

    public FormulaGraph GetFormulaGraph();
    public CRGTracer GetTracer();
}
```

```
class FormulaGraph
{
    public list<DUPair> EnumIncomingDUPairs();

    public Testedness GetTestdness();
    public FGNode GetFirstNode();
}

abstract class FGNode
{
    public list<DUPair> EnumIncomingDUPairs();
}

class ComputationNode extends FGNode
{
    public list<DUPair> EnumOutGoingDUPairs();
}

class PredicateNode extends FGNode
{
    public FGNode GetTrueBranch();
    public FGNode GetFalseBranch();
}

class DUPair
{
    public ComputationNode GetDefNode();
    public FGNode GetUseNode();
    public Testedness GetTestedness();
    public void Validate();
    public void UnValidate(int n);
}
```

A `CRGNode` represents a cell in the CRG. The method `NewFormula()` is called when a cell is given a new formula, and is passed a parsed representation of that formula. The method `Recomputed()` is called when the value of the cell has been recomputed. `MarkPlaced()` is called when a check or X mark has been placed on a cell. The `updateGUI()` method is called in order to update the GUI with new testing and debugging information. The `GetAffectingCRGNodes()` method returns the `CRGNodes` for all of the cells that are referenced by the `FormulaGraph` of the `CRGNode`. A transitive closure of this method would be a backward static slice. The `GetFormulaGraph()`

method returns the FormulaGraph associated with the cell. Subexpression nodes within the FormulaGraph are represented by the subclasses of FGNode. All FGNodes can return a list of the du-pairs entering that node. These correspond to places in the subexpression that reference other cells. Computation nodes are places where the cell's value can be defined, and therefore can also return a list of outgoing du-pairs. The FormulaGraph::GetFirstNode(), PredicateNode::GetTrueBranch(), and PredicateNode::GetFalseBranch() methods are used to traverse the formula graph.

The class DUPair represents one or more du-pairs. Note that it is possible for a single subexpression to reference another cell twice, therefore it is possible for there to be more than one du-pair between two FGNodes. We model this by using a single instance of the DUPair class that represents more than one du-pair. The testedness of a DUPair is defined as the number of validated du-pairs it represents, divided by the total number of du-pairs it represents. This information is encapsulated in the Testedness object returned by the GetTestedness() method of DUPair. The testedness of an entire FormulaGraph is defined to be the number of validated du-pairs entering the formula graph, divided by the total number of du-pairs entering the formula graph. This value is returned by the GetTestedness() method of FormulaGraph, and is the value used to determine the color used for cell borders.

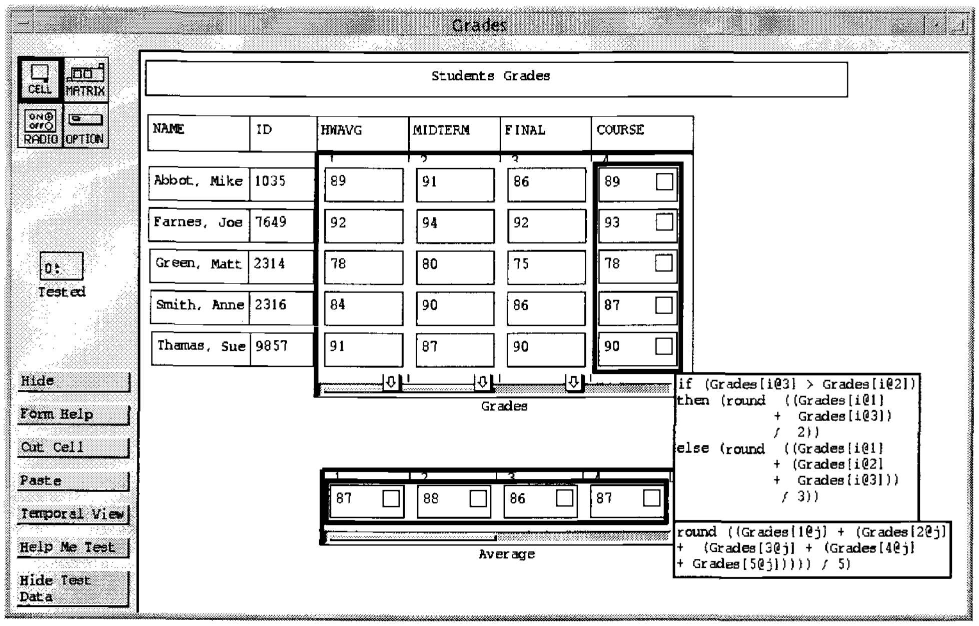
In addition to the static dependence information stored in the CRG, we also require dynamic dependence information to determine which du-pairs are currently exercised. The CRGTracer class is responsible for maintaining this information.

```
class CRGTracer
{
    public void StartTrace();
    public void TracePredicate(bool value);
    public void StopTrace();

    public list<CRGNodes> GetAffectingCRGNodes();

    public void ValidateExecutedDUPairs();
    public void UnvalidateExecutedDUPairs(int n);
}
```

This class stores the execution trace of a formula graph. The `StartTrace()`, `TracePredicate()`, and `StopTrace()` methods are called by the Forms/3 evaluation engine as the formula is evaluated. These are used by the `CRGTracer` to keep track of what `FGNodes` have been evaluated. The `GetAffectingCRGNodes()` method returns all the `CRGNodes` that directly affect the `FGNodes` in the execution trace stored by the `CRGTracer`. The transitive closure of this method would be a backward dynamic slice on the `CRGNode` associated with the `CRGTracer`. The `ValidateExecutedDUPairs()` and `UnvalidateExecutedDUPairs()` methods are described later in this chapter.



Students Grades

| NAME | ID | HWAVG | MIDTERM | FINAL | COURSE |
|---|---|---|---|---|---|
| Abbot, Mike | 1035 | 89 | 91 | 86 | 89 |
| Farnes, Joe | 7649 | 92 | 94 | 92 | 93 |
| Green, Matt | 2314 | 78 | 80 | 75 | 78 |
| Smith, Anne | 2316 | 84 | 90 | 86 | 87 |
| Thamas, Sue | 9857 | 91 | 87 | 90 | 90 |

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 87 | 88 | 86 | 87 |

```
if (Grades[i@3] > Grades[i@2])
then (round  ((Grades[i@1]
             +  Grades[i@3])
             /  2))
else (round  ((Grades[i@1]
             + (Grades[i@2]
             +  Grades[i@3]))
              / 3))
```

```
round ((Grades[1@j] + (Grades[2@j]
+  (Grades[3@j] + (Grades[4@j]
+ Grades[5@j])))) / 5)
```

FIGURE 4.2: A Forms/3 spreadsheet containing a matrix.

### *4.1.2 Region Representative Approach*

A *grid* is a two-dimensional matrix of cells. Most commercial spreadsheet systems are based entirely on grids. A *homogeneous grid* is one in which many cells share the same formula, or have formulas that are identical except for row/column indices in cell references. These similar formulas could be created by the user copying a formula along a row or column, or could be stored as a single formula for many cells. Forms/3 uses the latter approach. Figure 4.2 shows such a grid in a Forms/3 spreadsheet used to calculate student grades. The `Grades` cell in this spreadsheet is a Forms/3 *matrix*. The `Course` column in this matrix is composed entirely of a *region*. All of the cells in a region share the same formula. The pseudo-constants `i` and `j` can be used in region formulas to refer to the row and column number of the actual cell. This allows the cells in the `Course` region to share the same formula, but have different values.

Homogeneous grids present a problem both for the efficiency of the CRG, and for the user trying to test the grid. If each cell in a region stores its own formula graph, then a grades spreadsheet for 1,000 students would need to store 1,000 formula graphs, even though the formula graphs would be almost identical. Furthermore, in order for the user to turn all the cell borders blue, he or she would need to enter 1,000 identical test suites to test each row of the spreadsheet, even though the formula for each row is nearly identical.

To solve these problems, Sheretov, et al. introduced the *Region Representative* approach [37, 7]. In this approach, only one `FormulaGraph` is stored per region. However, because every cell in a region can have a different value, there is still a single `CRGTracer` for every cell. As a result, testedness information is stored for the entire region, and each cell in a region can be considered a test case for that region's formula. With the Region Representative approach, the user needs to enter only one set of test cases to test the formula for the Course region. Furthermore the system needs to store only one `FormulaGraph` object, whether there are 1, 50, or 1,000 students in the class.

To implement the Region Representative approach, the `CRGNode` is made an abstract class and three subclasses are created, `CellCRGNode`, `RgnCRGNode`, and `EltCRGNode`. A normal cell has a `CellCRGNode`, which has both a `FormulaGraph` and a `CRGTracer`. A Region has a `RgnCRGNode`, which has a `FormulaGraph`, but no `CRGTracer`. A cell inside of a region has a `EltCRGNode`, which has a `CRGTracer` but no `FormulaGraph`. The `EltCRGNode` delegates the `NewFormula()` and `GetFormulaGraph()` methods to its parent `RgnCRGNode`. It is an error to call the `GetTracer()` method of a `RgnCRGNode` object. Table 4.1 illustrates the relationships between these classes.

| | `FormulaGraph` | `CRGTracer` |
|---|---|---|
| `CellCRGNode` | **Has-a** | **Has-a** |
| `RgnCRGNode` | **Has-a** | |
| `EltCRGNode` | | **Has-a** |

TABLE 4.1: The relationships between different CRGNode subclasses and their components.

### *4.1.3 Reaching Mark Lists*

To add the fault localization information needed for this thesis to the CRG, we needed to keep track of the number of paths through which a particular mark reaches a cell, and whether a mark is blocked by another mark. To accomplish this, we use the `Mark` class to encapsulate a check or X mark, and a `ReachingMark` class to associate a number of paths with a mark.

```
class Mark
{
    private int MarkID;
    private int MarkType;
    private CRGNode MarkedNode;
    private BlockList BlockedMarks;
    private bool isCurrent;

    public int GetMarkID();
    public int GetMarkType();
    public CRGNode GetMarkedNode();

    public void SetBlockList(BlockList);
    public BlockList GetBlockList();

    public bool isCurrent();
    public void bool setIsCurrent(bool);

    public bool blocks(Mark m);
}

class ReachingMark
{
    public Mark mark;
    public int paths;
}
```

Each `Mark` has a unique `MarkID`, which identifies that mark. Each mark also has a `MarkType`, which identifies whether the mark is a check mark or an X mark. Additionally, each `Mark` knows what `CRGNode` it was placed on, and whether it was placed during the current test case. The `blocks(Mark m)` method returns true if the given mark `m` would be blocked by this mark if the value of the cell marked with `m` depended on the value of the cell marked by this `Mark`. This method returns true only if both marks are of a different type, and both marks were placed during the current test case.

A `ReachingMark` associates a `Mark` with a number of paths. `ReachingMarks` are stored in three types of lists. `BlockLists` store `ReachingMarks` that contain the number of paths along which a `Mark` has been blocked. A `Blocklist` imposes the constraint that `ReachingMark.paths` must be $> 0$. If this constraint is violated, the offending `ReachingMark` is removed from the `BlockList`. `DeltaLists` store `ReachingMarks` that contain the change in the number of paths along which a `Mark` reaches a cell. A `DeltaList` imposes no constraints on the `ReachingMarks` it contains. `ReachingMarkLists` store `ReachingMarks` that contain the number of paths along which a `Mark` reaches a cell. A `ReachingMarkList` imposes the constraint that `ReachingMark.paths` must be $\geq 0$. If this constraint is violated, the offending `ReachingMark` has its `paths` member variable set to 0. When a ReachingMark within a ReachingMarkList has 0 paths, it is considered to be blocked. Additionally the ReachingMarkList class keeps tracks of four integers, shown below.

```
class ReachingMarkList extends RMList
{
    private int numBlockedCheckMarks;
    private int numBlockedXMarks;
    private int numReachingCheckMarks;
    private int numReachingXMarks;

    public int CalculateFaultLikelihood();
    . . .
}
```

The CalculateFaultLikelihood() method uses these integers to estimate the fault likelihood of a cell. Keeping track of them in the ReachingMarkList allows the fault likelihood to be calculated without having to iterate through the ReachingMarkList and count which marks are blocked and which marks are not blocked.

The functionality that is common to BlockLists, DeltaLists, and ReachingMarkLists is factored out into a common superclass, shown below. This superclass is responsible for maintaining a list of ReachingMarks, supporting various operations on that list, and making sure that no two ReachingMarks within the same list refer to the same Mark.

```
abstract class RMList
{
    protected list<ReachingMark> theList;

    protected template_add(ReachingMark rm, int paths);

    public void Add(Mark m, int paths);
    public void Add(ReachingMark rm);
    public void Add(ReachingMarkList rml);

    public void Sub(Mark m, int paths);
    public void Sub(ReachingMark rm);
    public void Sub(ReachingMarkList rml);
    public ReachingMark Remove(Mark m);
    public ReachingMark Remove(ReachingMark rm);
    public list<int> Remove(ReachingMarkList rml);


    public ReachingMark Find(Mark mark);
    public ReachingMark Find(ReachingMark aRM);

    public void Clear();
    public RMList Copy();

    public int Size();

    public void DoBlock(Mark mark);
    public void BlockRM(ReachingMark rm);

}
```

The `RMList::Add(ReachingMark)` supports adding a `ReachingMark` to the `RMList`. If the argument `ReachingMark` matches the `Mark` of a `ReachingMark` that has already been stored in the list, their `paths` variables are added together, and the result is stored in the original `ReachingMark`. The `RMList::Add(Mark, int)` method is a short cut for calling the `RMList::Add(ReachingMark)` method with a newly created `ReachingMark`. The `Add(ReachingMarkList rml)` method is a short cut for calling the `RMList::Add(ReachingMark)` method with every `ReachingMark` within `rml`.

The `Sub()` methods behave the same as the `Add()` methods, but the `paths` variables of `ReachingMarks` are subtracted instead of added. In the case where the `ReachingMark` is not already in the list, the number of paths is subtracted from zero.

The `Remove(Mark)` method removes the `ReachingMark` that corresponds with the given `Mark`, and returns the `ReachingMark` that was removed (or null if no corresponding `ReachingMark` was found). The `Remove(ReachingMark rm)` method is a short cut for calling `Remove(rm.mark)`. The `Remove(ReachingMark-`

List rml) method is a short cut for calling Removes(ReachingMark) on all the elements of rml. The Remove(ReachingMarkList rml) method returns a list of two integers. The first is the total number of ReachingMarks removed that contained check marks. The second is the total number of ReachingMarks removed that contained X marks. The Find methods return the ReachingMark within the RMList that corresponds to the method's argument. The DoBlock() and BlockRM() methods are described in Section 4.2.1.

To support subclassing, the RMList::Add() and RMList::Sub() methods call the template_add() method to do the actual addition or subtraction. The subclasses of RMList can override template_add() to impose additional constraints on the ReachingMarks stored within the RMList.[1] Similarly, the Remove() methods all call the Remove(Mark) method to do the actual removal, allowing subclasses to override the Remove(Mark) method if needed.

Given these RMList subclasses, we can now show what we have added to the CRGNode class to keep track of fault localization information.

```
abstract class CRGNode
{
   . . .
   public Mark GetCurrentMark();
   public void SetCurrentMark(Mark m);
   public ReachingMarkList GetReachingMarkList();
   public void SetReachingMarkList(ReachingMarkList rml);
   public DeltaList GetTempDeltaList();
   public void SetTempDeltaList(DeltaList dl);
}
```

The GetCurrentMark() method returns the Mark that has been placed on the cell owning that CRGNode. The GetReachingMarkList() method returns the ReachingMarkList for the cell owning that CRGNode. Instances of class

[1] This is an application of the Template Method design pattern [13].

EltCRGNode will delegate this to their parent RgnCRGNode. Thus, as with testing information, all cells in a region share the same fault likelihood. The GetTempDeltaList() method returns a temporary DeltaList that is valid only while the algorithm for placing a mark is executing. This DeltaList is *not* shared by all cells in a region.

## 4.2 Algorithms and Complexity Analysis

In Section 3.1, we described the three operations the user can perform in our integrated testing and fault localization methodology. In this section we describe the algorithms that implement these operations, as well as the algorithm we use for determining an estimate of fault likelihood.

### *4.2.1 Placing a Mark*

The method CRGNode::MarkPlaced() is called by the user interface when a mark is placed on a cell. In our user interface, left clicking in the check box in the upper right hand corner of a cell places a check mark, while right clicking places an X mark. If the user places the same type of mark twice on the same cell, this is considered an undo operation, and the do argument to CRGNode::MarkPlaced() is set to false.

```
public void CRGNode::MarkPlaced(Mark mark, bool do)
{
    if (do) SetCurrentMark(mark);
    else SetCurrentMark(null);

    list <CRGNode> node_list = FirstPass(mark, do);

    remove first element from node_list;

    list<CRGNode> updateList = secondPass(node_list, mark, do);
    for each CRGNode N in updateList
    {
        N.updateGUI();
    }
}
```

The algorithm used by `CRGNode::MarkPlaced()` makes three passes over `CRGNode`s. The first and second passes are responsible for updating data structures, and the third is responsible for painting the results on the screen.

The first pass is along the backward dynamic slice of the marked cell. This pass is performed as a recursive depth-first search on `CRGTracers`. This pass has two responsibilities. First, it maintains testedness information as in the WYSIWYT methodology. This is accomplished using the `CRGTracer::Validate-ExecutedDUPairs()` method when a check mark is being placed, and the `CRGTracer::UnvalidateExecutedDUPairs()` method when a check mark is removed. Second, it performs a topological sort of the `CRGNodes` in the backward dynamic slice of the marked cell, and returns this list. Note that the first element in this list contains the node `mark` has been placed on. This node is not needed by the `CRGNode::Second- Pass()` method, so it is removed by the `CRGNode::MarkPlaced()` method before the list is passed to `CRGNode::SecondPass()`.

```
private list<CRGNode> CRGNode::FirstPass(Mark mark, bool do)
{
    list<CRGNode> topo_stack;

    FirstPass_recurse(mark, do, topo_stack);

    return topo_stack;
}

private void CRGNode::FirstPass_recurse(Mark mark, bool do,
                                         list<CRGNode> topo_stack)
{
    CRGTracer tracer = GetTracer();
    if (tracer has not been visited)
    {
        mark tracer as visited

        if (Mark.GetType() is a check mark)
        {
            if (do) tracer.ValidateExecutedDUpairs();
            else tracer.UnvalidateExecutedDUPairs(1);
        }

        for each CRGNode N in tracer.getAffectingCRGNodes()
        {
            N.FirstPass_recurse(mark, do, topo_stack);
        }

        topo_stack.AddToFront(this);
    }
}
```

The second pass is performed on the topologically sorted list of CRGNodes returned by the first pass. This pass is responsible for maintaining the ReachingMarkLists of the CRGNodes, as well as the BlockLists of any marks on these CRGNodes. It returns a list of CRGNodes that have testing and fault localization information that will need to be updated in the user interface. This list is stored in the updateList variable. The first part of this pass deals with the CRGNode that was marked. If a mark is being placed (i.e. do is true), all marks that reach this CRGNode must be blocked. If a mark is being removed, then all of the marks that were blocked by the mark must be unblocked. The variable blocklist is used to store the ReachingMarks for these marks.

```
private list<CRGNode> CRGNode::SecondPass(list<CRGNode> node_list,
                                          Mark mark, bool do)
{

    list<CRGNode> updateList;
    updateList.Add(this);

    BlockList blocklist;

    if (do)
    {
        blockList = GetReachingMarkList().DoBlock(mark);

        mark.SetBlockList(blockList);

        ReachingMark newReachingMark = new ReachingMark(mark, 1);
        GetReachingMarkList().Add(newReachingMark);
    }
    else
    {
        blocklist = mark.GetBlockList().Copy();
        mark.blocklist.Clear();
        GetReachingMarkList().Remove(mark);
        GetReachingMarkList().Add(blocklist);

    }

. . .
```

The DoBlock() method used in this step blocks all the ReachingMarks in a RMList that are blocked by a given Mark. It returns a BlockList which indicates how many marks were blocked, and along how many paths they were blocked.

```
public BlockList RMList::DoBlock(Mark mark)
{
    BlockList result = new BlockList();

    for each ReachingMark rm in theList
    {
        if (mark.blocks(rm.mark))
        {
            result.Add(new ReachingMark(rm.mark, rm.paths));
            template_add(rm, -rm.paths);
        }
    }
    return result;
}
```

Now that the node being marked has been processed, the blocked or unblocked marks must be propagated to the nodes that affect the marked node. This is done by adding or subtracting the changes to the temporary DeltaList of each of these nodes.

```
. . .
    for each CRGNode N in GetTracer().GetAffectingCRGNodes()
    {
        if (do)
        {
           N.GetTempDeltaList().Sub(blockList);
           N.GetTempDeltaList().Add(newReachingMark);
        }
        else
        {
           N.GetTempDeltaList().Add(blocklist);
        }
    }
}
. . .
```

Now the remaining nodes are processed in topological order. These nodes are first added to the `updateList`. For efficiency concerns, we will only perform one update per `FormulaGraph`. As a result, `EltCRGNodes` are not placed into the `updateList`. Instead, their parent `RgnCRGNode` is placed into the list instead.

```
. . .
    for each CRGNode N in node_list
    {
        if (! (N instanceof EltCRGNode) )
        {
            updateList.Add(N);
        }
        else
        {
            RgnCRGNode rgnNode = N.GetParentRgnCRGNode();
            if (rgnNode is marked)
            {
                updateList.Add(rgnNode);
                mark rgnNode;
            }
        }
. . .
```

The next step is to update the `Blocklist` on the `Mark` on the `CRGNode N`. If `mark` is being removed, it needs to be removed from `N`'s blocklist. Next, every `ReachingMark` that has been propagated to the temporary `DeltaList` of `N` is processed. If this `ReachingMark` contains a mark that is newly blocked by the mark on N, the ReachingMark is blocked. Otherwise, it is a mark that was previously blocked by the mark on N, and therefore does not need to be propagated further.

```
. . .
        if (N.GetCurrentMark() != null)
        {
           if (do == false && mark.blocks(N.GetCurrentMark()))
           {
               N.GetCurrentMark().GetBlockList.Remove(Mark)
           }

           for each ReachingMark M in N.GetTempDeltaList()
           {
               if (N.GetCurrentMark().blocks(M.mark))
               {
                   if (M.paths > 0)
                   {
                        N.GetCurrentMark().GetBlockList().Add(
                               N.GetTempDeltaList().BlockRM(M));
                   }
                   else
                   {
                        N.GetCurrentMark().GetBlockList().Add(M);
                        N.GetTempDeltaList().Remove(M);
                   }
               }
           }
       }
. . .
```

The BlockRM() method used above is given a ReachingMark within the RMList and blocks it. It returns a ReachingMark that indicates along how many paths the original ReachingMark was blocked.

```
public ReachingMark RMList::BlockRM(ReachingMark rm)
{
    ReachingMark result = new ReachingMark(rm.mark, rm.paths);
    template_add(rm, -rm.paths);
    return result;
}
```

Next the ReachingMarkList of N needs to be updated. This is done by applying the changes in N's temporary DeltaList to N's ReachingMarkList.

Because the nodes are iterated through in topological order, all of the nodes which N affects are guaranteed to have been visited. This ensures that the temporary DeltaList of N contains all the changes that need to be propagated to N.

```
. . .
          if (do == false)
          {
              N.GetReachingMarkList().Remove(mark);
          }

          N.GetReachingMarkList().Add(N.GetTempDeltaList());
. . .
```

Finally, these changes must be propagated to the nodes which affect N. Once this is done, the temporary DeltaList is no longer needed.

```
. . .
         for each CRGNode CN in N.GetTracer().GetAffectingCRGNodes()
         {
             CN.GetTempDeltaList().Add(N.GetTempDeltaList());
         }

         N.GetTempDeltaList().Clear();
    }
}
```

*An Example of the* MarkPlaced() *Algorithm*

To illustrate the MarkPlaced() algorithm, we will show how it handles placing a check mark on cell a in the spreadsheet shown in Figure 4.3(a). The result of placing this mark is shown in Figure 4.3(b). Notice that this mark blocks the X mark on cell z, but does not block the X mark on cell h. Figure 4.4(a) shows the CRGNodes and fault localization information for spreadsheet prior to the mark being placed. In this figure, the mark on cell z is given the name "X1", and the mark on cell h is given the name "X2." A ReachingMark is shown as a tuple consisting of the mark name and the number of paths associated with that mark. For example, in Figure 4.4(a), the mark X2 reaches cell d through one path, while the mark X1 reaches d through two paths.

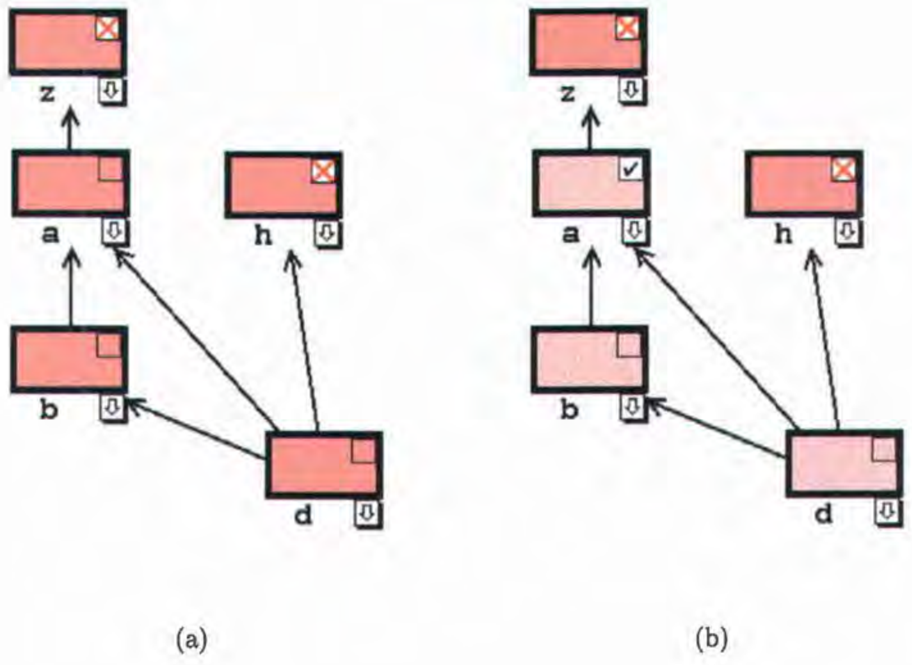


FIGURE 4.3: A sample spreadsheet shown before and after placing a check mark on cell a. Arrows show the data flow dependencies between cells.

In the first pass of the `MarkPlaced()` algorithm, a depth first search is performed starting from cell a that traverses its backward dynamic slice. As this is done, the `DUPairs` coming into each cell are validated, and a topological sort of the `CRGNodes` visited is collected. This sorted list of nodes is then passed to the second pass of the `MarkPlaced()` algorithm.

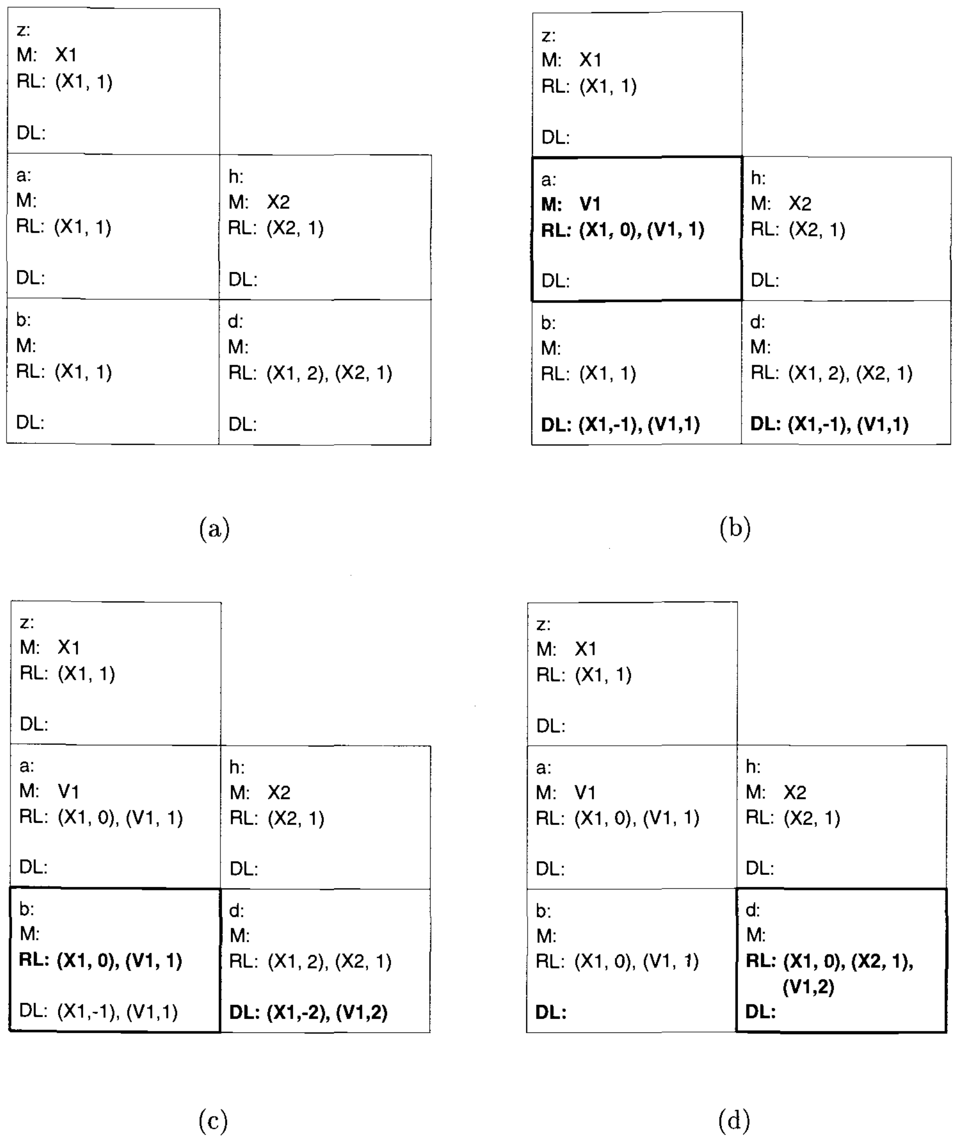

z: M: X1 RL: (X1, 1) DL:
a: M: RL: (X1, 1) DL:
h: M: X2 RL: (X2, 1) DL:
b: M: RL: (X1, 1) DL:
d: M: RL: (X1, 2), (X2, 1) DL:
(a)
z: M: X1 RL: (X1, 1) DL:
a: M: V1 RL: (X1, 0), (V1, 1) DL:
h: M: X2 RL: (X2, 1) DL:
b: M: RL: (X1, 1) DL: (X1,-1), (V1,1)
d: M: RL: (X1, 2), (X2, 1) DL: (X1,-1), (V1,1)
(b)
z: M: X1 RL: (X1, 1) DL:
a: M: V1 RL: (X1, 0), (V1, 1) DL:
h: M: X2 RL: (X2, 1) DL:
b: M: RL: (X1, 0), (V1, 1) DL: (X1,-1), (V1,1)
d: M: RL: (X1, 2), (X2, 1) DL: (X1,-2), (V1,2)
(c)
z: M: X1 RL: (X1, 1) DL:
a: M: V1 RL: (X1, 0), (V1, 1) DL:
h: M: X2 RL: (X2, 1) DL:
b: M: RL: (X1, 0), (V1, 1) DL:
d: M: RL: (X1, 0), (X2, 1), (V1,2) DL:
(d)

FIGURE 4.4: Four steps in an example run of the `MarkPlaced()` algorithm. `CRGNodes` are shown as boxes. The name of a cell is at the top of a `CRGNode`, followed by the `Mark` (if any) on each each node. This is followed by the `ReachingMarkList` and `DeltaList` associated with each node. Changes between each step are shown in bold, and node currently being visited is given a thick border.

The second pass begins by processing the node that was marked. This involves placing a Mark in cell a's CRGNode, changing a's ReachingMarkList to reflect the new mark, and propagating these changes to the DeltaLists of the cells that directly affect the value of cell a. The results of these operations are shown in Figure 4.4(b). The new check mark is named "V1" and is placed on cell a. The tuple (V1, 1) is added to the ReachingMarkList associated with a, indicating that this new mark reaches cell a through one path. Since the mark X1 is blocked by the new check mark, the ReachingMark associated with mark X1 is set to 0 paths. These changes are then converted to the DeltaList "(X1, -1), (V1, 1)." This indicates that the number of paths through which mark X1 reaches cell a has decreased by 1, while the number of paths through which mark V1 reaches cell a has increased by 1. This DeltaList is then added to the temporary DeltaLists associated with the cells that directly affect cell a's value, in this case, cells b and d.

The second pass then iterates through the topologically sorted list of CRGNodes returned by the first pass. Since cell b is affected by cell d, it is next in the topological sort. To process this node, the second pass adds the temporary DeltaList associated with cell b to the ReachingMarkList associated with cell b. This DeltaList is then propagated to all cells that directly affect the value of cell b, in this case cell d. The effects of these changes are shown in Figure 4.4(c). Once the temporary DeltaList of cell b has been propagated, it is no longer needed and is removed.

Cell d is the only cell left to be visited. This is processed in the same way as cell b, except there are no cells to which cell d's temporary DeltaList can be propagated. The final state of the spreadsheet after marking cell a is shown in Figure 4.4(d).

Note that it is critical for the cells to be visited in topological order. If cell `d` had been visited before cell `b`, then the `ReachingMarkList` of cell `d` would contain "(X1, 1), (V1, 1)", which is incorrect.

*Complexity Analysis of the* `MarkPlaced()` *Algorithm*

Let $N$ be the number of nodes in the backward dynamic slice of the marked node, let $E$ be the number of edges in the backward dynamic slice of the marked node, and let $M$ be the total number of marks in the system. The first pass of this algorithm is a depth first search, and so has $O(N+E)$ runtime complexity. The second pass of this algorithm iterates over N nodes, and propagates the temporary `DeltaList` to the cells that affect each node. This also has $O(N+E)$ steps. However, within this loop, the second pass performs addition and subtraction operations between two `RMLists`. The worst case performance for these operations is $O(M^2)$. As a result, the second pass of this algorithm has complexity $O((N+E)M^2)$. The third pass of this algorithm updates all of the affected nodes and has a runtime complexity of $O(N)$. The second pass dominates the complexity of this algorithm, making the runtime complexity of the `CRGNode::MarkPlaced()` algorithm $O((N+E)M^2)$.

The performance impact of this depends on how large the factor of $M^2$ is. If this factor is not large, the performance of this operation will approximate that of evaluating the formula of the marked cell (assuming that the spreadsheet environment has not cached the values of the cell's affecting the marked cell). This is because an algorithm to calculate the cell's value from scratch must first calculate the value of every cell in its backward dynamic slice. Therefore, the evaluation engine must perform a walk similar to that performed by `CRGNode::FirstPass()`. If the factor of $M^2$ is large, it may cause an unacceptable performance loss. However, it should be noted that although the worst case value of the $M^2$ factor depends on the total number of marks in the system, in reality the actual value could be less if only a small set of marked cells affect, and are affected by, the marked cell.

If the factor of $M^2$ is too large, it would be possible to reduce it by implementing the `ReachingMarkList` class with a hash table. This hash table would use the `Mark` and `CRGNode` as a key, and return the number of paths along which that `Mark` reaches that `CRGNode`. To support iteration efficiently, it would also be necessary to place the hash table elements in a series of linked lists, with the `CRGNode` containing a pointer to the first element of this list. This would allow the addition of two `ReachingMarkLists` to take $O(M)$ time, instead of $O(M^2)$ time. However, the penalty is that it would require a more complicated set of data structures that would take up more space. In the worst case situation, in which every mark reaches every cell, this hash table would have a space complexity of $O(CM)$, where C is the total number of cells in the spreadsheet.

### *4.2.2 Changing Test Cases*

When a change in test case occurs, (i.e. a constant cell formula is changed to another constant), all of the affected marks should be removed, although their effects on testing and debugging information are preserved. In our prototype, we accomplish this by way of the `CRGNode::Recomputed()` method. This method is called whenever the evaluation engine recomputes a cell. This happens when a cell has been affected by a change in a test case, *or* when it has been affected by a change in spreadsheet logic. When called, this method removes any marks on the affected `CRGNode`. Therefore this method can be used to remove the affected marks for both changes in test cases, or changes in spreadsheet logic. Furthermore, because this method takes advantage of work the spreadsheet environment must always do when a cell formula is changed, it adds only a constant run time overhead to the act of editing a formula.

### *4.2.3 Changing Spreadsheet Logic*

When a change is made to the spreadsheet's logic (i.e. a non-constant formula is edited, or a constant formula is changed to a non-constant formula), the `CRGNode::NewFormula()`[2] method is called on the `CRGNode` of the edited cell. This method is responsible for removing the effects of all marks that reach the edited cell. It does this by calling `UndoMarkEffects()` on each `CRGNode` that has a mark that reaches the edited `CRGNode`. Note that this is done *before* creating a CRG for the new formula. This is critical, as changing the CRG first would destroy the `DUPairs` that were in effect when the marks being removed were placed. The `UndoMarkEffects()` method returns a list of `CRGNodes` that contain testing and debugging information that needs to be updated in the user interface. After the CRG is changed, this information is updated.

```
public CRGNode::NewFormula(Formula)
{
    list<CRGNode> updateList;

    if (GetReachingMarkList().Size() > 0)
    {
        ReachingMarkList marklist = GetReachingMarkList.Copy();
        For every ReachingMark M in marklist
        {
```

[2] This method differs from that presented in [33, 37]. That algorithm only walked forward along data flow from an edited cell, and did not remove the effects of a removed mark on cells that were not affected by the edited cell.

```
            CRGNode markedNode = M.mark.GetMarkedNode();
            update-list.Add(markedNode.UndoMarkEffects(marklist));
        }
    }

    Create a new CRG for the new formula

    for each CRGNode N in updateList
    {
        N.updateGUI();
    }
}
```

The `UndoMarkEffects()` method of a `CRGNode` performs a recursive, depth first search on the cells that statically affect that `CRGNode`. Note that this cannot be a search through dynamic dependencies, as this method is responsible for removing effects of marks that were placed while different test cases, and therefore different dynamic dependencies, were in effect. For efficiency, this walk is careful not to visit the same `FormulaGraph` twice during a single invocation of `NewFormula()`.

This method removes all marks that reach the edited cell, and it removes all of the effects these marks had on the CRG. However, it is careful not to remove the effects of a mark that was placed on a value that was not affected by the edited cell.

```
private list<CRGNode>
CRGNode:UndoMarkEffects(ReachingMarkList marklist)
{
    list<CRGNode> updateList;
    FormulaGraph fg = GetFormulaGraph();
    int NumRemovedChecks = 0;

    if (fg is not visited)
    {
        mark fg as visited;
        updateList.Add(this);

        if (GetCurrentMark() != null)
        {
            if (marklist.Find(GetCurrentMark()) != null)
            {
                SetCurrentMark(null);
            }
            else
            {
                GetCurrentMark().GetBlockList().Remove(marklist);
            }
        }

        NumRemovedChecks = first element of
                           GetReachingMarkList().remove(marklist);

        for each DUPair p in fg.EnumIncomingDUPairs()
        {
            p.Unvalidate(NumRemovedChecks);
        }


        for each CRGNode N in fg.GetAffectingCRGNodes()
        {
           updateList.Add(N.UndoMarkEffects(marklist));
        }
    }
    updateList;
}
```

*An Example of the* `NewFormula()` *Algorithm*

To illustrate the `NewFormula()` algorithm, we will show how it handles an edit to cell `b` in the same spreadsheet used to illustrate the `MarkPlaced()` algorithm. The before and after states of this edit are shown in Figure 4.5. Note that the marks which were affected by this edit were removed, along with their effects on testing and fault localization information.

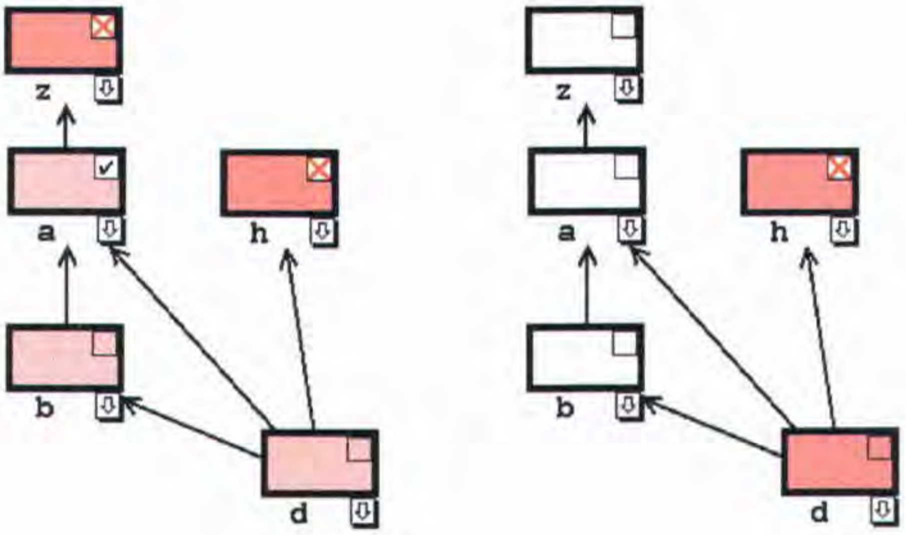


FIGURE 4.5: A sample spreadsheet shown before and after editing the formula in cell b. Arrows show the data flow dependencies between cells.

The `NewFormula()` algorithm first iterates through the `ReachingMarkList` contained in cell b. The first `Mark` in this list, X1, is on cell z, so the `UndoMarkEffects()` method is called on cell z. To process this cell, it removes the `Marks` that reach cell b from z's `ReachingMarkList`, as shown in Figure 4.6(a). The `UndoMarkEffects()` method then continues to visit every cell that statically affects cell z in depth-first order.

In this example, the next cells visited are cells a, d, and b. Again, the `Marks` that reach cell b are removed from the `ReachingMarkLists` of the visited

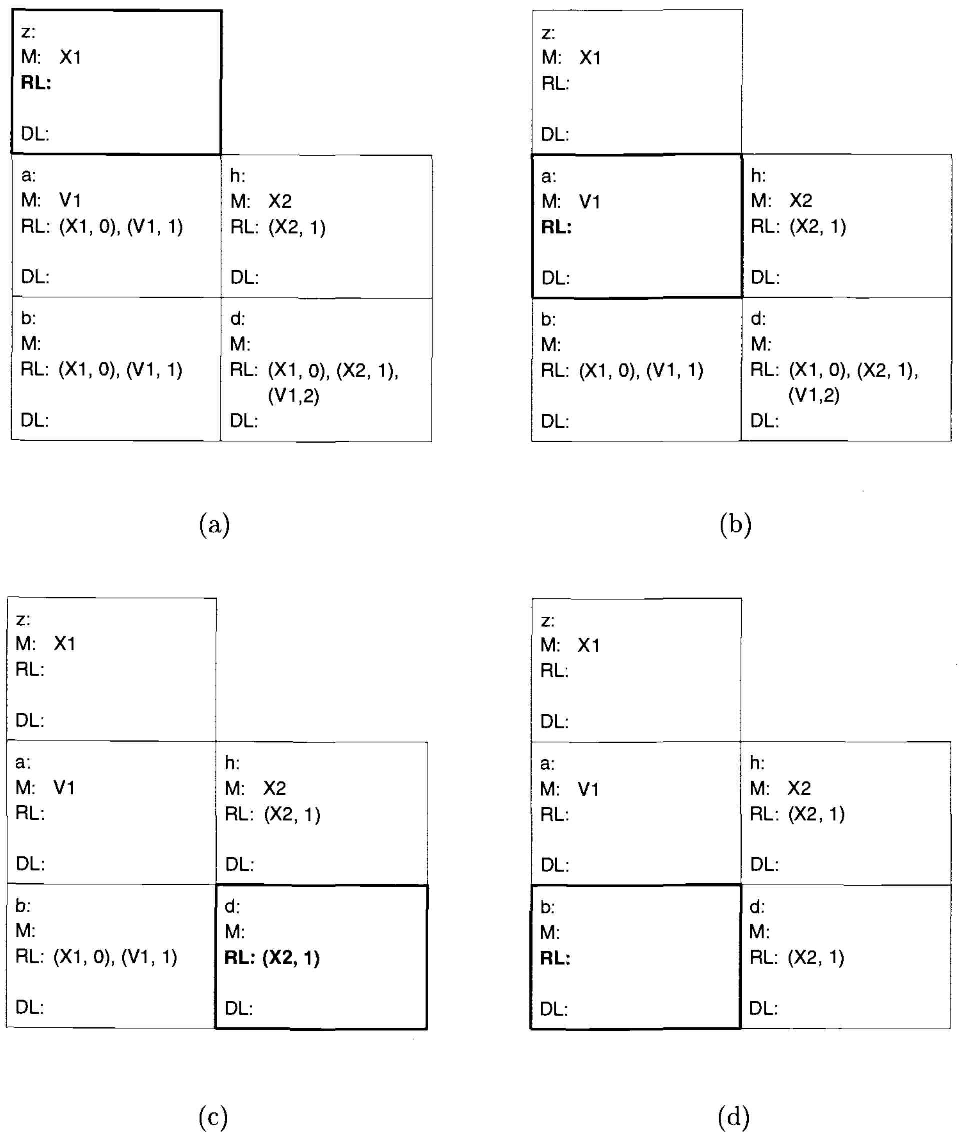


FIGURE 4.6: Four steps in a sample run of the NewFormula() algorithm.

cells. The result of these visits are shown in Figures 4.6(b), 4.6(c), and 4.6(d). Once `UndoMarkEffects()` visits these cells, `NewFormula()` will call it again for the other mark that reaches cell `b`. This will cause `UndoMarkEffects()` to visit cell `a`. However, this cell has already been visited, so `UndoMarkEffects()` returns and the `NewFormula()` algorithm terminates. The final result of the `NewFormula()` algorithm is shown in 4.6(d). Note that the `Marks` themselves have not been removed yet. This will be done later by the `Recomputed()` method when these cells are recomputed by the system.

*Complexity Analysis of the* `NewFormula()` *Algorithm*

The `NewFormula()` algorithm performs a depth first search on every marked cell that is affected by the edited cell. Let $N$ be the total number of `CRGNodes` (excluding `EltCRGNodes` which do not have `FormulaGraphs`) that are reached by a mark that also reaches the edited cell. Let $E$ be the total number of `DUPairs` that are incoming to these nodes. As the `NewFormula()` algorithm is careful not to visit a `FormulaGraph` more than once, only $N$ nodes are visited. For each node, the `UndoMarkEffects()` method performs at least one `RMList::Remove(RMList)` operation, which has a worst case performance of $O(M^2)$, where $M$ is the total number of marks in the system. Additionally, the `UndoMarkEffects()` method also visits all of the incoming `DUPairs` for these nodes during the course of the algorithm. The result is that the combined run-time complexity of the `NewFormula()` algorithm is $O(E + NM^2)$. As with the `MarkPlaced()` algorithm, the $M^2$ factor plays a role in the performance impact of this operation. However, the total number of marks that reach the edited cells also plays a role in determining how large $E$ and $N$ are. From this analysis, it appears that both the `MarkPlaced()` algorithm and `NewFormula()` algorithm perform worst when a large number of marks affect, or are affected by, the cell being edited or marked.

It is important to note that when using the Region Representative approach, the `NewFormula()` algorithm only needs to walk through regions. This is because the testing and fault localization information it removes is only stored on a per `FormulaGraph` basis. This is different from the `MarkPlaced()` algorithm, which follows dynamic dependencies, and therefore needs to consider every cell in a region (as each cell in a region can have different dynamic dependencies due to different execution traces). As a result of this, the region representative approach provides more performance benefits for the `NewFormula()` algorithm than the `MarkPlaced()` algorithm when a large number of cells can share the same formula.

### *4.2.4 Estimating Fault Likelihood*

The above algorithms are responsible for keeping the `ReachingMarkLists` of `CRGNodes` up to date. The four integers stored in these `ReachingMarkLists` are then used to calculate an estimate for fault likelihood. As described in Section 3.3, this fault likelihood is mapped into five distinct shades of pink or red. As a result, the fault likelihood value we calculate is from a range of six distinct values: “none”, “very low”, “low”, “medium”, “high”, and “very high”.

To estimate the fault likelihood for a cell, we first map the values of `numReachingCheckMarks` and `numReachingXMarks` into a range of values from 0 to 5. This mapping is performed using Table 4.2. We then combine these values into an estimate for fault likelihood. All of this is done in the method `ReachingMarkList::CalculateFaultLikelihood()`. This method also ensures

| Range | numReachingXMarks or or numReachingCheckMarks |
|---|---|
| 0 (“none”) | 0 |
| 2 (“low”) | 1–2 |
| 3 (“medium”) | 3–4 |
| 4 (“high”) | 5–9 |
| 5 (“very-high”) | 10+ |

TABLE 4.2: numReachingXMarks and numReachingCheckMarks yield five distinct fault likelihood values as shown. Note that a sixth value for “very-low” is skipped. This value occurs only through the interaction between numReachingXMarks and numReachingCheckMarks.

that if there are any blocked or unblocked X marks affected by a cell, then that cell has a fault likelihood that is at least “very low”, in keeping with Property 1 in section 3.4.

```
enum {NONE = 0, VERY_LOW, LOW, MEDIUM, HIGH, VERY_HIGH};

public int ReachingMarkList::CalculateFaultLikelihood()
{
    if (numReachingXMarks == 0)
    {
        if (numBlockedXMarks == 0) then return NONE;
        else return VERY_LOW;
    }
    else
    {
        int x = map numReachingXMarks to a 0 to 5 range.
        int c = map numReachingCheckMarks to a 0 to 5 range.

        return max(1, x - (c / 2));
    }
}
```

The formula “max(1, x - (c / 2))” was chosen to fulfill Properties 2 and 3 in Section 3.4. It ensures that as the number of X marks affected by the cell increases, the fault likelihood increases, and that as the number of check marks affected by the cell increases, the fault likelihood decreases. The number of X marks was given a higher weight than the number of check marks to prevent an equal number of X and check marks from canceling each other out.

# Chapter 5

# CONCLUSIONS AND FUTURE WORK

This thesis presents an integrated testing and fault localization methodology that is intended to help spreadsheet developers find faults in their spreadsheets. This methodology allows developers to modelessly switch between spreadsheet development, testing, and debugging, and does not make use of any formal testing or debugging terminology that might be unfamiliar to spreadsheet developers.

This methodology always highlights the union of the backward dynamic slices of *all* values marked incorrect, reducing the chances of giving false indications of faults where no faults exists. We feel that such false indications would cause spreadsheet developers, who may not understand the theory behind our methodology, to become disillusioned with and to lose confidence in our methodology.

Unlike program dicing, our methodology does not make a binary decision about whether a cell should be highlighted or not. Instead, it estimates the fault likelihood of cells and uses this estimate to highlight cells in varying shades of red. This allows our methodology to indicate that a small set of cells have a high likelihood of containing faults, while still suggesting that there may be faults within other cells in the union of the backward dynamic slices of all incorrectly marked values.

To find an estimate for fault likelihood, we maintain five properties that describe how we feel fault likelihood should behave. In an analysis of the algorithms used to maintain these properties, we have found that the performance of these algorithms depends largely on the number of marks that affect, or are affected by a given cell.

Future work is planned along two dimensions. First, user studies are required to ascertain whether this methodology helps spreadsheet developers find faults, and whether the algorithms used by this methodology perform well during actual usage. Second, we are considering several possibilities for enhancing this methodology. For example, we may wish to investigate the use of slicing and fault localization at the level of subexpressions. This would allow users to isolate a fault to the actual subexpression where it occurs, but would come at an additional cost. Another possibility is to enhance our method of estimating fault likelihood by replacing Table 4.2 with a more dynamic mapping. This mapping could depend on the number of cells highlighted, or possibly the number of cells in the spreadsheet.

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 without having to scroll within or switch between multiple worksheets.

## BIBLIOGRAPHY

[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] ANSI/IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, New York, 1983.

[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 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.

[6] 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.

[7] M. Burnett, S. Sheretov, and G. Rothermel. Scaling up a 'What You See Is What You Test' methodology to testing spreadsheet grids. In *IEEE Symposium on Visual Languages*, Tokyo, Japan, September 1999.

[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] R. Christ. Review and analysis of color coding research for visual displays. *Human Factors*, 17(6):542–570, 1975.

[11] 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.

[12] 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.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[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] 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.

[20] G. Murch. Physiological priciples for the effective use of color. pages 49–54, November 1984.

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

[22] G. J. Myers. *The Art of Software Testing.* John Wiley, New York, 1979.

[23] 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.

[24] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–369, July 1994.

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

[26] 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.

[27] H. Pan and E. H. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Purdue University, July 1992.

[28] 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.

[29] 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.

[30] J. Reichwein, G. Rothermel, and M. Burnett. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 25–38, October 1999.

[31] 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.

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

[33] 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.

[34] K. Rothermel. Empirical studies of a WYSIWYT testing methodology. Master's thesis, Oregon State University, March 2000.

[35] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. R. G. Green, and G Rothermel. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[36] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, May 1982.

[37] A. Sheretov and M. Burnett. A methodology for testing spreadsheet grids. Technical Report TR00-60-01, Oregon State University, Department of Computer Science, January 1996.

[38] B. Shneiderman. *Designing the User Interface.* Addison-Wesley, Reading, MA, 3rd edition, 1998.

[39] 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.

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

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

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