AN ABSTRACT OF THE THESIS OF

Andrei Sheretov for the degree of Master of Science in Computer Science presented on January 14, 2000. Title: A Methodology for Testing Spreadsheet Grids

Abstract approved: Redacted for Privacy

Margaret M. Burnett

Although there has been research into ways to design spreadsheet systems to improve the processes of creating new spreadsheets and of understanding existing ones, little attention has been given to helping users of these environments test their spreadsheets. To help address this need, we introduce two visual approaches to testing large grids in spreadsheet systems. The first approach is a straightforward extension of a visual testing methodology we previously developed for individual cells (and hence is termed the Straightforward approach), and serves as a useful baseline with which to compare the second approach. The second approach, termed the Region Representative approach, contributes scalability by overcoming the inefficiency of the Straightforward approach.

Prototypes of both approaches have been tightly integrated into Forms/3, a research spreadsheet language, and communication with the user happens solely through the use of checkbox devices and coloring mechanisms. The intent of this work is to bring to end users at least some of the benefits of formalized notions of testing, without requiring knowledge of testing beyond a naive level.
A Methodology for Testing Spreadsheet Grids

by

Andrei Sheretov

A Thesis Submitted
to
Oregon State University

In Partial Fulfillment of the requirements for the degree of

Master of Science

Presented January 14, 2000
Commencement June 2000
Master Science thesis of Andrei Sheretov presented on January 14, 2000

Approved:
Redacted for Privacy

Major Professor, representing Computer Science
Redacted for Privacy
Chair of Department of Computer Science
Redacted for Privacy
Dean of Graduate School

I understand that my thesis will become part of a permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Andrei Sheretov, Author
Acknowledgement

I thank Dr. Margaret Burnett for her time, energy, and endless patience that went into helping me complete my work at Oregon State University. She was simply the best major professor one could hope for.

I thank Dr. Gregg Rothermel for his help and guidance, and also for being my graduate advisor.

I thank all current and past members of Forms/3 Research Group. You are a great team! My special thanks go to Dusty Reichwein and John Atwood for answering or honestly trying to answer my questions, or just listening and nodding at the right times.

Finally, my gratitude goes to my beloved wife. She kept me going.
# Table of Contents

## CHAPTER 1: INTRODUCTION

## CHAPTER 2: BACKGROUND

- 2.1 Spreadsheet languages ................................................................. 3
- 2.2 Testing individual cells .................................................................. 6
- 2.3 An abstract model for spreadsheets ................................................. 7
- 2.3 DU-adequacy for spreadsheets ....................................................... 10
- 2.4 Attributes of grids ...................................................................... 10
  - 2.4.1 Homogeneity ........................................................................ 10
  - 2.4.2 Static versus dynamic ............................................................ 11
  - 2.4.3 Grids in Forms/3 .................................................................. 12

## CHAPTER 3: EXPANDING THE CRG MODEL

- 3.1 Blackbox view of the CRG ................................................................. 13
  - 3.1.1 User interface perspective ......................................................... 13
  - 3.1.2 Internal system perspective ....................................................... 14
- 3.2 CRG dissected ............................................................................. 15
  - 3.2.1 Task 1: Collecting static information ....................................... 15
  - 3.2.2 Task 2: Tracking execution ...................................................... 21
  - 3.2.3 Task 3: Validation ................................................................. 22
  - 3.2.4 Task 4: Adjusting test adequacy information ............................ 24

## CHAPTER 4: TESTING GRIDS

- 4.1 Problems raised by large grids ....................................................... 27
- 4.2 A straightforward approach ............................................................. 27
- 4.3 Region Representative approach ..................................................... 29
  - 4.3.1 What the user does ............................................................... 30
  - 4.3.2 Implications for the CRG model .............................................. 31
  - 4.3.3 Task 1: Collecting static information ..................................... 34
  - 4.3.4 Task 2: Tracking execution .................................................... 39
  - 4.3.5 Task 3: Validation ................................................................. 42
  - 4.3.6 Task 4: Adjusting test adequacy information ......................... 45
# Table of Contents

CHAPTER 5: COMPLEXITY ANALYSIS ................................................................. 47

5.1 Task 1: Collecting static information ......................................................... 47
  5.1.1 Level 4: DUPair::Create() ................................................................. 48
  5.1.2 Level 3: Use::Use() ........................................................................... 48
  5.1.3 Level 2: FGNode::Create() ................................................................. 49
  5.1.4 Level 1: StaticInfo::NewFormula() ..................................................... 50

5.2 Task 2: Tracking execution ........................................................................ 52

5.3 Task 3: Validation ....................................................................................... 52

5.4 Task 4: Adjusting test adequacy information ............................................. 55

CHAPTER 6: PERFORMANCE EXPERIMENT .................................................. 57

CHAPTER 7: CONCLUSION ............................................................................. 62

BIBLIOGRAPHY .............................................................................................. 63
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Programming a clock in Forms/3</td>
</tr>
<tr>
<td>2</td>
<td>The user view of the clock in Forms/3</td>
</tr>
<tr>
<td>3</td>
<td>Forms/3 grades spreadsheet</td>
</tr>
<tr>
<td>4</td>
<td>Cell Relation Graph for Clock</td>
</tr>
<tr>
<td>5</td>
<td>A call graph illustrating the process of formula graph creation</td>
</tr>
<tr>
<td>6</td>
<td>A simple spreadsheet and the corresponding CRG data structures</td>
</tr>
<tr>
<td>7</td>
<td>A version of the Grades spreadsheet using Forms/3 grids</td>
</tr>
<tr>
<td>8</td>
<td>Grades spreadsheet from Figure 7 with the COURSE column blue (fully tested) after two user validations indicated by the checkmarks</td>
</tr>
<tr>
<td>9</td>
<td>Partial CRG of Grades under the Region Representative approach</td>
</tr>
<tr>
<td>10</td>
<td>DualReferencing spreadsheet and a part of its CRG</td>
</tr>
<tr>
<td>11</td>
<td>The Questionnaire spreadsheet</td>
</tr>
<tr>
<td>12</td>
<td>The CRG data structures for spreadsheet shown in Figure 11 built by the Task 1 methods described in this section</td>
</tr>
<tr>
<td>13</td>
<td>Tracing information for one region from spreadsheet in Figure 11</td>
</tr>
<tr>
<td>14</td>
<td>Constructor call hierarchy for Task 1</td>
</tr>
<tr>
<td>15</td>
<td>10-student version of the experimental MatrixGrades spreadsheet</td>
</tr>
</tbody>
</table>
List of Tables

Table                      Page

1: The grammar for Forms/3 formulas. ................................................................. 5
2: The relationships between different CRG node types and their components. ...... 33
3: Statement groups in Use::Create........................................................................ 49
4: Statement groups in FNode::BuildIncomingDUPairs........................................... 50
5: Statement groups in CRGNode::Validate............................................................ 53
6: Statement groups in Use::RecursiveValidate..................................................... 54
7: Timings and system independent measures for Tasks 1 and 4. ......................... 58
8: Summary of user actions to validate the experimental spreadsheet. .................. 59
9: Timings and system-independent measures for Task 3....................................... 60
A Methodology for Testing Spreadsheet Grids

Chapter 1: Introduction

Testing is an important activity, used widely by professional and end-user programmers alike in locating errors in their programs. In recognition of its importance and widespread use, there has been extensive research into effective testing in traditional programming languages in the imperative paradigm. However, there are few reports in the literature on testing in other paradigms, and no reports that we have been able to locate on testing in spreadsheet systems.

Spreadsheet languages, which are also known as form-based languages in some of the research literature, provide a declarative approach to programming, characterized by a dependence-driven, direct-manipulation working model [2]. Users of spreadsheet languages create cells, and define formulas for those cells. These formulas reference values contained in other cells and use them in calculations. When a cell’s formula is defined, the underlying evaluation engine calculates the cell’s value and those of other affected cells (at least those that are visible to the user), and displays new results.

The spreadsheet paradigm includes not only commercial spreadsheet systems, but also a number of research languages that extend the paradigm with explicitly visual features, such as support for gestural formula specification [3, 9], graphical types [3, 20], visual matrix manipulation [18], high-quality visualizations of complex data [4], and specifying GUIs [10]. In this document, we use the term spreadsheet languages to describe all such systems following the spreadsheet paradigm.

Unfortunately, despite the perceived simplicity of spreadsheet languages, and even though spreadsheet creators devote considerable effort to finding and correcting their errors [11], errors often remain. In fact, a recent survey of spreadsheet studies [12] reports spreadsheet error rates ranging from 38% to 77% in controlled experiments, and from 10.7% to 90% in “production” spreadsheets—those actually in use for day-to-day decision making. A possible factor in this problem is the unwarranted confidence creators of spreadsheets seem to have in the reliability of their spreadsheets [19].

To help solve this problem, in previous work [14], we presented a testing methodology for spreadsheets. To accommodate the evaluation models used with
spreadsheets and the interactive process by which they are created, our methodology is validation-driven and incremental. This is accomplished through the use of a test adequacy criterion that focuses on dependencies that influence validated output cells, and through the use of incremental program analysis. To accommodate the user base of these languages, we provide an interface to the methodology that does not require an understanding of testing theory. This is accomplished through a fine-grained integration with the spreadsheet environment to provide testing information visually.

However, scalability issues were not addressed in that previous work. In this document*, we describe two ways to scale up the approach to support large grids of cells with shared or copied formulas.

* Much of the material contained in this thesis has previously appeared in [3] and [14]
Chapter 2: Background

2.1 Spreadsheet languages

Users of spreadsheet languages set up spreadsheets and specify their contents in order to program. The contents of a spreadsheet are a collection of cells; each cell’s value is defined by that cell’s formula, and as soon as the user enters a formula, it is evaluated and the result is displayed. The best-known examples of spreadsheet languages are found in commercial spreadsheet systems, but there are also many research systems (e.g. [3, 4, 9, 10, 16, 17]) based on this paradigm.

In this document, we present examples of spreadsheets in the research language Forms/3 [3]. Figure 1 and Figure 2 show how a user could construct a graphical clock in Forms/3. Figure 1 shows each cell with its formula. Clock consists of 13 cells, including two input cells (upper left) that could eventually be replaced with references to the system clock, one output cell (middle left), and several cells used in intermediate calculations (right). After the programming is finished, the cells that calculate intermediate results can be hidden, and other cells rearranged, to reach the user view shown in Figure 2.

In this document, we consider a "pure" spreadsheet language model, which includes ordinary spreadsheet-like formulas such as those described by the grammar in Table 1, but excludes advanced programmer oriented formulas such as macros, imperative sublanguages, and recursion. Table 1 reflects a subset of Forms/3. The subset shown uses ordinary spreadsheet formulas for both numeric and graphical computations; the figures presented in this document were programmed using this subset.
Figure 1: Programming a clock in Forms/3

Figure 2: The user view of the clock in Forms/3. On the input cells, formula tabs have been left visible to encourage inputting new hour and minute values. The formula tab has been hidden on the output cell.
Table 1: The grammar for Forms/3 formulas. (Note that subexpressions are fully parenthesized, thereby avoiding ambiguity.) As the top section shows, it has the usual spreadsheet formula operators and also some operators supporting computations on grids (dynamic matrices) and on graphics. The bottom section shows cell reference syntax, which includes row/column referencing for cells that are in a grid (Matrix).
2.2 Testing individual cells

The underlying assumption in our work has been that, as the user develops a spreadsheet incrementally, he or she is also testing incrementally. We have integrated a prototype implementation of our approach to incremental, visual testing into the spreadsheet VPL Forms/3 [3]. In our prototype, every cell in the spreadsheet is considered to be untested when it is first created, except “input cells” (cells whose formulas may contain constants and operators, but no cell references and no if-expressions), for which the testedness is not defined. For the non-input cells, testedness is reflected via border colors on a continuum from untested (red) to tested (blue).

Figure 3: Forms/3 grades spreadsheet. The user validated four of the cells, and then, to test further, entered a new input for Farnes's HwAvg. The Course formulas (not shown) have an if-expression; since only one branch of it has been tested, the borders for the two validated Course cells are between red and blue (gray and black, in this document).

The process is as follows. During the user's spreadsheet development, whenever the user notices a correct value, he or she lets the system know of this test (decision) by
validating the correct cell (clicking in the checkbox in its upper right corner), which causes a checkmark to appear, as in Figure 3. This communication allows the system to track successful tests, to propagate the implications of the successful test to cells that contributed to it, and to reflect this increase in “testedness” by coloring borders of the checked cell and its contributing cells more “tested” (more blue). On the other hand, whenever the user notices an incorrect value, rather than checking it off, he or she eventually finds the faulty formula and fixes it. This formula edit means that affected cells will now have to be re-tested; the system is aware of which ones those are, and re-colors their borders more “untested” (red).

But, what is “testedness” and what does it mean to be fully tested? Most spreadsheets can have an infinite number of inputs; hence, not all possible inputs can be tested. Test adequacy criteria are criteria used to decide whether a program has been tested “enough.” In our previous work, we developed an abstract model for simple spreadsheets and used it to define several test adequacy criteria [14]. The strongest criterion we defined, du-adequacy, is the criterion we use in this document to define when a spreadsheet has been tested “enough”. We describe and extend the model and du-adequacy as they relate to spreadsheet grids in the following sections. The border colors described above are a mapping from \( n \), a percent tested according to the du-adequacy criterion, to the color at \( n + K_n \)% past the start of a red-blue continuum, where each \( K_n \) adjusts to ensure that 100% tested is considerably more blue than 99% tested, and that 0% tested is considerably more red than 1% tested.

Thus, if the user manages to turn all the red borders blue, the test adequacy criterion has been satisfied. In our empirical work on simple spreadsheet cells, several measurements of users’ testing effectiveness and efficiency were significantly higher for subjects using Forms/3 supplemented by this scheme than for those using Forms/3 without the testing supplement [15].

2.3 An abstract model for spreadsheets

Test adequacy criteria provide help in selecting test data and in deciding whether a program has been tested "enough". Test adequacy criteria are often defined on abstract models of programs rather than on code itself. We have created such an
abstract model for spreadsheet languages [14]; we call our model a cell relation graph (CRG). A CRG is a pair \((V, E)\), where \(V\) is a set of formula graphs, and \(E\) is a set of directed edges connecting pairs of elements in \(V\). Figure 4 depicts the CRG for Clock.

Each formula graph in \(V\) models flow of control within a cell’s formula, and is comparable to a control flow graph representing a procedure in an imperative program [1, 13]. There is one formula graph for each cell in the spreadsheet. The process of translating an abstract syntax tree representation of an expression into its control flow graph representation is well known [1]; a similar translation applied to the abstract syntax tree for each formula in a spreadsheet yields that formula’s formula graph. For example, Figure 4 shows the formula graphs for the cells in Clock, delimited by dotted rectangles. In these graphs, nodes labeled "E" and "X" are entry and exit nodes, respectively, and represent initiation and termination of evaluation of formulas. Nodes with multiple out-edges (represented as rectangles) are predicate nodes. Other nodes are computation nodes. Edges within formula graphs represent flow of control between expressions, and edge labels indicate the value to which conditional expressions must evaluate for particular branches to be taken.

The set \(E\) of edges in the CRG contains cell dependence edges, which model data dependencies between cells. Figure 4 depicts these edges by dashed lines. Each edge encodes the fact that the destination cell refers to the source cell in its formula; thus, the arrows show direction of dataflow. Note that cell dependence information is typically available to evaluation engines within spreadsheet systems as a consequence of the need to evaluate formulas; thus, this information need not be specially calculated in order to construct CRGs.

Finally, we require a way to associate execution of formulas with CRG components. Let \(F\) be a formula with formula graph \(F'\), and let \(F_e\) and \(F_x\) be the entry and exit nodes, respectively, of \(F'\). An evaluation of \(F\) traverses a path through \(F'\), beginning at \(F_e\) and ending at \(F_x\). We call this path the execution trace for that evaluation.
Figure 4: Cell Relation Graph for Clock
2.3 DU-adequacy for spreadsheets

We use this abstract model to define the definition-use test adequacy criterion. In spreadsheets, cells serve as variables, and the value for cell C can be defined only by expressions in C's formula. Let C be a cell in spreadsheet S, with formula F and formula graph G. If C is an input cell, then G contains only one node other than entry and exit nodes, and that node is a definition of C. If C is not an input cell, then each computation node in G that represents an expression referring to cell D is a c-use (computation use) of D and a definition of C. Each edge in G that has as its source a predicate node n such that n represents a conditional expression referring to another cell D is a p-use (predicate use) of D.

Let S be a spreadsheet with CRG R. A definition-use association (du-association) links definitions in R with uses that those definitions may reach. Two types are of interest. A definition-c-use association (du-association) is a triple (n1,n2,C), where C is a cell, n1 is a definition of C, n2 is a c-use of C, and there exists a path in R from n1 to n2. A definition-p-use association is a triple (n1,(n2,n3),C), where C is a cell, n1 is a definition of C, (n2,n3) is a p-use of C, and there exists a path in R from n1 to (n2,n3).

The du-adequacy criterion is defined as follows. Let S be a spreadsheet, and let T be a test suite for S. T is du-adequate for spreadsheet S if and only if, for each executable du-association x in S, there exists at least one test case t in T that exercises x.

2.4 Attributes of grids

2.4.1 Homogeneity

A grid is a two-dimensional matrix of cells. Most commercial spreadsheet systems are entirely grid-based. The grids of particular interest to us are largely homogeneous—i.e., many of their cells have identical formulas except perhaps for row/column indices. Thus, in this document, the term grid implies some homogeneity,
and a region means a subgrid in which every cell has the same formula, except perhaps for row/column indices.

A spreadsheet language needs knowledge of the homogeneity of a grid region's formulas as a necessary first step in taking advantage of the approach described in this document, but this knowledge is easily obtained. It is already present in those spreadsheet languages in which the user is allowed to explicitly share a single formula among several cells (e.g. Lotus™, Forms/3 [3], Formulate [18], Prograph spreadsheets [16], and Chi et al.'s visualization spreadsheet language [4]). If not already present, it can easily be gathered "behind the scenes" by a spreadsheet system, such as by maintaining knowledge of the relationships among copied formulas as in [5].

2.4.2 Static versus dynamic

There are two attributes of grids and regions that are static in some spreadsheet languages and dynamic in others, and these attributes significantly impact the manner in which "testedness" of grid cells can be tracked. The first is whether a grid's size (number of rows and columns) is specified statically or dynamically. Static specification of grid size is the norm for commercial spreadsheet systems, but some research systems use dynamic size specifications (e.g., Forms/3 and Formulate).

The second of these two attributes is whether determination is static or dynamic as to exactly which cells are being referenced in a formula. The most common approach in commercial spreadsheet systems is static, restricting cell row/column references to be based only on static position, optionally offset by a constant.

Traditional imperative languages—for which most research in testing has occurred—typically support statically-sized, dynamically-referenced grids via arrays. Approaches for reasoning about the testedness of array elements have been suggested [6, 7, 8]; in general, however, the problem of precisely treating array references at the element level is unsolvable for the dynamic referencing that is the norm in imperative programs. Thus, the prevalence of static referencing in the spreadsheet paradigm affords unusual opportunities for reasoning about testedness.

In summary, for viable application to commercial spreadsheet systems, a testing methodology must at least support statically-sized, statically-referenced grids. The two
approaches described in this document do support this type of grid, and also support the dynamically-sized, statically-referenced grid type.

2.4.3 Grids in Forms/3

Our work was prototyped using a grid called a matrix in Forms/3. To define values for a Forms/3 grid's (matrix's) cells, the user statically partitions the grid into rectangular regions and, for each region, enters a single formula for all cells in it. To statically derive a cell's formula from its shared region formula, any "pseudo-constants" i and j in the formula are replaced by the cell's actual row and column number. Each grid has two additional cells, its row dimension cell and column dimension cell, to specify its number of rows and columns. These cells can have arbitrarily complex formulas.
Chapter 3: Expanding the CRG Model

In section 2.2.3 we described the cell relation graph (CRG) – an abstract model on which our testing methodology is based. Now we shall present a framework of classes that supports the CRG. On this framework we shall illustrate the approaches for testing grids using and extending the CRG model in Chapter 4.

We shall use a pseudocode, consisting of a mixture of Java and C++ to describe our framework. A commonly used facility will be iteration over a collection of objects. Since the syntax of this operation can vary greatly depending on the specific data structure containing the collection, we adopt the following notation. Type list<TYPE> indicates a collection containing objects of type TYPE.

3.1 Blackbox view of the CRG

The CRG can be viewed as a collection of CRG nodes each encapsulating data and behavior of one cell. We depict CRG nodes with dashed rectangles, as in Figure 4. The interaction between CRG nodes and the rest of the system can be viewed from two perspectives.

3.1.1 User interface perspective

The CRG node supplies information about cell testedness to the UI (User Interface), which can represent testedness as a colored border (or perhaps in some other form). Validation tab status information is also maintained in the CRG node and supplied to the UI when necessary (later we will see why this is important). The UI signals to the CRG node when the user clicks on the validation tab. The following two interfaces summarize the interaction between CRG nodes and the UI.

```java
interface CRGNodeToUI {
    void SetTestingBorderStatus(CellRef aCellRef, int whatToSetTo);
    void SetValidationTabStatus(CellRef aCellRef, int whatToSetTo);
};
```
interface UIToCRGNode {
    void ValidationTabClicked();
};

A CellRef is a reference (in a formula) to a cell and uniquely identifies a cell. It can be used to access any data structures associated with the cell.

Interface CRGNodeToUI is how the UI receives information from the CRG, and should be implemented by the user interface components of the system. Interface UIToCRGNode should be implemented by the CRG nodes.

3.1.2 Internal system perspective

Interaction between the CRG and the rest of the underlying system involves maintaining two kinds of information: static and dynamic. Everything that is shown in Figure 4 is static information. It includes formula graphs, and du-associations connecting formula graph nodes. In addition, to determine which du-associations were involved in the computation when the user validates a cell, the system keeps track of which formula graph nodes were executed in the most recent evaluation. With this in mind, we can define two interfaces comprising methods that are called by the system when events of interest to a CRG node occur.

interface StaticInfo {
    void LoadCell(Formula aFormula);
    void NewFormula(Formula aFormula);
};

interface DynamicInfo {
    void StartTrace();
    void TracePredicate(bool value);
    void StopTrace();
};

The argument to both of the StaticInfo interface methods is a parsed representation of a new formula for a cell. For example, in Forms3, formula "if (A=B) then (A*B) else (A-B)" is parsed into (if (= A B) (* A B) (- A B)). Although a textual formula could be provided, the underlying system has already done the parsing. Not using the results of that parsing in construction of the formula graph would be unnecessary duplication of effort.

DynamicInfo::StartTrace() is called when the cell is about to be evaluated. DynamicInfo::TracePredicate() is called when the evaluation engine has evaluated one of the predicate expressions and made a decision about which execution branch will
be followed. DynamicInfo::StopTrace() is called immediately after the cell's evaluation.

Now we can write a skeleton for CRGNode class

```java
class CRGNode
    implements CRGNodeToUI
    implements StaticInfo
    implements DynamicInfo
{
    // nothing here yet
};
```

### 3.2 CRG dissected

Now that we have established a protocol by which CRG nodes will communicate with the outside world, we turn our attention to the internal structure of the CRGNode class. Four main tasks that the CRGNode and its components need to perform are:

**Task 1:** Whenever the corresponding cell receives a new formula, build static formula graphs and collect du-associations.

**Task 2:** When the cell is being evaluated, keep track of which formula graph nodes participated in the computation of the cell's value.

**Task 3:** Whenever the user validates the cell by clicking on the validation tab, mark relevant du-associations covered, as well as those of its producers, and show the increased testedness via colors.

**Task 4:** Whenever a cell receives a new formula, mark the du-associations in its consumers that are affected by the old formula “not covered”.

### 3.2.1 Task 1: Collecting static information

#### 3.2.1.1 The formula graph

Recall that each CRG node contains a graph, called a formula graph, representing a single cell's formula. A formula graph consists of predicate and computation nodes, which share some common data and functionality: All formula graph
nodes may have uses and, therefore, incoming du-associations. However, only computation nodes may serve as definitions and have outgoing du-associations. In addition, predicate nodes need to maintain links to both of their branches.

```csharp
class FGNode {
    public static FGNode Create(Formula aFormula);
    public list<DUPair> EnumIncomingDUPairs();
    public Testedness GetTestedness();

    protected list<Use> uses;
};
class ComputationNode extends FGNode {
    public ComputationNode(Formula aFormula);
    public list<DUPair> EnumOutGoingDUPairs();

    protected list<DUPair> outgoingDUPairs;
};
class PredicateNode extends FGNode {
    public PredicateNode(Formula predicateExpression,
                         FGNode trueBranch,
                         FGNode falseBranch);

    protected FGNode trueBranch;
    protected FGNode falseBranch;
};
```

Method `FGNode::Create` is used as a factory with constructors of `ComputationNode` and `PredicateNode` directly initializing member-variables from parameters.

```csharp
FGNode FGNode::Create(Formula aFormula){
    if (aFormula.IsAnIf())
        return new PredicateNode(  
            aFormula.GetPredicateExpression(),  
            FGNode::Create(aFormula.GetTrueBranchExpression()),  
            FGNode::Create(aFormula.GetFalseBranchExpression()));
    else
        return new ComputationNode(aFormula);
}
```

The set of `FGNode` constructors provides the context in which the routines that create du-associations are called. Consider the following formula:

```
if (A=B) then (if (C=A) then (A*B) else (B+C)) else (A-B)
```

Figure 5 illustrates the method calls used to create the corresponding formula graph.
**Figure 5**: A call graph illustrating the process of formula graph creation. Thin arrows show where the returned values are passed.

### 3.2.1.2 DUPairs: the edges connecting formula graphs

Abstract class **DUPair** represents a du-association that can return its testedness and provide access to both definition and use nodes. Definition-c-use associations and definition-p-use associations are represented by **DCUPair** and **DPUPair** classes respectively. Both of them are subclasses of **DUPair** and implement the `GetTestedness()` method.

```csharp
class DUPair {
    public static DUPair Create(ComputationNode definition, Use use);
    public DUPair(ComputationNode definition, Use use);
    public Testedness GetTestedness() = 0;
    public DefinitionNode GetDefNode();
    public FGNode GetUseNode();
};

class DCUPair extends DUPair {
    public DCUPair(ComputationNode definition, Use use);
    public Testedness GetTestedness();

    protected bool covered;
};
```
class DPUPair extends DUPair {
    public DPUPair(ComputationNode definition, Use use);
    public Testedness GetTestedness();

    protected bool trueBranchCovered;
    protected bool falseBranchCovered;
};

Class Use holds information about a specific use, including the cell reference that was used in a formula and the number of times it occurred. For example, formula "A + A - B" contains two uses: "A" with multiplicity 2, and "B" with multiplicity 1. This is a shortcut for storing duplicate identical uses for "A".

class Use {
    public Use(FGNode useNode, CellRef aCellRef, int multiplicity);
    public Testedness GetTestedness();

    FGNode fgNode;
    CellRef cellRef;
    int multiplicity;
    list<DUPair> duPairs;
};

Figure 6 illustrates the formula graph and du-association objects for a simple spreadsheet.
Figure 6: A simple spreadsheet and the corresponding CRG data structures. Dotted rectangles show the CRGNodes. Arrows indicate single pointers (such as trueBranch) or collections of pointers (such as outgoingPairs).
Use and du-association objects are created during formula-graph node initialization. This functionality is shared by predicate and computation nodes, and can therefore be implemented in their superclass FGNode.

```cpp
void FGNode::BuildIncomingDUPairs(Formula aFormula) {
    for each cellRef in aFormula do {
        multiplicity = the number of times cellRef occurs in aFormula;
        uses.Add(new Use(this, cellRef, multiplicity));
    }
}
```

A naive implementation using two linked lists and two nested loops would take $O(n^2)$, where $n$ is the number of cell references in the formula. In the Forms/3 implementation, we use a temporary hash-table large enough to achieve $O(n)$ runtime complexity for this method.

```cpp
ComputationNode::ComputationNode(Formula aFormula) {
    BuildIncomingDUPairs(aFormula);
}

PredicateNode::PredicateNode(Formula predicateExpression, FGNode trueBranch, FGNode falseBranch) {
    BuildIncomingDUPairs(predicateExpression);
    this.trueBranch = trueBranch;
    this.falseBranch = falseBranch;
}
```

At this point we have established the context in which individual Use objects are created. The constructor of Use needs to find all definitions of the cell being referenced and create du-associations connecting those definitions to the use.

```cpp
Use::Use(FGNode useNode, CellRef aCellRef, int multiplicity) {
    fgNode = useNode;
    cellRef = aCellRef;
    this.multiplicity = multiplicity;
    for each computationNode in cellRef.FindCRGNode().EnumDefinitions() do
        DUPair::Create(computationNode, this);
}
```

Once the Use objects exist, it is possible to create DUPair objects. Similarly to the FGNode class hierarchy, static method Create in the base abstract class serves as a factory and makes decisions about the types of objects to be created.

```cpp
DUPair DUPair::Create(ComputationNode definition, Use use) {
    if (use.getUseNode() instanceof ComputationNode)
        return new DCUPair(definition, use);
    else
        return new DFUPair(definition, use);
}
```
3.2.2 Task 2: Tracking execution

To track execution traces, which enable up-to-date knowledge of which du-
associations have been exercised, we have inserted a probe into the evaluation engine. This probe calls methods of interface DynamicInfo for the corresponding CRGNode object. Each cell's CRGNode stores the trace on its formula graph. If the cell is subsequently reevaluated, the old execution trace is replaced with the new one. Storing only the most recent execution trace is sufficient for coverage computation because the cumulative coverage is updated incrementally during validation, as we shall describe in our discussion of Task 3.

There are two main types of queries regarding the execution trace:
1. Given a cell, what formula graph nodes participated in the last execution?
2. Given a formula graph node, did it participate in the last execution?

Methods supporting these queries and the implementation of DynamicInfo interface are combined into class CRGTracer. Each CRGTracer object has a pointer to the formula graph node that it traces. This handles the most general case: concurrent evaluation of cells is possible, and hence each cell needs a separate CRGTracer object. Therefore, each CRGNode implements methods of the DynamicInfo interface by forwarding calls to its CRGTracer.

```csharp
class CRGTracer implements DynamicInfo {
    public bool IsNodeExercised(FGNode aNode);
    public list<FGNode> EnumTrace();

    protected FGNode currentFGNode;
};
```

In previous work [14] we have established that recording a trace only adds a constant factor of runtime complexity to what the system is already doing during evaluation. Our implementation of CRGTracer keeps an "exercised" flag in each FGNode object as its way of storing the trace. An alternative could be maintaining a list of FGNode objects that comprise the execution trace.
3.2.3 Task 3: Validation

Whenever the user pronounces a displayed value valid by clicking on a validation tab, the system uses the static du-association information and execution traces, previously calculated and stored as discussed in the descriptions of Tasks 1 and 2, to identify the du-associations that participate in the production of the currently validated value, and to update borders of participating cells.

Validation is special purpose graph walk. The algorithm starts with a set of nodes in the trace of a validated cell, and recursively follows exercised du-associations in the direction opposite to data flow. All du-associations encountered during this walk are validated and corresponding border colors updated.

To implement an efficient graph walk algorithm we needed a mechanism to prevent multiple visits to the same nodes of the graph. One approach to this task would have been to include a "visited" flag into each node. Every time the walk algorithm encountered a new node, the "visited" flag of that node would be set. If later this node were reached by a different path, the flag would indicate that there is no need to revisit the node. A drawback to this approach would be that after each graph walk, all visited nodes would need to have their "visited" flags reset.

We use a slightly modified version of the above technique.

class GSearchable {
    // Should be called before a graph walk/search
    public static void NewSearch(){
        globalSearchCounter++;
    }

    // Marks an object "visited"
    public void MarkVisited(){
        searchCounter = globalSearchCounter;
    }

    // Determines if the object has been visited
    // during the current walk/search
    public bool Visited(){
        return searchCounter == globalSearchCounter;
    }

    private static long globalSearchCounter = 0;
    private long searchCounter = 0;
};

Any class of objects that we desire to mark "visited" during a graph walk is subclassed under GSearchable. There is no need to reset the "visited" flag after each graph walk, since every walk has a unique ID. The number of unique IDs is limited by
the size of type long. (After the pool of available IDs is exhausted, the behavior of the
system is undefined. In the research prototype of Forms/3 we decided not to handle this
case, because the number of available IDs is more than sufficient for our applications).

The validation process is implemented by the following set of methods:

```cpp
void CRGNode::Validate(){
    GSearchable::NewSearch();
    RecursiveValidate();
}
```

CRGNode::Validate() is called when the user clicks on a validation tab. This
method initializes the walk and invokes RecursiveValidate() method, which is
implemented by all objects in formula graph structure.

```cpp
void CRGNode::RecursiveValidate(){
    if(!Visited()){
        MarkVisited();
        for each node in tracer.EnumTrace() do node.RecursiveValidate();
        UpdateGUI();
    }
}
```

CRGNode::RecursiveValidate() accomplishes three goals: make sure the
node is only visited once during the graph walk; pass validation request to the formula
graph nodes that are in the trace; update the cell’s border to reflect new testedness
information. In order for this method to work, CRGNode is made a subclass of
GSearchable.

```cpp
void FGNode::RecursiveValidate(){
    for each use in uses do use.RecursiveValidate();
}
void Use::RecursiveValidate(){
    for each duPair in duPairs do duPair.RecursiveValidate();
}
```

FGNode::RecursiveValidate() and Use::RecursiveValidate() simply pass
the validation requests to the DUPair objects.

```cpp
void DCUPair::RecursiveValidate(){
    CRGTracer defTracer = getDefNode().getTracer();
    if(defTracer.IsNodeExercised(getDefNode())){
        covered = true;
        getDefNode().getCRGNode().RecursiveValidate();
    }
}
```
void DPUPair::RecursiveValidate()
{
    CRGTracer defTracer = getDefNode().getTracer();
    CRGTracer useTracer = getUseNode().getTracer();

    if(defTracer.IsNodeExercised(getDefNode())){
        if(useTracer.IsNodeExercised(getUseNode().getTrueBranch()))
            trueBranchCovered = true;
        else
            falseBranchCovered = true;
    }
    getDefNode().getCRGNode().RecursiveValidate();
}

For both types of du-associations receiving a validation request indicates that the
use node should be assumed exercised. For a DCUPair it is sufficient to verify that the
definition node is exercised before the pair is marked covered. For a DPUPair, the
system also needs to determine which part of the association participated in the
computation.

The main body of RecursiveValidate() is called no more than once per cell,
and the entire validation process terminates in worst-case time proportional to the
number of du-associations validated. Because the set of uses in a cell's trace
corresponds to a set of definitions in that cell's direct producers, which in turn lead to that
cell's indirect producers, the cost of validation is bounded by the number of direct and
transitive producers of a cell. This is less that or equal to the cost of calculating the cell's
value the first time (when no reusable values are present in the cache), but the algorithm
is triggered by a user interaction that does not require evaluation, so, unlike the other
algorithms we have presented, its cost cannot be masked by the cost of evaluation
process.

3.2.4 Task 4: Adjusting test adequacy information

So far, we have focused on how our methodology handles cell formulas as they
are added to a spreadsheet. We now consider the other basic edits possible with
spreadsheets, namely, deleting a cell or changing a cell's formula. Changes to a
constant-formula cell are equivalent to the application of a new test input (which may or
may not be followed by validations by the user), and requires no action beyond that
involved in recalculating execution traces as discussed under Task 2. Deletion of a cell
is equivalent to modifying that cell's formula to BLANK. Thus, we need only consider modifications to non-constant formulas.

A formula modification may have far-reaching consequences, and the system must immediately reflect the new test adequacy status of the spreadsheet whenever a cell is modified. To accomplish this, the system must (1) update static information about the formula graph and execution trace, and (2) update the coverage flags on affected du-associations.

We have discussed how item (1) is handled in Task 1 and Task 2. For item (2) we use a conservative approach that recursively visits potentially affected cells. The set of methods UnValidate() and RecursiveUnValidate() is similar to Validate() and RecursiveValidate(), but instead of using dynamic information to walk backwards through producers, it uses static information to walk forward through consumers.

```cpp
void CRGNode::UnValidate()
{
    GSearchable::NewSearch();
    list<CRGNode> visitedCRGNodes = RecursiveUnValidate();
    for each node in visitedCRGNodes do node.UpdateGUI();
}
```

Unlike the validation, unvalidation may visit the same CRGNode more than once because different parts of the formula graph may be reached through different paths in the CRG. It is desirable that the user interface update is done only once per cell. Therefore, the updates are performed only after the unvalidation walk is complete.

```cpp
list<CRGNode> CRGNode::RecursiveUnValidate()
{
    if(!Visited()){
        MarkVisited();
        return new list(this);
    }
    else
        return null;
}
```

```cpp
list<CRGNode> PredicateNode::RecursiveUnValidate()
{
    if(!Visited()){
        MarkVisited();
        list<CRGNode> visitedCRGNodes = new list<CRGNode>;
        visitedCRGNodes.add(getCRGNode().RecursiveUnValidate());
        visitedCRGNodes.append(getTrueBranch().RecursiveUnValidate());
        visitedCRGNodes.append(getFalseBranch().RecursiveUnValidate());
        return visitedCRGNodes;
    }
}
```
list<CRGNode> ComputationNode::RecursiveUnValidate(){
    if(!Visited()){
        MarkVisited();
        list<CRGNode> visitedCRGNodes = new list<CRGNode>;
        for each duPair in outgoingDUPairs do
            visitedCRGNodes.append(duPair.RecursiveUnValidate());
        return visitedCRGNodes;
    }
}
list<CRGNode> DCUPair::RecursiveUnValidate(){
    covered = false;
    return getUseNode().RecursiveUnValidate();
}
list<CRGNode> DPUPair::RecursiveUnValidate(){
    trueBranchCovered = false;
    falseBranchCovered = false;
    return getUseNode().RecursiveUnValidate();
}

Because UnValidate's processing is consumer-driven, then as with Task 1, the cell visits required by it are already required for display and value cache maintenance under most evaluation engines.
Chapter 4: Testing Grids

4.1 Problems raised by large grids

The methodology for testing spreadsheets described in Chapter 3 applies at the granularity of individual cells. However, most large grids in spreadsheets are fairly homogeneous, i.e., consist of many cells whose formulas are identical except for some of the row/column indices. For example, suppose the spreadsheet in Figure 3 were expanded to calculate student grades for a class containing 300 students. There were two problems with the previous testing system for this kind of grid:

Problem 1: For the user, the problem was that each of the 300 course grade cells would have to be explicitly validated for the spreadsheet to appear completely tested (blue). The user is unlikely to go to this much trouble for essentially-identical cells, which would mean the user would be burdened with keeping track of which cells “really” need testing and which ones do not because of their similarities to other cells.

Problem 2: For the system, the problem was that the performance of the testing subsystem depended on the number of cells. Hence, responsiveness was impaired by the presence of large grids.

For both the user and the system, these burdens seem inappropriate, given that the Grades spreadsheet’s logic with 300 students is exactly the same as in the same Grades spreadsheet with only 5 students. In order to solve these problems, the previous methodology needed to be extended to explicitly support homogeneous grids.

4.2 A straightforward approach

One approach to explicitly supporting grid testing is to let the user validate all or part of an entire region in one operation, but to have the system continue to maintain testedness information about each cell individually. We term this approach the “Straightforward” approach. For the Straightforward approach, the CRG model does not
need to be modified, although extensions to the way the CRG model is used are required. The Straightforward approach is a straightforward extension of our previous methodology, and is described here primarily because it is a useful baseline for comparison for our other, more effective, approach.

Because all information is kept individually for each cell, the user has the flexibility to validate any arbitrary group of cells, or even any cell individually. For example, the user has chosen to rubberband most of the Course column of Figure 7 and validate that group in one click, since all of those cells use the “else” part of the formula, but to attend individually to the bottom cell, which uses the “then” part. This approach does not address Problem 2, but it provides a highly flexible solution to Problem 1.

Figure 7: A version of the Grades spreadsheet using Forms/3 grids.

As we pointed out in Section 2.3.2, the static referencing in spreadsheets creates excellent opportunities for reasoning about testedness. Most importantly, resolving a reference with relative indices to the actual cell is an $O(1)$ operation. We term this operation StaticallyResolve. For example, if M[1,3] refers to P[i,j-1], to determine the actual cell that is being referenced, the system calls StaticallyResolve(P[i,j-1], 1, 3), which returns P[1,2]. Given static referencing, StaticallyResolve works even in the case of dynamically sized grids, because in that combination each region size except
one must be static. It is reasonable to rely on the underlying system to provide the
S\texttt{taticallyResolve} operation as a method of class CellRef. Based on that, we
provide a \texttt{S\texttt{taticallyResolve}} method for a CRGNode class.

\begin{verbatim}
CRGNode CRGNode::S\texttt{taticallyResolve}(CellRef aCellRef){
  if(aCellRef.IsAMatrixReference() && this.GetCell(),IsAMatrixCell()){
    CellRef resolvedRef = aCellRef.S\texttt{taticallyResolve}(
      this.getI(), this.getJ());
    return resolvedRef.getCRGNode();
  } else
    return aCellRef.getCRGNode();
}
\end{verbatim}

Using this method, we rewrite \texttt{Use::Use()} constructor for the Straightforward
approach.

\begin{verbatim}
Use::Use(FGNode useNode, CellRef aCellRef, int multiplicity){
  this.fgNode = useNode;
  this.cellRef = aCellRef;
  this.multiplicity = multiplicity;
  CRGNode defCRGNode = getCRGNode().S\texttt{taticallyResolve}(aCellRef);
  for each computationNode in defCRGNode.EnumDefinitions() do
    duPairs.Add(DUPair::Create(computationNode, this));
}
\end{verbatim}

This constructor uses \texttt{CRGNode::S\texttt{taticallyResolve}()} to locate
CRGNode objects for all types of references that may be encountered during the construction of
static du-associations. The decision whether or not to invoke
\texttt{CellRef::S\texttt{taticallyResolve}()} is made within \texttt{CRGNode::S\texttt{tationallyResolve}()}.

The worst-case time costs of the Straightforward approach for tasks 1, 3, and 4
approach on a region of n cells, not surprisingly, is at least n * the cost of testing an
individual cell. Task 2 remains a simple O(1) probe in the evaluation engine. The
dependency on region size can be a significant detriment to responsiveness for large
grids. However, this approach does provide the expressive power to allow the user to
easily and flexibly validate all or part of an entire region in a single operation.

\subsection*{4.3 Region Representative approach}

The "Region Representative" approach aims directly at Problem 2 (system
efficiency) by working at least partially at the granularity of entire regions rather than at
the granularity of individual cells in those regions. This is accomplished by sharing most
of the CRG data structures described in Chapter 3 among all cells in a region. This improves system efficiency over the Straightforward approach and provides some conveniences to the user that are greater than in the Straightforward approach, but it does not provide quite as much flexibility.

4.3.1 What the user does

The visual devices are the same as in the Straightforward approach, but the implications of the user’s actions are different: the user’s validation of one grid cell X now propagates—to every cell in its region—the du-associations covered by executing X. For example, if no cells in Figure 7 were validated yet and then the user validated the top Course cell, which executes the predicate and the else-expressions in the formula, all of the Course column’s cells would be shown in purple (partially tested). If the user subsequently validated the bottom Course cell, which executes the then-expression, the entire column’s borders would become blue (fully tested) as shown in Figure 8.

![Fig 8](image-url)
The Region Representative approach offers several problem-solving advantages from the user’s perspective. These advantages stem from the fact that the user does less test generation manually: a large grid already provides a variety of input data. The first advantage, obviously, is that the user may not need to conjure up new test inputs. For example, in the Grades spreadsheet, the user tested the top Course cell in part by selecting another cell for validation—the bottom Course cell—because it had a useful set of test inputs already contributing to it. In contrast to this, in the Straightforward approach the user could achieve coverage on the top Course cell only by forcing execution of both branches in that particular cell. This leads to a mechanical advantage as well: the Region Representative approach requires fewer physical actions, i.e. edits and validation clicks, to achieve full coverage. The third advantage is that, when the user does not provide a new test input, he or she does not need to modify the “real” input data and then remember to restore it. Fourth, the user’s job as “oracle” (decider of the correctness of values) may be a little easier with the Region Representative approach, because with so many inputs from which to choose, it may be possible to notice one that produces obvious answers, such as a row with identical inputs in the first 3 columns in the Grades example.

An apparent disadvantage is loss of flexibility: the user seems to have no way to prevent the propagation of testedness to all the cells in the region. Hence, some functionality is lost. For example, the user cannot exclude a cell from group tests in favor of individualized testing, such as a cell that refers to an out-of-range value.

However, most instances of this disadvantage can be removed by allowing the user to subdivide a region into more than one region for testing purposes. For example, suppose there is a region R in which each cell is computed by adding one to the cell above it. The user might want to test the top row of a rectangular region separately because it is based on an initial set of values (those provided by a different region above it) rather than upon cells in the same region. To do this, the user simply subdivides R into two regions, R1 and R2, and tests them separately.

4.3.2 Implications for the CRG model

The Region Representative approach requires the changes to the CRG model. Instead of a formula graph for each cell in a region R, R’s cells are modeled by a single
formula graph of a *region representative* cell $R_{ij}$ of that region. Du-associations are further divided into two classes: those with a constant definition formula and those with a non-constant definition formula. This allows du-associations between multiple constant definitions and a single region's use to be treated as one. We shall now discuss how these changes affect the CRG framework developed in Chapter 3.

### 4.3.2.1 Sharing formula graphs

Recall that all cells in a matrix region share the same formula. Pseudo-constants $i$ and $j$ may be used in that formula, but are replaced by the actual row and column numbers during the evaluation. Maintaining only one formula graph for all cells within one matrix region (also called "region elements"), obviously helps with Problem 1 and 2. However, key to the Region Representative approach is also what is *not* shared among the region elements: cell trace information. Evaluation of different region elements may take different paths through the formula graph. Therefore, it is vital that the region elements maintain individual traces. This enables the system to accurately determine the exercised du-associations during validation, while saving resources by sharing the formula graph.

To support formula graph sharing, the functionality of CRGNode is divided into FormulaGraph and CRGTracer components. FormulaGraph encapsulates data and functionality for maintaining a collection of formula graph nodes and du-associations. CRGTracer class has been described in section 3.2.2. In the Region Representative approach, the formula graph component is maintained at the region level, while the CRGTracer component is maintained individually for each cell; see Table 2.

<table>
<thead>
<tr>
<th></th>
<th>FormulaGraph</th>
<th>CRGTracer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CellCRGNode</td>
<td>Has-a</td>
<td>Has-a</td>
</tr>
<tr>
<td>RgnCRGNode</td>
<td>Has-a</td>
<td></td>
</tr>
<tr>
<td>EltCRGNode</td>
<td></td>
<td>Has-a</td>
</tr>
</tbody>
</table>
Table 2: The relationships between different CRG node types and their components. Regular cells have CellCRGNodes, regions have RgnCRGNodes, and region elements (cells inside regions) have EltCRGNodes.

4.3.2.2 Constant du-associations

To further address Problem 2 in the Region Representative approach, we introduce constant definition-use associations. The definitions in these du-associations are located in constant cells. One difference between regular du-associations and constant du-associations lies in the fact that to determine whether a constant du-association is dynamically exercised the system need only verify that the use node is exercised, which implies that the sole definition node has been exercised. Another important distinction related to matrices is that multiple constant du-associations terminating in the same use within a region formula can, and should, be validated simultaneously. In fact, they can even be considered to be the same du-association participating in several different test cases. Consider Figure 9, which shows a portion of Region Representative CRG for the Grades spreadsheet in Figure 7.

All cells in columns HWAVG, MIDTERM, and FINAL have constant formulas, and each is in a separate region. There are five constant du-associations between cells in column FINAL and the use Grades[i@3] in node 9. If the system required each of these du-associations to be validated separately, the user would need to enter different inputs into rows 1 through 4 to force the execution of the true branch before all of these du-pairs could be validated. However, if the constant du-associations are validated together, the user could pick a row, in which one of them was exercised (row 5 in our example) and validate the entire set in one click.

To realize this advantage, in addition to a list of regular incoming du-associations, the Use class now maintains a separate list of incoming constant du-associations. Simultaneous validation of all constant du-associations in a use is simplified by keeping the coverage information within the Use class.
Figure 9: Partial CRG of Grades under the Region Representative approach without constant du-associations (left) and with constant du-associations (right)

4.3.3 Task 1: Collecting static information

Recall that whenever a cell receives a new formula, Task 1 is triggered by a call to one of the methods of interface StaticInfo given in Section 3.1.2. In the Region Representative approach only two of the three types of CRG node objects - RgnCRGNode and CellCRGNode - forward calls to methods of the StaticInfo interface. Since EltCRGNode does not have a formula graph, in its implementation the methods of StaticInfo are replaced by stubs.

Since the formulas of region elements are shared, the du-associations are recorded only between definitions and uses located in regular cell formula graphs and region formula graphs. This leads to changes in the way that the StaticallyResolve() works. Consider Figure 10.
In this case, the system will need to statically resolve the reference to M1[i@j+1] in node 11 for the entire region in matrix M2. The result will be a list of two regions in M1 that M2 may potentially reference through this use. Thus, \texttt{StaticallyResolve()} returns a list of CRG nodes instead of just one as in the Straightforward approach. For example, suppose the routine shown below is called for the region in M2 with M1[i@j+1] passed as an argument. Here \texttt{aCellRef} is M[1@j+1] and \texttt{aCellRef.StaticallyResolve(1,1)} will resolve \texttt{topLeft} into M1[1@2], and \texttt{aCellRef.StaticallyResolve(1,2)} will resolve \texttt{bottomRight} into M1[1@3]. This says the referenced area rectangle will be \{(1,2) (1,3)\}. In the second step, all three regions of M1 will be tested for overlap with the referenced area rectangle, resulting in the final return value being a list containing M1’s two rightmost regions.
list<CRGNode> RgnCRGNode::StaticallyResolve(CellRef aCellRef) {
    list<CRGNode> results = new list<CRGNode>;
    if(aCellRef.IsAMatrixReference()) {
        // 1. Using aCellRef.StaticallyResolve() determine the
        // referenced area.

        // In our example will resolve to M1[1@2]
        aCellRef.topLeft = aCellRef.StaticallyResolve(1,1);

        // In our example will resolve to M1[1@3]
        aCellRef.bottomRight = aCellRef.StaticallyResolve(
            this.getLastRowNumber(),
            this.getLastColumnNumber());

        Rectangle referencedArea = new Rectangle(topLeft.GetI(),
            topLeft.GetJ(),
            bottomRight.GetI(),
            bottomRight.GetJ());

        // 2. Determine which regions overlap with the referenced
        // area, and add them to the list of results.

        for each region in topLeft.ParentMatrix() do
            if(region.OverlapsWith(referencedArea))
                results.Add(region.GetCRGNode());

        return results;
    } else {
        results.Add(aCellRef.getCRGNode());
        return results;
    }
}

RgnCRGNode::StaticallyResolve() relies on rectangular regions in its work. If
non-rectangular regions were introduced, they too could be viewed as a collection of
rectangles. StaticallyResolve(), in such cases, would employ multiple calls to
StaticallyResolve() for rectangular subregions. The number of such calls would be
directly proportional to the number of rectangles used to represent the regions being
resolved.

We now provide the new (replacing the one in Section 3.2.1) implementation of
Use::Use(), which shows how StaticallyResolve() is applied.

Use::Use(FGNode useNode, CellRef aCellRef, int multiplicity) {
    this.fgNode = useNode;
    this.cellRef = aCellRef;
    this.multiplicity = multiplicity;

    // see discussion of constant du-associations below
    this.constantDUPairCore = DUPairCore::Create(useNode);

    for each defCRGNode in useNode.getCRGNode().StaticallyResolve(aCellRef) do
        for each computationNode in defCRGNode.EnumDefinitions() do
            DUPair::Create(computationNode, this);
}
We now turn our attention to mechanisms that support constant du-associations described in section 4.3.2. First, the du-associations are now orthogonally divided into constant / non-constant and definition-c-use / definition-p-use sets. Since the two sides of this division are orthogonal, they are impossible to model with just one inheritance hierarchy of DUPair classes. Second, the coverage data for all constant du-associations terminating in one use needs to be shared, so it can be efficiently updated and retrieved. Based on these considerations, we modify the previous DUPair hierarchy by creating an additional class hierarchy DUPairCore.

class DUPairCore {
    static DUPairCore Create(FGNode useNode) {
        if (useNode instanceof ComputationNode)
            return new DCUPairCore(useNode);
        else
            return new DPUPairCore(useNode);
    }
    void Validate(CRGTracer tracer) = 0;
    void UnValidate() = 0;
    Testedness GetTestedness() = 0;
};
class DCUPairCore extends DUPairCore {
    public DCUPairCore(ComputationNode useNode);
        protected bool covered
};
class DPUPairCore extends DUPairCore {
    public DPUPairCore(PredicateNode useNode);
        protected PredicateNode useNode;
        protected bool trueBranchCovered;
        protected bool falseBranchCovered;
};

Objects of these classes are contained by both DUPair and Use objects.

However, DUPair is now subclassed by ConstantDUPair and NonConstantDUPair instead of DCUPair and DPUPair.

class ConstantDUPair extends DUPair {
};
class NonConstantDUPair extends DUPair {
    public NonConstantDUPair(ComputationNode definition, Use use) {
        super(definition, use);
        core = DUPairCore::Create(use.getFGNode());
    }
    private DUPairCore core;
};
class Use {
    public Use(FGNode useNode, CellRef aCellRef, int multiplicity);
    public Testedness GetTestedness();

    private FGNode fgNode;
    private CellRef cellRef;
    private int multiplicity;
    private list<DUPair> constantDUPairs;
    private list<DUPair> nonConstantDUPairs;
    private DUPairCore constantDUPairCore;
};

DUPair::Create(), a factory method, now determines if the object to be created represents a constant du-association and invokes the constructor of the appropriate subclass (ConstantDUPair or NonConstantDUPair). Based on the type of DUPair, it is placed onto the appropriate list in its Use.

Figure 11 and Figure 12 illustrate how the components described in this section work together to maintain the static information when the Region Representative approach is applied.

Figure 11: The Questionnaire spreadsheet. The first column contains the input data of a questionnaire. In the second column the number of positive answers is calculated. Cell M[i@2] contains the number of positive answers in cells M[1@1] through M[i@1].
4.3.4 Task 2: Tracking execution

Recall that within this task we provide the implementation for interface DynamicInfo, whose methods are triggered whenever a cell is evaluated. These methods enable the system to store information about which parts of the formula graphs were exercised during evaluation.
The task of tracking execution in the Region Representative approach is accomplished essentially the same as before, with the only differences being in the way the trace information is stored. In the Region Representative approach both regular cell and matrix region element CRG node objects contain trace information. In case of regular cells, there is only one CRGTracer object associated with a formula graph, just as before, which does not require any changes to the original framework. But in the Region Representative approach the CRGTracers of region elements may record different traces through the same shared formula graph of the parent region. Mechanisms for supporting such traces are described here.

We define two subclasses of CRGTracer - CellCRGTracer and EltCRGTracer - to support these two different trace recording schemes. CRGTracer thus becomes an abstract class that delegates the implementation of three methods highlighted in the class definition below to its subclasses.

class CRGTracer implements DynamicInfo {
    public list<FGNode> EnumTrace();

    public bool IsNodeExercised(FGNode aNode) = 0;
    protected void MarkNodeExercised(FGNode aNode) = 0;
    protected void UnmarkNodeExercised(FGNode aNode) = 0;

    protected FGNode currentFGNode;
};

These three methods manipulate and query the state of the "exercised" flag associated with the tracer object. For CellCRGTracer there is only one such flag per an FGNode. It is trivially stored as a boolean member variable with each FGNode object. All three methods in this case have an O(1) run-time complexity.
Figure 13: Tracing information for one region from spreadsheet in Figure 11.

There are several ways in which EltCRGTracer methods can be implemented, among which the use of a hash-table is the most run-time efficient. A combination of region element indices serves as a key with the "exercised" flag being the value (see Figure 13). Again, the run-time complexity of the three virtual method implementations is O(1), but a hash-table must be maintained for each FGNode object in the formula graphs of matrix regions.
4.3.5 Task 3: Validation

Validation is triggered when the user clicks in a validation tab. This task in Region Representative approach remains a graph-walk through dynamically exercised du-associations in the direction opposite to data flow. However, several important modifications to the techniques described in 3.2.3 need to be made in order to accommodate the changes in the CRG model.

The same formula graph objects may receive validation messages more than once, since region elements can reference each other even within the same region (see Figure 11 and Figure 12). However, objects that contain tracers will only be visited once per validation walk, because spreadsheets do not have circular referencing. Hence, the walk is considered to be performed “through tracer objects”, which implies that CRGTracer inherits to become a GSearchable object.

Since there is no direct mapping from a formula graph to any one tracer, a reference to the current tracer object must be passed along through the recursive calls to validation routines of formula graph objects.

Consider Figure 12, which shows the CRG data structures for the spreadsheet in Figure 11. When cell M[6@2] is validated, its tracer is retrieved, which gives the formula graph nodes that participated in the most recent evaluation of M[6@2], namely PredicateNode 6 and ComputationNode 7. In the figure, the first is use M[i@1], pointed at by PredicateNode: 6. Because this use contains only constant DUPairs, the use's DUPairCore is immediately validated. The next use in the example is M[i-1@2], pointed at by ComputationNode: 7. This use contains both constant and non-constant DUPairs. The DCUPairCore, representing all constant DUPairs, is immediately validated, and then the system proceeds to validate the NonConstantDUPairs 7 and 10. Before any of these pairs may be validated the system needs to find out which of them are exercised. In order to determine this, a definition-side tracer needs to be found for each DUPair so that the definition formula graph nodes may be checked as to whether they are exercised. For both NonConstantDUPairs 7 and 10, the use cell reference is M[i-1@2]. The system is aware that the validated use occurred in cell M[6@2]. Using StaticallyResolve translates M[i-1@2] into M[6@2]'s perspective and yields cell M[5@2] as the definition cell for both NonConstantDUPairs 7 and 10. The tracer of M[5@2] is retrieved and ComputationNode 8 is found to be in the trace, while
ComputationNode 7 is not. The definition node for NonConstantDUPair 7 is ComputationNode 7, and for NonConstantDUPair 10 is ComputationNode 8. Thus, only NonConstantDUPair 10 will be validated (by validating the DCUPairCore it is pointing at). The system will now recursively proceed to validate cell M[5@2].

Since the effects of validating a single region element propagate to every element of the region, GUI updates are necessary for all affected element borders and validation tabs. When combined with the possibility of multiple validations occurring on the same formula graph during one validation walk, this creates a danger of unnecessary duplication of GUI update calls. To address this problem, the system simply returns a set of visited CRGTracer objects from the validation walk. (That the set’s elements are unique is guaranteed by the marking scheme described in Section 3.2.3.) A second one-step graph walk is done starting from this set and reaching all affected formula graphs, and this walk is where the calls to UpdateGUI() occur. Through this mechanism, the system guarantees that a GUI update method will be called no more than once per cell, per validation walk.

CRGNode::Validate()
{
    // Validation walk "through tracers"
    GSearchable::NewSearch(); // initializing the global search counter

    list<CRGTracer> affectedTracers = this.getTracer().RecursiveValidate();

    // GUI update walk "through formula graphs"
    GSearchable::NewSearch(); // initializing the global search counter

    for each tracer in affectedTracers do {
        affectedFG = tracer.getFormulaGraph()
        if(!affectedFG.Visited()){
            affectedFG.MarkVisited();
            affectedFG.getCRGNode().UpdateGUI();
        }
    }
}

// Override to conceal Validate() method in RgnCRGNode class,
// because regions may not be validated directly.
RgnCRGNode::Validate()
{
    // Empty
}

During the second walk EltCRGNode::Validate() calls only CellCRGNode::UpdateGUI() and RgnCRGNode::UpdateGUI().

RgnCRGNode::UpdateGUI() passes the UpdateGUI message to all elements in the region.

Due to the changed nature of the graph walk, the RecursiveValidate() methods had to be completely rewritten to support the Region Representative approach.
list<CRGTracer> CRGTracer::RecursiveValidate(){
    if(!Visited(){
        MarkVisited();
        list<CRGTracer> results = new list<CRGTracer>; // Validate all constant du-associations as one
        for each node in this.EnumTrace() do
            results.Append(node.RecursiveValidate(this));
    }
    results.Add(this);
}

list<CRGTracer> FGNode::RecursiveValidate(CRGTracer tracer){
    list<CRGTracer> results = new list<CRGTracer>
    for each use in uses do results.Append(use.RecursiveValidate(tracer));
    return results;
}

list<CRGTracer> Use::RecursiveValidate(CRGTracer tracer){
    list<CRGTracer> results = new list<CRGTracer>
    // Validate all constant du-associations as one
    constantDUPairCore.Validate(tracer);
    for each duPair in nonConstantDUPairs do{
        CRGTracer definitionTracer = duPair.IsExercised(tracer);
        if(definitionTracer != null){
            duPair.Validate();
            results.Append(definitionTracer.RecursiveValidate());
        }
    }
    return results;
}

    The DUPair::IsExercised() method does a little bit of extra work for efficiency's sake in Use::RecursiveValidate(). As the name implies, it determines whether the given du-association is exercised. In order to do that
DUPair::IsExercised() locates the definition-side tracer for the given du-association. This may or may not involve a call to StaticallyResolve() depending on whether the definition is a matrix cell. But in Use::RecursiveValidate() the same definition-side tracer has to be located if the pair is indeed exercised and the RecursiveValidate() message needs to be passed to that tracer. This would have involved a duplicate call to StaticallyResolve(). In order to avoid this duplication of effort, instead of just returning a boolean, DUPair::IsExercised() returns a reference to the definition-side tracer if the du-association is exercised and null otherwise.
CRGTracer DUPair::IsExercised(CRGTracer tracer){
    // First, find the right tracer for the definition side of
    // this du-association
    CRGTracer defTracer;
    CellRef aCellRef = this.GetCellRef();
    if(aCellRef.IsAMatrixReference())
        // If this du-association involves a matrix cell reference, the
        // definition-side formula graph may have more than one corresponding
        // tracer. StaticallyResolve() must be used to find the right
        // definition-side tracer.
        defTracer = tracer.GetCRGNode().StaticallyResolve(aCellRef).GetTracer();
    else
        // If this du-association involves an ordinary cell reference,
        // the definition-side formula graph has just one corresponding
        // tracer, which can be trivially found.
        defTracer = this.GetDefNode().GetTracer();
    // Second, after the definition-side tracer has been found, determine
    // if this du-association is indeed exercised.
    if(defTracer.IsNodeExercised(this.GetDefNode()))
        // There is no need to check the use-size tracer, since if this
        // method has been called, the use formula graph node is exercised.
        return defTracer;
    else
        return null;
}

list<CRGTracer> NonConstantDUPair::Validate(){
    GetDUPairCore().Validate();
}

In summary, the methods constituting the implementation of Task 3 in the Region
Representative approach significantly differ from the ones in the original framework.
Instead of CRG nodes, the graph walk is performed through the tracer objects. The
additional implementation complexity is due to the need for calls to
StaticallyResolve() when du-associations involving matrix cell definitions are
involved. This implementation complexity is part of the price that has to be paid for the
advantages of the Region Representative approach.

4.3.6 Task 4: Adjusting test adequacy information

Very few modifications to the way the system handles Task 4 are required to
support the Region Representative approach. Adjusting test adequacy information still
remains a graph walk through the CRG in the direction of data flow, and uses only static
information.

Introduction of constant du-associations causes slight rearrangements in the set
of RecursiveUnValidate() methods described in Section 3.2.4. Notably, methods
DCUPair::RecursiveUnValidate() and DPUPair::RecursiveUnvalidate() are replaced by the following set of methods:

```cpp
list<CRGNode> DUPair::RecursiveUnvalidate(){
  this.GetDUPairCore().UnValidate();
  return getUseNode().RecursiveUnValidate;
}

void DCUPairCore::UnValidate(){
  covered = false;
}

void DPUPairCore::UnValidate(){
  trueBranchCovered = false;
  falseBranchCovered = false;
}
Chapter 5: Complexity Analysis

This chapter analyzes the worst-time complexities of the Region Representative approach algorithms. The essence of this chapter is that the time costs of running these algorithms except for the validation task are approximately the same as the other ordinary spreadsheet tasks that are taking place due to the same triggers. This means that all except the validate task add little or no more than O(1) to the time cost of operations in a spreadsheet language that does not have a testing subsystem. (The validate task's time cost is similar to the time cost of calculating the value of the cells in the region being validated.)

5.1 Task 1: Collecting static information

This task is triggered when the user changes a region's formula. At this point, the tasks any spreadsheet system must perform even without the existence of a testing subsystem are (1) to parse the formula, which costs at least the number of characters in the formula, (2) to calculate at least the on-screen cells of the region receiving a new formula, and (3) to notify any consumers of the edited region that their cached values must be discarded.

Given the new formula the system constructs a formula graph and du-associations. This process may be viewed as a set of cascading constructor calls (Figure 14). Each constructor in this hierarchy creates a number of lower level objects while also performing tasks specific to its level. For example, FGNode::Create() calls a Use::Create() for each use that it encounters. But first, the information about uses and their multiplicities is extracted from the corresponding formula part. And obviously, FGNode::Create() is also responsible for the construction of formula graph structure. (This process is described in detail in Sections 3.2.1 and 4.3.3.)
5.1.1 Level 4: DUPair::Create()

At this point both definition and use nodes of the new du-association have been determined. Only one lower level object - DUPairCore - is created if the constructed object is of type NonConstantDUPair, which takes $O(1)$ time. The decision about which pair type (ConstantDUPair or NonConstantDUPair) to create is made based on whether the definition cell has a constant formula, which can be determined in $O(1)$ time from its formula graph. Thus DUPair::Create() has the worst case complexity $O(1)$.

$$\text{Level4Cost} = O(1)$$ (1)

5.1.2 Level 3: Use::Use()

We divide the Use::Use() constructor (taken from Section 4.3.3) into four groups of statements shown in Table 3.

In Group 1 member variables are initialized, which clearly takes $O(1)$.

Group 2 is a loop through the list CRG nodes that aCellRef may refer to. This list is produced by StaticallyResolve(), whose worst-time complexity is $O$(number of regions in grid referenced by aCellRef). Group 3 is a loop through the list of definition formula graph nodes in the formula graph of defCRGNode. The complexity of this group is $O$(number of definitions in defCRGNode).
Use::Use(FGNode useNode, CellRef aCellRef, int multiplicity)(
  this.fgNode = useNode;
  this.cellRef = aCellRef;
  this.multiplicity = multiplicity;
  this.constantDUPairCore = DUPairCore::Create(useNode);
)

for each defCRGNode
  in useNode.getCRGNode().StaticallyResolve(aCellRef) do
  for each computationNode in defCRGNode.EnumDefinitions()
    DUPair::Create(computationNode, this);

Table 3: Statement groups in Use::Create

Time cost of Group 4 is Level4Cost. And thus, the total time cost of
Use::Create() is:

\[
\text{Level3Cost} = \mathcal{O}(N_{\text{Regions}} \times N_{\text{DefsReferenced}} \times \text{Level4Cost}) = \mathcal{O}(N_{\text{Regions}} \times N_{\text{DefsReferenced}})
\]

where: \( N_{\text{Regions}} \) is the number of grid regions (statically) referenced by the created
use,
\( N_{\text{DefsReferenced}} \) is the maximum number of definition nodes in any
referenced region.

5.1.3 Level 2: FGNode::Create()

FGNode::Create() employs constructors PredicateNode::PredicateNode() and
ComputationNode::ComputationNode() (see Section 3.2.1) to create the formula
graph structure. All of these methods add \( \mathcal{O}(1) \) overhead to the cost of
FGNode::BuildIncomingDUPairs(), which is used to construct the incoming du-
associations for all types of formula graph nodes. Table 4 shows this method divided
into groups of statements.
```c
void FGNode::BuildIncomingDUPairs(Formula aFormula)
{
    for each cellRef in aFormula do {
        multiplicity = the number of times cellRef
        occurs in aFormula;
        uses.Add(new Use(this, cellRef, multiplicity));
    }
}
```

Table 4: Statement groups in FGNode::BuildIncomingDUPairs

Some details of the algorithm for extracting the use information from a formula (Group 1) have been purposefully omitted in this description. As we discussed in Section 3.2.1, a naive implementation using two linked lists and two nested loops would take $O(n^2)$, where $n$ is the number of cell references in the formula. In the Forms/3 implementation, we use a temporary hash-table large enough to achieve $O(n)$ runtime complexity.

Given that the addition to a list in Group 2 is an $O(1)$ operation, the time cost of
FGNode::BuildIncomingDUPairs() is

$\text{Level2Cost} = O(\text{NCellRefs}) + \text{NUses} \times \text{Level3Cost}$

where: $\text{NCellRefs}$ is the number of cell references in part of the formula part for the
formula graph node being created (For example, a boolean expression for a
predicate node),
$\text{NUses}$ is the number of uses (or unique cell references) in the corresponding
formula part.

5.1.4 Level 1: StaticInfo::NewFormula()

On the topmost level, the StaticInfo::NewFormula() method implemented by
the CRGNode class disposes of the old formula graph and invokes FGNode::Create() to
build a new one. Removal of the old formula graph can be done either by a garbage
collector or explicitly by the system, but will never exceed the worst case complexity of
constructing a new formula graph, and can therefore be ignored in our analysis. Thus, the total cost of Task 1 is:

\[ Task1Cost = Level1Cost = NFGNodes \times Level2Cost \]  

(4)

where: \( NFGNodes \) is the number of formula graph nodes in the formula graph of the new formula.

This expression is dominated by \( Level2Cost \), and can be rewritten so that it does not contain \( NFGNodes \):

\[ Task1Cost = O(NCellRefs) + NUses \times Level3Cost = O(NCellRefs) + O(NUses \times NRegions \times NDefsReferenced) \]  

(5)

where:

- \( NCellRefs \) is the total number of cell references in the new formula,
- \( NUses \) is the total number of uses in the new formula,
- \( NRegions \) is the maximum number of grid regions (statically) referenced by any use in the new formula,
- \( NDefsReferenced \) is the maximum number of definition nodes in any region references in the new formula.

In some cases \( O(NCellRefs) \) may become larger than \( O(NUses \times NRegions \times NDefsReferenced) \). \( NRegions \) and \( NDefsReferenced \) would have to be small while the multiplicities of uses would have to be large, such as in "if(\( m[1@1]=1 \) or \( m[1@1]=10 \) or \( m[1@1]=15 \) or \( m[1@1]=33 \) or (...) ) then 2 else 3". But no matter which of the two halves of (5) is more expensive, \( NCellRefs \) and \( NUses \) are both bounded by the length of the formula, unless the formula contains range references (e.g.: "sum(A1:A10)"). If there are references in range format, then for strict operators, all of \( NUses \) still must be visited by the evaluation engine to calculate the value of at least one element in region receiving the new formula. But for non-strict (lazy) operators, such as "or", the cost of N-uses can exceed the cost of evaluation. It is also worth noting that spreadsheet languages do not provide syntax allowing a single range reference to reference multiple grids, so the \( NCellRefs \) or \( NUses \) in any one range are bounded by the size of the grid. (The number of such range references cannot, of course, exceed the size of the formula.)

\( NRegions \) reflects the homogeneity of the grid. Since the methodology given here is targeted for grids with a large amount of homogeneity, the number of regions in any grid will always be small compared to the size of the grid. (We will denote this quantity as \( \ll |\text{grid}| \).)
**NDefsReferenced** is proportional to the size of a region’s formula, which must be traversed by the formula parser.

Hence, the time cost added to a spreadsheet system by Task 1 is for formulas without range-format referencing:

$$O(1) \times O(\langle\langle|\text{grid}|\rangle) \times O(1)$$  \hspace{1cm} (6)

and for formulas with range-format referencing:

$$O(|\text{grid}|) \times O(\langle\langle|\text{grid}|\rangle) \times O(1)$$  \hspace{1cm} (7)

As explained above, $|\text{grid}|$ is in many cases less than the number of cells already visited by the evaluations invoked by the same trigger.

### 5.2 Task 2: Tracking execution

This task is triggered whenever a cell is evaluated. It is implemented by a probe in the evaluation engine, which adds $O(1)$ to the cost of evaluating a cell. In the Region Representative approach this is guaranteed by the use of hash-tables in maintaining trace information as described in Section 4.3.4.

### 5.3 Task 3: Validation

The validation task is triggered when the user clicks on a validation tab. As previously discussed, it is a graph-walk through dynamically exercised du-associations in the direction opposite to data flow. In terms of cell visits, this process is similar to calculating the value of validated cell. A subtle difference between the two is that cached values of producers may be used when calculating a cell, whereas the validation walk exhaustively pursues the exercised du-associations. Thus, validation is similar in complexity to calculating the value of the validated cell for the first time, when no values of producer cells have yet been cached.
Table 5: Statement groups in CRGNode::Validate. This is the non-recursive part of the validation task. Its purpose is to update each cell's appearance through calls to UpdateGUI.

The validation task consists of a non-recursive and a recursive part. The non-recursive part is implemented by CRGNode::Validate() (described in Section 4.3.5). This method, divided into groups of statements, is shown in Table 5.

Group 1 clearly has $O(1)$ complexity. In Group 2 the recursive part of validation is initiated. Returned is the list of visited tracers affectedTracers. The purpose of Group 3 is to ensure that UpdateGUI message is sent exactly once to each cell affected by the validation. The set of affected cells, however, includes not just the producers of the validated cell, but also all cells that are in the same regions as any of the producers. This is because the effects of a single region element validation propagate to every other cell in that region. Thus, the cost of the non-recursive part of validation is:

$$\text{NonRecursiveCost} = O(\text{NProducers} \times \text{MaxProducerRegionSize})$$  \hspace{1cm} (8)

where: $\text{NProducers}$ is the number of producers of the validated cell (i.e., the size of the backwards dynamic slice),

$\text{MaxProducerRegionSize}$ is the maximum number of cells in the region of any producer.

The recursive part of the validation task is accomplished by RecursiveValidate methods of three classes: CRGTracer, FGNode, and Use given in Section 4.3.5. All of the recursive calls in these methods are tail recursive. Hence, the recursion could be
eliminated by a loop like "for cell and every producer of cell do ... ". Using \( NProducers \) we express the cost of recursive part of Task 3 as follows:

\[
\text{RecursiveCost} = NProducers \times NUses \times O(\text{Use::RecursiveValidate})
\]

(9)

where: \( NUses \) is the maximum number of uses in any of the producers of the validated cell,

\( O(\text{Use::RecursiveValidate}) \) is the cost of \text{Use::RecursiveValidate} excluding tail recursion.

Thus, we bypass \text{RecursiveValidate} methods of \text{CRGTracer} and \text{FGNode} in our analysis, since they add just \( O(1) \) overhead to simply passing the \text{RecursiveValidate} message to the exercised uses. Method \text{Use::RecursiveValidate}, divided into groups of statements, is shown in Table 6.

Group 1 clearly has \( O(1) \) complexity. Group 2, where all constant du-associations coming into this use are validated by a single action, also costs \( O(1) \). In many cases this shortcut will improve the performance of the system, but in the worst case all of the incoming du-associations are non-constant.

```
1 list<CRGTracer> Use::RecursiveValidate(CRGTracer tracer){
   2 list<CRGTracer> results = new list<CRGTracer>;
   3 constantDUPairCore.Validate(tracer);
   4 for each duPair in nonConstantDUPairs do{
      5       CRGTracer definitionTracer = duPair.IsExercised(tracer);
      6       if(definitionTracer != null){
             7           duPair.Validate();
             8           results.Append(definitionTracer.RecursiveValidate());
       9       }
   10  }
  11  return results;
}
```

Table 6 : Statement groups in \text{Use::RecursiveValidate}

The number of non-constant du-associations that the loop in Group 3 is iterating through is bounded by the number of definitions in non-constant producers of the cell this use belongs to.
Group 4 costs \( O(1) \) even though \texttt{DUPair::IsExercised} method may involve a call to \texttt{StaticallyResolve}. In this case a single cell reference that is involved in this use is resolved, which has \( O(1) \) complexity. (See 4.2 for a description of single cell reference \texttt{StaticallyResolve}.)

Since we ignore the recursive calls in this analysis and assume that the \texttt{Append} operation in Group 6 takes \( O(1) \), the time complexity of Group 6 is also \( O(1) \).

Based on the above considerations, the cost of the recursive part of Task 3 becomes:

\[
\text{RecursiveCost} = O(NDUPairs) = O(NProducers \times NUses \times NDefs) \quad (10)
\]

where: \( NDUPairs \) is the maximum number of incoming du-pairs in any of the producers of the validated cell

\( NDefs \) is the maximum number of definitions in any of the producers of the validated cell.

Both \( NUses \) and \( NDefs \) are bounded by formula size (a constant maximum number of characters). Hence \( \text{RecursiveCost} \) reduces to:

\[
\text{RecursiveCost} = O(NProducers \times 1 \times 1) = O(NProducers) \quad (11)
\]

From (8) and (11) we derive the total cost of Task 3:

\[
\text{Task3Cost} = O(NProducers \times \text{MaxProducerRegionSize}) + O(NProducers) \quad (12)
\]

If not for the \( \text{MaxProducerRegionSize} \) factor, Task3Cost would be equal to \( O(NProducers) \), which is similar to the cost of evaluating a cell when there are no relevant cached values. The \( \text{MaxProducerRegionSize} \) is the price of the Region Representative approach advantages (in this case propagation of validation results to all region elements no matter which one has been validated). This is no more expensive than the cost of the use validating each cell in the region individually, or using the straightforward approach.

5.4 Task 4: Adjusting test adequacy information

This task, like Task 1, is triggered when a new formula is entered for a region. The evaluation engine must visit all the consumers of the edited cell for purposes of discarding cached values.
This task is similar to Task 3 in that it is also a graph walk through du-associations, but they are the du-associations of consumers rather than producers, and the trace information does not drive the walk. Despite the difference, the worst case performance of this walk is similar to the one of Task 3, since the worst case of Task 3 occurs when all du-associations in the backward dynamic static slice are exercised and the validation walk has to traverse them. In Task 4, the graph walk always traverses all du-associations in the forward static slice. Thus, replacing $N_{Producers}$ with $N_{Consumers}$ in (10) yields:

$$\text{Task4Cost} = O(N_{Consumers} \cdot N_{Defs} \cdot N_{Uses})$$  \hspace{1cm} (13)

which similarly to (10) reduces to:

$$\text{Task4Cost} = O(N_{Consumers} \cdot 1 \cdot 1) = O(N_{Consumers})$$  \hspace{1cm} (14)

Thus, the cost added by this task is $O(1)$ to the other work that must be performed even without any testing subsystem in place.
Chapter 6: Performance Experiment

To investigate how the Region Representative approach compares in performance to the Straightforward approach we ran a series of tests on the spreadsheet shown in Figure 15.

Figure 15: 10-student version of the experimental MatrixGrades spreadsheet. The actual spreadsheet on which the experiment had been conducted contained data for 100 students. Student names are not shown for brevity.

The MatrixGrades spreadsheet computes a course letter grade (A, B, C, D, or F) from 4 quiz scores and one extra credit score. The letter grade is A, B, C, or D if the total score is greater than or equal to 90, 80, 70, or 60 respectively and is an F if the total score is less than 60. The total score is the sum of the average of the three highest quiz scores, the points awarded based on the extra credit score, and bonus points.

Inputs to the spreadsheet are four quiz scores and one extra credit score. A student is awarded 0 extra credit points if the extra credit score is not greater than 20, 3
extra credit point if the extra credit score is not greater than 25 and 5 extra credit points if the extra credit score is greater than 25. If a student shows consistent improvement in the 4 quiz scores 5 bonus points are awarded. The quiz and extra credit scores range from 0 to 100.

We compared both actual timings and data structure operations as system-independent measures. All timings are averages of ten consecutive runs, and were taken using Forms/3 on a Sun UltraSparc with one user, under Liquid Common Lisp 5.0.3 with Garnet [10].

<table>
<thead>
<tr>
<th>Region Representative Approach (total for 5 formula edits)</th>
<th>Straightforward Approach (total for 5 formula edits)</th>
<th>Region Representative Approach (average for 1 formula edit)</th>
<th>Straightforward Approach (average for 1 formula edit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1 in seconds</td>
<td>0.3269</td>
<td>5.0367</td>
<td>0.0653</td>
</tr>
<tr>
<td>Task 4 (recursive part) in seconds</td>
<td>0.0224</td>
<td>0.3182</td>
<td>0.0044</td>
</tr>
<tr>
<td>Task 4 (non-recursive part with GUI updates) in seconds</td>
<td>11.4322</td>
<td>11.2614</td>
<td>2.28644</td>
</tr>
<tr>
<td>CFGNodes created</td>
<td>21</td>
<td>2100</td>
<td>4.2</td>
</tr>
<tr>
<td>DUPairs created</td>
<td>827</td>
<td>4500</td>
<td>165.4</td>
</tr>
<tr>
<td>CFGNodes invalidated</td>
<td>70</td>
<td>7300</td>
<td>14</td>
</tr>
<tr>
<td>DUPairs invalidated</td>
<td>58</td>
<td>6200</td>
<td>11.6</td>
</tr>
</tbody>
</table>

Table 7: Timings (averaged over 10 runs) and system independent measures for Tasks 1 and 4.
In the first part of our experiment, all non-constant formulas in the experimental spreadsheet were re-entered, thus simulating formula edits and triggering Tasks 1, 2, and 4. Task 2 is not measured; it is very small, and there is no significant difference in its performance under the Region Representative versus Straightforward approaches. Results for Tasks 1 and 4 are shown in Table 7. The non-recursive part of Task 4 is dominated by GUI updates while the recursive part of Task 4 represents the graph walk. Although the total time to complete Tasks 1 and 4 was dominated by GUI updates, this is mainly due to the fact that our GUI updating is not "smart", and even updates cells that are not on the screen.

Table 8 summarizes the user’s actions required to test the experimental spreadsheet under the Region Representative and Straightforward approaches.

<table>
<thead>
<tr>
<th></th>
<th>Region Representative approach</th>
<th>Straightforward approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of students’ data values to enter</td>
<td>500 (only the original data values)</td>
<td>900 (500 original data values plus at least one value for each test case for each student)</td>
</tr>
<tr>
<td>Number of user validation actions</td>
<td>5 (to achieve maximum coverage)</td>
<td>5-500 (could rubberband entire region to validate 100 changes at once, or could validate after each student’s new test data is entered)</td>
</tr>
</tbody>
</table>

Table 8 : Summary of user actions to validate the experimental spreadsheet.

By validating cells LetterGrade[1 @ 1] ... LetterGrade[4 @ 1] and LetterGrade[6 @ 1] the user will achieve maximum coverage of the experimental spreadsheet under the Region Representative approach. Under the Straightforward approach to force the
execution in each row of cells through the same five test cases the user will have to do at least 500 formula edits. In order to achieve maximum coverage, each cell in the LetterGrade column will have to be validated 5 times, thus bringing the number to 500 validations. Since the user is allowed to select a set of cells and validate them with one click, the number of physical mouse clicks required to perform the 500 validations may be reduced to 10 (5 for selections and 5 for actual validations). However, the amount of computations the system has to perform remains equivalent to 500 single-cell validations.

Table 9 shows the results of our performance experiment for Task 3. The cost of the recursive part of one call to validation under the Region Representative approach is slightly more than that under the Straightforward approach, which we attribute to the use of StaticallyResolve during validation.

<table>
<thead>
<tr>
<th>Region Representative Approach (1 validation)</th>
<th>Straightforward Approach (1 validation)</th>
<th>Straightforward Approach (100 validations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 3 (non-recursive part with GUI updates) in seconds</td>
<td>1.5752</td>
<td>0.0153</td>
</tr>
<tr>
<td>Task 3 (recursive part) in seconds</td>
<td>0.0122</td>
<td>0.0113</td>
</tr>
<tr>
<td>CFGNodes validated</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>DUPairCores validated</td>
<td>24</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 9 : Timings (averaged over 10 runs) and system-independent measures for Task 3. Note that in the Region Representative approach one validation covers the same test case for all cells in a region. One validation under the Straightforward approach covers only one cell. 100 validations under the Straightforward approach can be triggered by one user action, but accomplish the same amount of testing as one validation under the Region Representative approach.
Although the timings of the two approaches appear about the same, due to the cost of GUI updates, this is because the GUI updating routine naively updates even off-screen appearances. The Region Representative approach is much more scalable than the Straightforward approach as evidenced by the data presented in this chapter. With introduction of a “smart” GUI, the updates would be run for only cells that are visible on the screen, thereby bounding this part by the number of pixels on the screen. In this situation, the time and space savings provided by the Region Representative approach in the reasoning (recursive) parts, will become more and more important to maintaining the system performance as the size of spreadsheets increases.
Chapter 7: Conclusion

In this thesis we have presented two entirely visual approaches to testing spreadsheet grids. The approaches presented incorporate the homogeneity of spreadsheet grids into the system's reasoning and the user's interactions about testedness.

Both the Straightforward and the Region Representative approaches allow one user validation action to be leveraged across the entire region, which reduces user actions. However, the Straightforward approach has critical disadvantages that are solved by the Region Representative approach. From the perspective of the work the user must do to test the spreadsheet, the Straightforward approach requires significantly more manual test generation, which in large spreadsheets may render the use of the testing subsystem impractical. The Region Representative approach by sharing more information about testedness among cells with the same formulas, allows to use the test cases already present in the spreadsheets. This method drastically reduces the required amount of manual test generation.

From the perspective of system performance, the Region Representative approach reduces the system time required to maintain testedness data, so that it removes the dependency of system time on grid region size. Thus, it is scalable to very large grids. This is key to maintaining the high responsiveness that is expected in spreadsheet languages.

A key element of future work is to add intelligence to the GUI updating routines to repaint only on-screen cells and bound this task by the number of pixels on the screen.

Both approaches to testing are designed for tight integration into the environment, with the only visible additions being checkboxes and coloring devices. There are no testing vocabulary words, such as "du-association" displayed, no CRG graphs displayed, no dialog boxes about testing options, and no separate windows of testing results. This design reflects the goal of our research into testing methodologies for this kind of language, which is to bring at least some of the benefits that can come from the application of formal testing methodologies to spreadsheet users.
Bibliography


