How can rigorous forms of testing be supported in a way that is both compatible with the visual aspect of visual programming languages, and usable by the audiences using those languages — even when the audience has no background in software engineering? Visual programs are likely to contain at least some errors, and supporting a visual form of testing would give users a way to spot those errors early in the program's life. In previous work, we have developed a visual testing methodology known as WYSIWYT, for use in visual spreadsheet languages, and in this work, we show formally that this methodology can be generalized to screen transition diagrams. The algorithms and accompanying proof of the coverage equivalence that they ensure provide the mechanisms needed for the screen transition paradigm to incorporate WYSIWYT testing for both professional and end-user programming audiences.
Generalizing WYSIWYT For Use in the Screen Transition Paradigm

by

Darren Brown

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed May 13, 2003
Commencement June 2004
Master of Science thesis of Darren Brown presented on May 13, 2003

APPROVED:

Redacted for Privacy

Co-Major Professor, representing Computer Science

Redacted for Privacy

Co-Major Professor, representing Computer Science

Redacted for Privacy

Chair of the Department of Computer Science

Redacted for privacy

Dean of the Graduate School

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

Redacted for Privacy

Darren Brown, Author
ACKNOWLEDGMENTS

I would like to thank Margaret Burnett and Gregg Rothermel for their seemingly infinite patience with myself and my error-prone ways.

I would also like to thank Saurabh Sethia for his help with some of the proofs.

I would like to thank Erin van Olden, for allowing me to place her second on my priority list on so many occasions. And along the same lines, I would like to thank Kevin McCluskey, who gets pushed even further down the priority queue and takes it in stride.

I also would like to thank my family, Michael, Marji, Ryan, Heather, Jennifer, Steve, Wyatt and Gavin, for really coming together over the past couple years and making trips home so very relaxing. And I would like to thank Wyatt, for teaching me patience. Still have a ways to go in that category though.

I would like to thank Pat, Holly, and Cam, for being there when I needed them.

I would like to thank Josh, for his aid in the acquisition of my cottage.

I would also like to thank Dragos Margineantu, Alexey Malishevsky, Brian Lindsay, Marc Fisher II and Gregg Rothermel for aiding me in the discovery and use of various thesis templates for this document.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1: Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2: Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 The WYSIWYT Testing Methodology</td>
<td>5</td>
</tr>
<tr>
<td>2.2 The Screen Transition Paradigm for WYSIWYT</td>
<td>11</td>
</tr>
<tr>
<td>Chapter 3: Testing Screen Transition Diagrams</td>
<td>16</td>
</tr>
<tr>
<td>3.1 WYSIWYT Testing a Screen Transition Diagram</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Issues Introduced by WYSIWYT for Screen Transition Diagrams</td>
<td>25</td>
</tr>
<tr>
<td>Chapter 4: Translating Screen Transition Diagrams to CRGs</td>
<td>26</td>
</tr>
<tr>
<td>4.1 The Translation Method</td>
<td>26</td>
</tr>
<tr>
<td>4.2 Comparing Du-Associations in CRGs and Screen Transition Diagrams</td>
<td>31</td>
</tr>
<tr>
<td>Chapter 5: Coverage Equivalence of Translated Programs</td>
<td>35</td>
</tr>
<tr>
<td>5.1 Subsumption within the spreadsheet paradigm</td>
<td>36</td>
</tr>
<tr>
<td>5.1.1 Coverage Equivalence without conditionals</td>
<td>36</td>
</tr>
<tr>
<td>5.1.2 Coverage Equivalence with Conditionals</td>
<td>40</td>
</tr>
<tr>
<td>5.2 Subsumption in the Screen Transition Paradigm</td>
<td>43</td>
</tr>
<tr>
<td>Chapter 6: Conclusions and Future Work</td>
<td>46</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>----</td>
</tr>
<tr>
<td>Bibliography</td>
<td>49</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Visual depiction of testedness of a student grades spreadsheet. Check marks were placed by the user to indicate that a value is correct, and question marks point out the cells in which check marks would increase testedness according to the adequacy criterion.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>A partial cell relation graph for the spreadsheet of Figure 2.1, showing the formula graphs for the top row (&quot;Abbott, Mike&quot;). Dashed arrows indicate dataflow edges between cells’ formula nodes. For clarity in this figure, we preface cell names with “Abbott” instead of with the internal IDs actually used.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>A SILK sketch (front) of a five-day weather forecast and storyboard (rear) [23]. An experienced user-interface designer created the sketch, including buttons, and arrows that show the screen transitions that occur when the user presses the buttons.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>An example in a (hypothetical) language of what the grades screen transition diagram might look like if the condition (bottom) is met. The testedness indicator depicts the ratio of validated du-pairs in the spreadsheet. The center window shows the current screen. From Transitions (left) are all transitions that can reach the current screen and To Transitions (right) are all transitions that can be exercised from the current screen. Transition Taken (bottom) is the transition most recently exercised.</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>The user has changed the boolean EC to false. The system tells the user that this situation has not been fully validated by placing a “?” in objects in which testedness can be increased through validation.</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>The user has marked the value in object Total as validated. Testedness has increased.</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>An example screen transition diagram. Screens names are represented with rectangles, input objects are represented with soft-cornered rectangles and ovals, and output objects are represented with hexagons.</td>
<td>20</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.4</td>
<td>Algorithm sketch for one run of a WYSIWYT test generator for screen transition diagrams.</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Translating a screen transition diagram to FAR rules.</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>FAR rules derived by applying the algorithm in Figure 4.1 to the screen transition diagram of Figure 2.4 and Table 2.1.</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Translating FAR rules to a spreadsheet.</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>Sketch of Grades translated to a spreadsheet via FAR rules. Cells show values, each cell’s name is above the cell, and each formula is depicted at the cell’s lower right.</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Grades CRG. Nodes 20 and 29 have been introduced because all arrows to exit nodes must begin at computation nodes to ensure a well-formed spreadsheet program. Predicate nodes with undefined branches must define the cell with a well-defined value as “undefined”.</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>Examples of a line, fork and join in the spreadsheet paradigm. Each cell is named with index notation (e.g. P[0]). Below the name is a box containing the cell’s value, and below the value is a box containing the cell’s formula. Arrows represent dataflow dependencies between cells.</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Definition-use association types: This is a modified CRG to highlight the definition-use types added by conditionals. We began with a cell relation graph, removing the cell dependency arrows. We then removed the entry and exit nodes. Finally, we added explicit depictions of definition-uses with dashed arrows.</td>
<td>42</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Transitions for example screen transition diagram grades</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Transitions/actions for the screen transition diagram depicted in Figure 3.3.</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>(&quot;NAN&quot; stands for &quot;not a number&quot;)</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>Definition-use associations in the Calc example</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>The beginning of one possible run of a WYSIWYT-based test generation</td>
<td>24</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Visual programming languages (VPLs) are becoming increasingly common in several domains. For example, visual programming languages or sublanguages are becoming the most common way to do some kinds of GUI programming, the most common way of specifying visualization graphics depicting scientific data, and a common vehicle for macro generation for end-user applications. However, despite the increase in the use of VPLs for these and other programming tasks, there has been almost no attention to providing software engineering support mechanisms to programmers working in these languages.

Two issues relevant to VPLs have implications for software engineering in VPLs. The first is diversity of audience: while some users of VPLs are professional programmers, some are end users with no training in professional software engineering notions and methods. The second is the need to develop rigorous testing approaches that are fully compatible with the non-traditional paradigms and mechanisms used in VPLs, such as the specification of program semantics by directly manipulating objects or demonstrating with concrete examples.

We have previously worked to bring some of the benefits of applying for-
malized notions of testing to the informal, incremental, development world of spreadsheet-like VPLs through a highly interactive visual testing mechanism known as the "What You See Is What You Test" (WYSIWYT) methodology [12, 34, 32]. The methodology is completely visual, and is designed to support end users as well as more sophisticated programmers.

WYSIWYT has mainly been explored in the spreadsheet paradigm. There has also been work to adapt it to the dataflow paradigm [20], but since "underneath the hood" the spreadsheet paradigm uses a dataflow evaluation engine, this adaptation does not prove very much about the potential generality of WYSIWYT. In this work, we consider whether WYSIWYT can be used for another visual paradigm, namely, the screen transition paradigm.

The screen transition paradigm uses screen transition diagrams to specify program behavior. These are an adaptation of state transition diagrams in which output states are represented via screen contents (and report contents and database contents), and transitions among these output states specify the conditions under which state changes occur. Screen transition diagrams are the primary communication device by which customer requirements are entered into the Lyee methodology [27, 37, 36], a program generation facility used by a major Japanese software corporation\(^1\) for commercial software development. The real-world needs of the Lyee methodology provide the context for the work reported here. In addition, the screen transition paradigm has been prototyped in a system called SILK for GUI development, and SILK empirical work shows that it can be effectively used by end-user programmers [23]. Finally, as a

\(^1\) Catena Corporation, Tokyo.
screenshot of SILK will demonstrate, the screen transition paradigm bears a visual similarity to another important VPL paradigm, namely, the visual rule-based paradigm. The similarity between these two paradigms, which has been exploited in part by Altaira [30] and Kara [17], implies that our findings for the screen transition paradigm may well extend to visual rule-based languages as well.

For end-user testing to become a reality in the screen transition paradigm, many questions relating to the human aspects of this goal must also be investigated, such as:

- Is it feasible to expect end users to enter software requirements at all?
- If so, to what level of detail must they descend to create requirements that are complete enough to generate the desired software?
- How can we ensure that requirements entered by such users are non-contradictory?
- Can end users understand the implications of the requirements they are entering well enough to recognize errors and correct them?

There is preliminary research contributing partial answers to the above questions, some of which is discussed in [7]. These partial results allow optimism that the questions above can be resolved. But even if they have been solved, at least one question remains:

- How can end users test the requirements they enter? Specifically, can a testing methodology that has previously been developed for end users,
known as the WYSIWYT methodology, be adapted to the screen transition paradigm?

Addressing this question is the subject of this work. This work makes the following specific contributions:

1. We present a structure of requirements that is based on an approach of proven usefulness to end users.
2. We show that WYSIWYT test adequacy criteria can be used with such a structure.
3. We present a systematic translation from the structure above to the formal model underlying WYSIWYT.
4. We show that this translation process does not affect testing using the all du-associations testing criterion.

We begin our consideration of generalizing WYSIWYT to screen transition languages by providing the necessary background in summarizing the WYSIWYT methodology and the screen transition paradigm. We then define the properties of screen transition diagrams necessary for WYSIWYT, and sketch WYWISYT's visual aspects in this paradigm as a concrete example. Finally, we explore the viability of this approach through translation to a formalism and verification of that translation from a testing perspective.

---

2 This work draws significantly from [4, 5, 6]
CHAPTER 2

BACKGROUND

In this section, we discuss the WYSIWYT testing methodology and some visual programming paradigms including the screen transition paradigm.

2.1 The WYSIWYT Testing Methodology

In previous work [12, 32, 33, 34], we presented a testing methodology for spreadsheets termed the "What You See Is What You Test" (WYSIWYT) methodology. The WYSIWYT methodology provides feedback about the "testedness" of cells in spreadsheets in a manner that is incremental, responsive, and entirely visual. It is aimed at a wide range of spreadsheet users, including both end users and professional programmers. We have performed extensive empirical work, and our results consistently show that both end users and professional programmers test more effectively and efficiently using WYSIWYT than they do unaided by WYSIWYT (e.g., [13, 21, 32, 35]).

The underlying assumption behind the WYSIWYT methodology has been that, as the user develops a spreadsheet incrementally, he or she could also be testing incrementally. We have integrated a prototype of WYSIWYT into our research spreadsheet language Forms/3 [8, 10]. In our prototype, each 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 or if-expressions), which do not require testing. For the non-input cells, testedness is reflected via border colors on a continuum from untested (red, or light gray in this work) to tested (blue, or black in this work). Figure 2.1 shows a spreadsheet used to calculate student grades in Forms/3. The spreadsheet lists several students, and several assignments performed by those students. The last row in the spreadsheet calculates average scores for each assignment, the rightmost column calculates weighted averages for each student, and the bottom right cell gives the overall course average (formulas not shown).

With WYSIWYT, the process of testing spreadsheets such as the one in Figure 2.1 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 decision by validating the correct cell (clicking in the decision check box in its right corner), which causes a check mark to appear, as shown in Figure 2.1. This communication lets the system track judgments of correctness, propagate the implications of these judgments to cells that contributed to the computation of the validated cell’s value, and 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 knows which ones those are, and re-colors their borders more untested (more red). In this document, we depict red as light gray, blue as black, and the colors between the red and blue endpoints of the continuum as shades of gray.

WYSIWYT is based on an abstract testing model we developed for spreadsheets called a cell relation graph (CRG) [33]. A CRG is a pair \((V, E)\), where
V is a set of formula graphs and E is a set of directed edges modeling dataflow relationships between pairs of elements in V. A formula graph models flow of control within a single cell’s formula, and is comparable to a control flow graph. In simple spreadsheets, there is one formula graph for each cell (see [11, 12] for discussions of how complex spreadsheets are treated.) For example, Figure 2.2 shows a portion of the CRG for Figure 2.1’s cells, represented by dotted rectangles.

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. In the formula 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 are predicate nodes (represented as rectangles). 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.

We used the CRG model to define a test adequacy criterion for spreadsheets [32]. (A test adequacy criterion is a definition of what it means for a program to be tested “enough.”) Our du-adequacy criterion is a type of dataflow adequacy criterion [16, 24, 31, 38]. Such criteria relate test adequacy to interactions between definitions and uses of variables in source code (definition-use associations, abbreviated du-associations). In spreadsheets, cells play the role of variables; A definition of cell C is a node in the formula graph for C representing an expression that defines C’s value, and a use of cell C is either a computation use (a non-predicate node that refers to C) or a predicate use (an
FIGURE 2.1: Visual depiction of testedness of a student grades spreadsheet. Check marks were placed by the user to indicate that a value is correct, and question marks point out the cells in which check marks would increase testedness according to the adequacy criterion.
FIGURE 2.2: A partial cell relation graph for the spreadsheet of Figure 2.1, showing the formula graphs for the top row ("Abbott, Mike"). Dashed arrows indicate dataflow edges between cells' formula nodes. For clarity in this figure, we preface cell names with "Abbott" instead of with the internal IDs actually used.
out-edge from a predicate node that refers to C). For example, in Figure 2.2, nodes 2, 5, 8, 12, and 13 are definitions of their respective cells, nodes 12 and 13 are computational uses of the cells referenced in their expressions, and edges (11,12) and (11,13) are predicate uses of the cells referenced in predicate node 11. Under this criterion, a cell X will be said to have been tested enough when all of its du-associations have been covered (executed) by at least one test. In this model, a test is a user decision (which is communicated via a checkmark) that a particular cell contains the correct value given the inputs upon which it depends. Decisions are communicated to the system when the user checks off a cell to validate it. Thus, given a cell X that references Y, du-adequacy is achieved with respect to the interactions between X and Y when each of X's uses of each definition in Y has been covered by a test.

Thus, if the user manages to turn all the red (light gray) borders blue (black), the du-adequacy criterion has been satisfied. This may not be achievable, since not all du-associations are executable in some spreadsheets (termed infeasible). Even so, subjects in our empirical work have been more likely to achieve du-adequate coverage and do so efficiently when using the WYSIWYT methodology than those not using it [21, 35], du-adequate test suites have frequently been significantly more effective at fault detection than random test suites [32], and subjects have been significantly more likely to correctly eliminate faults using the WYSIWYT methodology than those not using it [13]. Both programmer [13, 35] and end-user [21] audiences have been studied.
2.2 The Screen Transition Paradigm for WYSIWYT

In the screen transition paradigm, the general idea is that a user can visually depict input- and output-based states by explicitly sketching how the intended screens, reports, and databases appear and behave. A screenshot from the SILK system for user-interface specification [22, 23] is shown in Figure 2.3. The difference between such depictions and traditional state machines is that in the screen transition paradigm, the event and conditions that are required to execute the actions are only a partial specification of state.

Those familiar with visual rule-based languages will note the visual similarity between Figure 2.3 and rule-based languages such as KidSim/Cocoa/Stagecast [18]. The former's screen includes the latter's graphical preconditions, the former's events and conditions correspond to the latter's additional preconditions, the former's transition out-arrows are shown as "then" arrows in the latter, and the former's screen at the end of the arrow along with its transition actions represents the latter's postconditions. Altaira is a rule-based VPL designed specifically for the navigation of small robotic devices. The basic premise of the rule-based paradigm is that the preconditions for the execution of a rule are represented as the left hand side (LHS) of an expression and the actions executed given the preconditions are represented on the right hand side (RHS) of the same expression. In Table 2.1, columns Screen, Event and Predicate represent the preconditions of an action, the LHS of a rule-based expression. Dest and Actions are the resulting actions and state changes, or the RHS. This similarity extends below the surface, and we will make use of this fact in Chapter 4.

We are assuming that end users will enter a complete set of requirements, without the help of a professional developer. Thus, testing the screen transition
diagram is testing the program—because there will be no information in the program that was not generated by the user’s screen transition diagrams. (We simply assume that this is true for now; future research may design exactly how this will work.)

To consider how WYSIWYT might be applied in the screen transition paradigm, we begin by providing terminology for the elements of screen transition diagrams.

![Figure 2.3: A SILK sketch (front) of a five-day weather forecast and storyboard (rear) [23]. An experienced user-interface designer created the sketch, including buttons, and arrows that show the screen transitions that occur when the user presses the buttons.](image)

With screen transition diagrams, the user specifies a program using screens, objects on those screens, and transitions. (We are attempting to relate only what the user specifies, not how — the scope of this work is testing, not the
design of a screen transition language.) A screen is a window containing a formatted collection of objects, whose values will be used and/or produced by computations. For example, in Figure 2.4, a screen is the center window, containing the screen name and the objects that are on the screen. The transitions corresponding to the numbered arrows are given in Table 2.1.\textsuperscript{1}

<table>
<thead>
<tr>
<th>#</th>
<th>Screen</th>
<th>Event</th>
<th>Predicate</th>
<th>Dest</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Main</td>
<td>Edit any object</td>
<td>(EC==0 AND ![Error? Asgn0] AND ![Error? Asgn1])</td>
<td>Main</td>
<td>Asgn = Asgn0 + Asgn1 Total = Asgn</td>
</tr>
<tr>
<td>2</td>
<td>Main</td>
<td>Edit any object</td>
<td>(EC==1 AND ![Error? Asgn0] AND ![Error? Asgn1])</td>
<td>Main</td>
<td>Asgn = Asgn0 + Asgn1 Total = Asgn + AsgnEC</td>
</tr>
<tr>
<td>3</td>
<td>Main</td>
<td>Edit any object</td>
<td>(Error? Asgn0) OR (Error? Asgn1)</td>
<td>Main</td>
<td>Asgn = Error Total = Error</td>
</tr>
<tr>
<td>4</td>
<td>Main</td>
<td>Edit any object</td>
<td>Else</td>
<td>Main</td>
<td>Asgn = Undefined Total = Undefined</td>
</tr>
</tbody>
</table>

TABLE 2.1: Transitions for example screen transition diagram grades

There are two types of objects: input objects and output objects. Input objects are objects into which the user is allowed to enter input, via the keyboard or the mouse, but they are also updatable by the program. A specialized kind of

\textsuperscript{1} We emphasize that the transition format shown throughout this article is only for precision of this discussion, and is not suitable for end users. As Pane and Myers showed empirically [28], end users are not very successful at using Boolean AND and OR, and do not tend to understand the use of parentheses as ways to specify precedence. They suggest some alternatives to these constructs, and empirically show that end users can use one set of such alternatives successfully [29]. In addition to their suggested alternatives, other possibilities include the demonstrational rewrite rules of Cocoa [18] or Visual AgentTalk [19], which have been demonstrated to be usable by end users.
FIGURE 2.4: An example in a (hypothetical) language of what the grades screen transition diagram might look like if the condition (bottom) is met. The testedness indicator depicts the ratio of validated du-pairs in the spreadsheet. The center window shows the current screen. From Transitions (left) are all transitions that can reach the current screen and To Transitions (right) are all transitions that can be exercised from the current screen. Transition Taken (bottom) is the transition most recently exercised.
input object is an event object, which generates a user event if the user interacts with it. Output objects cannot receive user inputs. Instead, their purpose is to receive the results of computations, but they can also provide values to other computations. Transitions can be defined by the tuple (source screen, destination screen, event, condition-action pairs). As such, transitions are a slightly more powerful form of transition than is traditional. In the tuple, source screen and destination screen enumerate the screens the transition connects. Event is a user or computational event under which the transition fires: the condition-action pairs are processed and the destination screen is displayed. Each condition-action pair contains a condition and a corresponding action to take if the condition is fulfilled. A condition is any arbitrary predicate, and each action consists of zero or more assignments to input or output objects.
CHAPTER 3
TESTING SCREEN TRANSITION DIAGRAMS

3.1 WYSIWYT Testing a Screen Transition Diagram

As explained earlier, the WYSIWYT methodology uses du-adequacy as its test adequacy criterion. The following define "definitions" and "uses" in the screen transition paradigm in a manner that parallels those in Section 2.1, leading to a parallel notion of du-adequacy in this paradigm:

Definition 3.1.1 A definition of input object $A$ is:

- the specification of $A$ as an input value (including its initial value and any future values input), or
- an assignment to $A$ in an action (presence of $A$ in the action's left-hand side).

Definition 3.1.2 A definition of output object $A$ is:

- the specification of $A$'s initial value, or
- an assignment to $A$ in an action ($A$ is in the action's left-hand side).

Definition 3.1.3 A use of object $A$ is:

- a reference to $A$ the right-hand side of an action (computational use), or
- a reference to $A$ in a condition (predicate use).
Building on these definitions in the same manner as in Section 2.1, du-associations are interactions between definitions and uses, and the definition of du-adequacy, as in the spreadsheet paradigm, is to cover each du-association by at least one test. Using this model, WYSIWYT’s dataflow-based testing techniques can be employed to try to cover each du-association. As with WYSIWYT, the user interactively communicates whether, given a particular set of inputs, results are correct.

How might a user see testing in this paradigm? Figure 2.4 sketches a WYSIWYT interface. Note that, to be compatible with the WYSIWYT methodology, two constraints must be maintained: (1) the presentation of testedness must be integrated with the screen transition diagrams, as in the original WYSIWYT methodology, and (2) any update or test made by the user must be immediately and visually reflected in the presentation.

In Figure 2.4, the user has assigned values to input objects Asgn0, Asgn1, and AsgnEC, and set the boolean EC to true. The Transition Taken window shows that transition #2 is the transition previously taken. (A complete list of transitions is given in Table 2.1.) The user has then validated object Total which is denoted by the checkmark in the object. In validating object Total, six of the 21 total du-associations have been covered and thus the Testedness window shows the screen transition diagram as being 28% tested. To test further, the user then sets the boolean object EC to false, thus exercising transition #1. This new situation is shown in Figure 3.1. Now there is a “?” in each object in which validation will increase testedness. In comparing the program specifications to the values produced on Screen Main, the user again validates Total, covering seven more du-associations. Figure 3.2 shows that testedness has reached 62%, and because there are no more “?”s on Screen Main, the user
knows that different values are required to further increase testedness. The user is brought closer to the testing criterion in that it is used directly as the testedness metric.

FIGURE 3.1: The user has changed the boolean EC to false. The system tells the user that this situation has not been fully validated by placing a "?" in objects in which testedness can be increased through validation.

Also, the search for new untested situations could be automated in a similar manner as in “Help Me Test” for spreadsheets. Figure 3.4 sketches the algorithm for this aspect of the methodology. We also adopt WYSIWYT’s notion that testing is adequate when all feasible definition-use associations have been exercised by at least one test. The screen transition diagram model in Figure 3.3 and Table 3.1 serve as an example for a mock run of our automatic test case
FIGURE 3.2: The user has marked the value in object Total as validated. Testedness has increased.

generation algorithm.

The algorithm in Figure 3.4 covers only one testing situation, namely that in which the user has asked the system to suggest possible input values (termed the “Help Me Test” feature in our Forms/3 prototype of WYSIWYT) [15]. As in our previous work, this feature will work in tandem with the user’s ability to specify their own input values when they prefer. Also as in the original WYSIWYT methodology, feedback given to the user under this approach will be entirely visual, using devices such as those used in WYSIWYT. Our collaborators are currently in the process of analyzing empirical data about the way end users make use of “Help Me Test”; early indications suggest that the feature positively impacts users’ abilities to find faults.
FIGURE 3.3: An example screen transition diagram. Screens names are represented with rectangles, input objects are represented with soft-cornered rectangles and ovals, and output objects are represented with hexagons.

1. Pick a transition $T$; let $O$ be the set of objects with uses in $T$.
2. Find definitions for $O$, whose du-associations are not yet covered. Set new specific values for these definitions as necessary.
3. Keep setting definitions’ values until $T$ is traversed.
4. Allow Oracle to say if, given current definitions and transition $T$, the output is correct.

FIGURE 3.4: Algorithm sketch for one run of a WYSIWYT test generator for screen transition diagrams.
<table>
<thead>
<tr>
<th>Transition number</th>
<th>Screen</th>
<th>Event</th>
<th>Condition(s)</th>
<th>Destination Screen</th>
<th>Action(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Calc::Main</td>
<td>Pressed calculate object</td>
<td>AND ( (input_1 != NAN) (input_2 != NAN) OR (operator != /) (AND (operator == /) (input_2 != 0) ) )</td>
<td>Calc::Exec</td>
<td>output_1 = input_1 operator input_2</td>
</tr>
<tr>
<td>2</td>
<td>Calc::Main</td>
<td>Pressed calculate object</td>
<td>(input_1 == NAN)</td>
<td>Calc::Error</td>
<td>Error_Disp = &quot;input 1 is not a number&quot;</td>
</tr>
<tr>
<td>3</td>
<td>Calc::Main</td>
<td>Pressed calculate object</td>
<td>(input_2 == NAN)</td>
<td>Calc::Error</td>
<td>Error_Disp = &quot;input 2 is not a number&quot;</td>
</tr>
<tr>
<td>4</td>
<td>Calc::Main</td>
<td>Pressed calculate object</td>
<td>AND ( (operator == /) (input_2 == 0) )</td>
<td>Calc::Error</td>
<td>Error_Disp = &quot;divide by zero&quot;</td>
</tr>
<tr>
<td>5</td>
<td>Calc::Error</td>
<td>Pressed return object</td>
<td></td>
<td>Calc::Main</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Calc::Exec</td>
<td>Pressed return object</td>
<td>(clear == &quot;yes&quot;)</td>
<td>Calc::Main</td>
<td>input_1 = &quot;null&quot; input_2 = &quot;null&quot; operator = &quot;null&quot;</td>
</tr>
<tr>
<td>7</td>
<td>Calc::Exec</td>
<td>Pressed return object</td>
<td>OR ( (clear == &quot;no&quot;) (clear == &quot;null&quot;) )</td>
<td>Calc::Main</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 3.1: Transitions/actions for the screen transition diagram depicted in Figure 3.3. ("NAN" stands for "not a number").
Note that the algorithm allows the user to validate the value, but it does not actually require the user to do anything. This is an important aspect of our approach, and follows principles implied in Blackwell’s end-user programming model of attention investment [3]. That is, our approach does not attempt to alter the user’s work priorities by requiring them to answer dialogues about correctness, because the user might find that counter-productive and stop using the feature. Rather, our approach provides opportunities (through decorating the diagrams with clickable objects) for the user to provide information if they choose. This allows the user to take the initiative to validate, but does not interrupt them from their current processes by requiring information at any particular time. For example, this allows the user to fix a fault as soon as a test reveals it, if they immediately spot the cause, rather than requiring them to continue with additional test values.

The screen transition diagram in Figure 3.3 has 27 du-associations, which are listed in Table 3.2. The algorithm of Figure 3.4 begins by picking a transition, and tries to cover the du-associations associated with that transition. To do so, it sets values of associated definitions until the condition for taking the transition is met. It then visibly executes the program given these values, and provides the user an opportunity to pronounce the demonstrated behavior correct for these inputs (i.e., to validate). If the user validates, then the objects, du-associations, and transitions are colored closer to the testedness color (blue in our previous prototypes). One possible run of this method on Calc might start out as in the sequence of Table 3.3.
<table>
<thead>
<tr>
<th>DU-assoc. number</th>
<th>Definition</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input_1 (as program input)</td>
<td>Transition 1 action</td>
</tr>
<tr>
<td>2</td>
<td>Input_1 (resulting from Transition 6 action)</td>
<td>Transition 1 action</td>
</tr>
<tr>
<td>3</td>
<td>Input_2 (as program input)</td>
<td>Transition 1 action</td>
</tr>
<tr>
<td>4</td>
<td>Input_2 (resulting from Transition 6 action)</td>
<td>Transition 1 action</td>
</tr>
<tr>
<td>5</td>
<td>Operator (as program input)</td>
<td>Transition 1 action</td>
</tr>
<tr>
<td>6</td>
<td>Operator (resulting from Transition 6 action)</td>
<td>Transition 1 action</td>
</tr>
<tr>
<td>7</td>
<td>Input_1 (as program input)</td>
<td>Transition 2 condition</td>
</tr>
<tr>
<td>8</td>
<td>Input_1 (resulting from Transition 6 action)</td>
<td>Transition 2 condition</td>
</tr>
<tr>
<td>9</td>
<td>Input_2 (as program input)</td>
<td>Transition 3 condition</td>
</tr>
<tr>
<td>10</td>
<td>Input_2 (resulting from Transition 6 action)</td>
<td>Transition 3 condition</td>
</tr>
<tr>
<td>11</td>
<td>Operator (as program input)</td>
<td>Transition 4 condition</td>
</tr>
<tr>
<td>12</td>
<td>Operator (resulting from Transition 6 action)</td>
<td>Transition 4 condition</td>
</tr>
<tr>
<td>13</td>
<td>Input_2 (as program input)</td>
<td>Transition 4 condition</td>
</tr>
<tr>
<td>14</td>
<td>Input_2 (resulting from Transition 6 action)</td>
<td>Transition 4 condition</td>
</tr>
<tr>
<td>15</td>
<td>Input_1 (as program input)</td>
<td>Transition 1 condition (1st occurrence)</td>
</tr>
<tr>
<td>16</td>
<td>Input_1 (resulting from Transition 6 action)</td>
<td>Transition 1 condition (1st occurrence)</td>
</tr>
<tr>
<td>17</td>
<td>Input_2 (as program input)</td>
<td>Transition 1 condition (1st occurrence)</td>
</tr>
<tr>
<td>18</td>
<td>Input_2 (resulting from Transition 6 action)</td>
<td>Transition 1 condition (1st occurrence)</td>
</tr>
<tr>
<td>19</td>
<td>Operator (as program input)</td>
<td>Transition 1 condition (1st occurrence)</td>
</tr>
<tr>
<td>20</td>
<td>Operator (resulting from Transition 6 action)</td>
<td>Transition 1 condition (1st occurrence)</td>
</tr>
<tr>
<td>21</td>
<td>Operator (as program input)</td>
<td>Transition 1 condition (2nd occurrence)</td>
</tr>
<tr>
<td>22</td>
<td>Operator (resulting from Transition 6 action)</td>
<td>Transition 1 condition (2nd occurrence)</td>
</tr>
<tr>
<td>23</td>
<td>Input_2 (as program input)</td>
<td>Transition 1 condition (2nd occurrence)</td>
</tr>
<tr>
<td>24</td>
<td>Input_2 (resulting from Transition 6 action)</td>
<td>Transition 1 condition (2nd occurrence)</td>
</tr>
<tr>
<td>25</td>
<td>Clear (as program input)</td>
<td>Transition 6 condition (1st occurrence)</td>
</tr>
<tr>
<td>26</td>
<td>Clear (as program input)</td>
<td>Transition 6 condition (2nd occurrence)</td>
</tr>
<tr>
<td>27</td>
<td>Clear (as program input)</td>
<td>Transition 7 condition</td>
</tr>
</tbody>
</table>

**TABLE 3.2:** Definition-use associations in the Calc example.
Transition 2:
Uncovered DU’s: 7, 8;
DU Picked: 7.
1st try: input1 = 3, input2 = NAN, operator = *, can’t cover.
2nd try: input1 = NAN, input2 = NAN, operator = *, T covered.
Oracle validates.
DU’s covered so far: 7.
Infeasible DU’s tried to cover: none.

Transition 2:
Uncovered DU’s: 8; DU Picked: 8.
1st try: input1 = null, input2 = null, operator = null. T covered.
Oracle validates.
DU’s covered so far: 7, 8.
Infeasible DU’s tried to cover: none.

Transition 1:
Uncovered DU’s: 1, 2, 3, 4, 5, 6, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24; DU Picked: 15.
1st try: input1 = NAN, input2 = 4, operator = +, can’t cover.
2nd try: input1 = 3, input2 = NAN, operator = *, can’t cover.
3rd try: input1 = 3, input2 = NAN, operator = /, can’t cover.
4th try: input1 = 3, input2 = 4, operator = +, T covered.
Oracle validates.
DU’s covered so far: 1, 3, 5, 7, 8, 15, 19, 21, 23.
Infeasible DU’s tried to cover: none.

Transition 1:
Uncovered DU’s: 2, 4, 6, 16, 18, 20, 22, 24; DU Picked: 16.
1st try: input1 = null, input2 = null, operator = null, can’t cover. (cannot manipulate constants).
DU’s covered so far: 1, 3, 5, 7, 8, 15, 17, 19, 21, 23.
Infeasible DU’s tried to cover: 2, 4, 6, 16, 18, 20, 22, 24
...(and so on)...

TABLE 3.3: The beginning of one possible run of a WYSIWYT-based test generation.
3.2 Issues Introduced by WYSIWYT for Screen Transition Diagrams

Two properties must be satisfied to apply WYSIWYT to the screen transition paradigm. First, spreadsheets are deterministic. In part, this is accomplished by nesting “if expressions”. One way to ensure that this property holds in the screen transition diagram is to specify the order in which transitions’ predicates are checked. (In fact, this is common in visual rule-based languages.) Second, in the spreadsheet paradigm, there is a distinguished well-defined value “undefined” that is assigned to cells in which no predicate is satisfied and no “else” clause is reachable. We introduce a similar mechanism in the screen transition paradigm for compatibility. Transition 4 in Table 2.1 is an example of a transition that would be built-in in some fashion for all output objects as a fall-through to ensure that all values are well defined. We refer to such transitions as fall-through transitions.
CHAPTER 4

TRANSLATING SCREEN TRANSITION DIAGRAMS TO CRGS

To precisely define WYSIWYT in the screen transition paradigm, a formal model of a screen transition diagram is needed. WYSIWYT for spreadsheets already has a formal model, the CRG. Can the same formal model be used to reason about WYSIWYT in screen transition diagrams? This chapter presents a translation method that starts with an arbitrary screen transition diagram and produces a CRG; Chapter 5 then considers its equivalence from a testing perspective to the original screen transition diagram. The diagram in Figure 2.4 and the associated Table 2.1 will serve as a running example.

4.1 The Translation Method

The translation method (1) translates the screen transition diagram to a set of rules, (2) translates the rules to a spreadsheet, and (3) translates the spreadsheet to its formal model, the CRG.

The first step of the translation method exploits the similarity between the rule-based paradigm and the screen transition paradigm. The outcome of this step is a set of rules, and the particular form of rules we choose is FAR rules. FAR (Formulas and Rules) is an end-user VPL that supports programming in both the spreadsheet and rule-based paradigms by representing the same code in both paradigms [9]. FAR rules are used because FAR’s translation model
provides the vehicle needed for translating FAR rules to a spreadsheet and hence to WYSIWYT’s formal model, the CRG. Figure 4.1 presents an algorithm to translate a screen transition diagram to an ordered collection of FAR rules. Its result on our running example is shown in Figure 4.2.

1. Define a FAR rule of the form \((C, \text{predicate, consequence expression})\) for each transition action preserving the order per assignment action. \(C\) is the LHS of the assignment action, \(\text{predicate}\) is the predicate of the transition and \(\text{consequence expression}\) is the RHS of the transition action.

2. Define a FAR rule of the form \((C, \text{predicate, consequence expression})\) for each input variable, where \(C\) is the name of the variable, \(\text{predicate}\) is “always” and \(\text{consequence expression}\) is the constant that has previously been input.

FIGURE 4.1: Translating a screen transition diagram to FAR rules.

The derived FAR rules can now be translated to a spreadsheet via the algorithm shown in Figure 4.3. This translation algorithm is based on the translation model presented in [9], but adds support for “else”.

Applying the algorithm in Figure 4.3 to the example’s results produces a set of cells with size equal to the number of unique \(C\)’s in the FAR rules created by the algorithm shown in Figure 4.1. Figure 4.4 sketches the resulting spreadsheet.

The algorithm for translating a spreadsheet to a CRG is given in [32]. The input cells’ resulting formula graphs have three nodes: an entrance node, a constant node, and an exit node. The remaining cells’ resulting formula graphs have control flow between the entrance and exit nodes. See Figure 4.5.
An ordered set of transition action rules (one per action in Table 2.1):

- (Asgn, (EC == 0 AND !(Error? Asgn0) AND !(Error? Asgn1)), Asgn0 + Asgn1)
- (Asgn, (EC == 1 AND !(Error? Asgn0) AND !(Error? Asgn1)), Asgn0 + Asgn1)
- (Asgn, ((Error? Asgn0) OR (Error? Asgn1)), Error)
- (Asgn, Else, Undefined)
- (Total, (EC == 0, !(Error? Asgn0) AND !(Error? Asgn1)), Asgn)
- (Total, (EC == 1, !(Error? Asgn0) AND !(Error? Asgn1)), Asgn + AsgnEC)
- (Total, ((Error? Asgn0) OR (Error? Asgn1)), Error)
- (Total, Else, Undefined)

Input rules (one for each input in Figure 2.4):

- (EC, Always, 1)
- (Asgn0, Always, 56)
- (Asgn1, Always, 20)
- (AsgnEC, Always, 12)

FIGURE 4.2: FAR rules derived by applying the algorithm in Figure 4.1 to the screen transition diagram of Figure 2.4 and Table 2.1.

1. For each FAR rule of the form \((C, \text{"always"}, \text{consequence expression})\), translate to cell \(C\) with formula "consequence expression".

2. For each collection of two or more FAR rules containing the same LHS \(C\):
   
   (a) Create a cell \(C\) with formula empty string.

   (b) Preserving the order of Figure 4.1’s algorithm:

   i. For each FAR rule of the form \((C, \text{predicate}, \text{consequence expression})\) except for the fall-through, append to \(C\)'s formula "if \text{predicate} then \text{consequence expression} else".

   ii. For the fall-through FAR rule of the form \((C, \text{"else"}, \text{consequence expression})\), append to \(C\)'s formula "consequence expression".

FIGURE 4.3: Translating FAR rules to a spreadsheet.
FIGURE 4.4: Sketch of Grades translated to a spreadsheet via FAR rules. Cells show values, each cell's name is above the cell, and each formula is depicted at the cell's lower right.
FIGURE 4.5: Grades CRG. Nodes 20 and 29 have been introduced because all arrows to exit nodes must begin at computation nodes to ensure a well-formed spreadsheet program. Predicate nodes with undefined branches must define the cell with a well-defined value as "undefined".
4.2 Comparing Du-Associations in CRGs and Screen Transition Diagrams

The reason for translating an arbitrary screen transition diagram to a cell relation graph is to show that testing the cell relation graph accomplishes testing the screen transition diagram. Thus, it must be shown that the translation algorithms produce a CRG that is test equivalent to the original screen transition diagram. In other words, it must be shown that fully testing in either paradigm covers the same situations. Situations are represented by definition-use associations within both paradigms. We discuss specific discrepancies in du-associations between our example screen transition diagram and CRG here. We explore coverage equivalence generally in Chapter 5.

The methodology presented in [32] is used to find all of the du-associations that are associated with the sample program from the CRG. The following examples start with the sample program that has been the running example in this work. Figure 2.4 and Table 2.1 showed this program in the screen transition paradigm, and Figure 4.5 shows a CRG representing the same program. Examples 1 and 2 detail the definitions and du-associations corresponding to the CRG. Examples 3 and 4 then detail the definitions and du-associations in the screen transition version. These examples provide a concrete vehicle for comparison.

Example 1: Definitions in the CRG

Definitions in a CRG are all the computation nodes connected to an exit node. Thus, all of the definitions in the CRG represented in Figure 4.5 are:

1. 2, EC
2. 5, Asgn0
3. 8, Asgn1
4. 11, AsgnEC
5. 15, Asgn
6. 17, Asgn
7. 19, Asgn
8. 20, Asgn
Example 2: Du-associations in the CRG

The du-associations from the CRG represented in Figure 4.5 are:

1. 2,14t,EC
2. 5,14t,Asgn0
3. 8,14t,Asgn1
4. 2,16t,EC
5. 5,16t,Asgn0
6. 8,16t,Asgn1
7. 5,18t,Asgn0
8. 8,18t,Asgn1
9. 2,14f,EC
10. 5,14f,Asgn0
11. 8,14f,Asgn1
12. 2,16f,EC
13. 5,16f,Asgn0
14. 8,16f,Asgn1
15. 5,18f,Asgn0
16. 8,18f,Asgn1
17. 2,23t,EC
18. 5,23t,Asgn0
19. 8,23t,Asgn1
20. 2,25t,EC
21. 5,25t,Asgn0
22. 8,25t,Asgn1
23. 5,27t,Asgn0
24. 8,27t,Asgn1

Example 3: Definitions in the screen transition diagram

In Table 2.1, definitions all occur in the action portion of a transition, or in editing cells. Note the one-to-one correspondence of these definitions with those of Example 1, as definitions are explicitly preserved in the translation process.

Thus, all of the definitions in Figure 2.4 and Table 2.1 are:
Example 4: Du-associations in the screen transition diagram

Now all definition-use associations are defined using the transitions and uses:

1. Edit EC, 1 Predicate
2. Edit Asgn0, 1 Predicate
3. Edit Asgn1, 1 Predicate
4. Edit EC, 2 Predicate
5. Edit Asgn0, 2 Predicate
6. Edit Asgn1, 2 Predicate
7. Edit Asgn0, 3 Predicate
8. Edit Asgn1, 3 Predicate
9. Edit Asgn0, 1 Action Asgn
10. Edit Asgn1, 1 Action Asgn
11. Edit Asgn0, 2 Action Asgn
12. Edit Asgn1, 2 Action Asgn
13. 1 Action Asgn, 1 Action Total
14. 2 Action Asgn, 1 Action Total
15. 3 Action Asgn, 1 Action Total
16. 4 Action Asgn, 1 Action Total
17. 1 Action Asgn, 2 Action Total
18. 2 Action Asgn, 2 Action Total
19. 3 Action Asgn, 2 Action Total
20. 4 Action Asgn, 2 Action Total
21. Edit AsgnEC, 6 Action Total

The set of du-associations in a screen transition diagram was defined in Chapter 3. The methodology presented in [32] defines the set of du-associations
in a CRG. A surprising outcome of the translation is that these sets for a screen transition diagram and its CRG produced by the translation do not match! There can be more du-associations in the new CRG than in the original diagram.

Definitions are mapped 1:1 from screen transition diagrams to CRGs, as are computational du-associations. The difference is in the predicate du-associations. In our running example, whereas there are 45 du-associations in the CRG, there are only 21 in the screen transition diagram. There are 8 predicate uses (p-uses) in the screen transition diagram and 32 p-uses in the CRG. There are two reasons for this difference. First, the CRG repeats predicates for each cell affected. For example, each transition in Table 2.1 affects two cells. Thus, the predicate that appears only once in the screen transition diagram is repeated for both Asgn and Total in the CRG (as pointed out in the caption of Figure 4.5). Second, screen transition diagrams have only true du-associations in their conditions, whereas both true and false du-associations are explicit in CRGs, as Figure 4.5 shows. Thus, having p-uses on both true and false branches as in the CRGs is irrelevant in screen transition diagrams. This shows why there are four times as many p-uses in the CRG as there are in the screen transition diagram.

This difference leaves us with two choices. The first is to directly define a new formal model (CRG-like or not) for screen transition diagrams that exactly matches the diagram's du-associations, but this would remove the explicit tie between the screen transition paradigm and WYSIWYT's CRG. This explicit tie is important, both because it establishes WYSIWYT's generalizability, and because it allows previous findings about WYSIWYT to transfer directly to the screen transition paradigm. Thus, we take the other choice: to consider whether the smaller set of du-associations in the screen transition diagram actually provides as much coverage as the larger set in the CRG.
CHAPTER 5

COVERAGE EQUIVALENCE OF TRANSLATED PROGRAMS

To study the reduction of uses described in the preceding chapter further, we draw from research on “subsuming sets” [25, 26]. For example, if covering one du-association implies covering others, it may not be necessary to explicitly consider the du-associations that are coverable by the other du-associations, maybe requiring less input from the end user. Here we define subsuming sets.

Definition 5.0.1 Let S be a set of du-associations in program P. Let S' be a subset of S. We say S' is a subsuming set of S and that S' subsumes S if a set of tests that is adequate for S' is also adequate for S.

We now show that the set of du-associations in the screen transition paradigm, when translated to the spreadsheet paradigm, constitutes a subsuming set in the spreadsheet as defined by the spreadsheet’s CRG. This shows that the CRG, which is the formal model defining WYSIWYT for spreadsheets, generalizes to also define WYSIWYT for the screen transition paradigm. This is the final step in showing the feasibility of applying the WYSIWYT methodology to the screen transition paradigm.

The translation algorithms listed allow for an increase in du-associations when translating from the screen transition paradigm to the spreadsheet paradigm. Thus it must be shown that the set of du-associations in the screen transition paradigm subsumes the set of du-associations in the spreadsheet paradigm.
in order to equate all du-associations test coverage across both the spreadsheet and the screen transition paradigms.

We begin by considering the spreadsheet, as modeled by its CRG, resulting from the translation algorithms shown in Chapter 4. Section 5.1 shows the construction of a subsuming set and proves that it subsumes the entire spreadsheet. We then show in Section 5.2 that the du-associations in the screen transition diagram are a subsuming set for the translated spreadsheet.

5.1 Subsumption within the spreadsheet paradigm

We start by ignoring conditionals in Section 5.1.1, then we include them in Section 5.1.2.

5.1.1 Coverage Equivalence without conditionals

We begin by considering basic constructs and properties in the spreadsheet paradigm. Without control-altering constructs (conditionals are the only control altering constructs in spreadsheets), spreadsheets are composed of combinations of lines, forks, and sinks as shown in Figure 5.1.

**Definition 5.1.1** A line is an ordered sequence of two or more cells. Let program (spreadsheet) $P$ be a line. Each cell $P_i$ in $P$ references only cell $P_{i-1}$. Cell $P_0$ references nothing.

**Definition 5.1.2** A fork consists of a program, two lines, and an intermediate connecting cell. Let $P$ be a program with a sink cell $P_n$. Let $Q$ and $R$ each be line programs consisting of $m$ and $o$ cells, respectively. Let $A$ be a single cell that references $P_n$. $A$ is the connecting cell and both $Q_0$ and $R_0$ reference $A$. 
Definition 5.1.3 A join consists of two programs, a line, and an intermediate connecting cell. Let \( R \) be a line program with \( n \) cells. Let \( P \) and \( Q \) each be programs including sinks \( P_n \) and \( Q_m \) respectively. Let \( A \) be a single cell that references both \( P_n \) and \( Q_m \). \( A \) is the connecting cell and \( R_0 \) references \( A \).

Lemma 5.1.1 For two cells \( C_0 \) and \( C_1 \) without conditionals, such that \( C_1 \) references \( C_0 \), a subsuming set of \( C_1 \) also subsumes \( C_0 \).

Proof. Exercising the du-associations whose uses are in \( C_1 \) guarantees the exercising of the du-associations whose uses are in \( C_0 \). This follows directly from the definition of what it is to exercise a du-association in the context of WYSIWYT [32]. Thus any set of tests that are adequate for \( C_1 \) are also adequate for \( C_0 \). By the definition of subsumption, \( C_0 \) is subsumed by any subsuming set for \( C_1 \).

Theorem 5.1.1 For line programs, the subsuming set is the du-association whose use is found in the sink cell.

Proof. The proof proceeds by induction.

Base Cases. The one-cell case is trivial, as the set of du-associations is null. Consider a spreadsheet consisting of line \( M \) composed of two cells, \( M_1 \) and \( M_2 \). \( M_2 \) is the sink. \( M_2 \) contains a reference to \( M_1 \). Exercising the du-associations contained in \( M_2 \) guarantees the exercising of du-associations in \( M_1 \) by Lemma 5.1.1.

Inductive Hypothesis. Consider a program \( P \) with only line \( N \) composed of \( n \) cells, \( n > 2 \). Cell \( N_n \) is the sink. The subsuming set consists of the du-association whose use is contained in cell \( N_n \).
FIGURE 5.1: Examples of a line, fork and join in the spreadsheet paradigm. Each cell is named with index notation (e.g. P[0]). Below the name is a box containing the cell’s value, and below the value is a box containing the cell’s formula. Arrows represent dataflow dependencies between cells.
Inductive Step. Consider a program $Q$ with only line $O$ composed of $n+1$ cells, $n > 2$. Cell $O_{n+1}$ is the sink. If the cell $O_{n+1}$ is removed, by the inductive hypothesis, $O_n$ contains the subsuming set for the entire spreadsheet. Since $O_{n+1}$ references $O_n$, exercising $O_{n+1}$ guarantees exercising the subsuming set defined for $O_n$ by Lemma 5.1.1. So a subsuming set for $Q$ consists of the du-association whose use is in cell $O_{n+1}$.

Theorem 5.1.2 For programs without conditionals, the subsuming set is the set of all du-associations whose uses are found in sink cells.

Proof. Theorem 5.1.1 showed this to be true for line programs. Without conditionals there are only two other program structures that can occur in a program: a fork and a join. It is helpful to note that in a spreadsheet without control flow altering constructs, reaching a cell without error guarantees that all cells which it references directly or indirectly must be executed. Thus, if all du-associations in sinks are included in the spanning set $S$, and they are feasible, then $S$ does indeed subsume all du-associations in the spreadsheet. All that is left to show is that in spreadsheets without conditionals some sink is reachable from each cell. A spreadsheet without conditionals can be considered a directed acyclic graph (DAG) $(V,E)$, where $V$ is the set of cells, and $E$ represents cell references (edges are directed away from references). Now the proof becomes: In a connected, directed acyclic graph, some sink node is reachable from each node in the graph. The proof proceeds by induction on the depth of the graph.

Base Cases: Depth = 0. The zero-depth case is trivial. Let $G(V,E)$ be a connected DAG with depth 0. $V$ consists of a single node $N$ and $E$ is an empty set. As $N$ is the only node in the graph and $N$ is also a sink node, a sink node is reachable from all nodes in the graph.
**Base Cases:** Depth = 1. Let $G(V,E)$ be a connected DAG with depth 1. $V$ can be separated into two sets: let $M$ be the set of source nodes and $N$ be the set of sink nodes. Obviously, as set $N$ is the set of sink nodes, sink nodes are reachable from every member in set $N$. As the graph has depth 1, edges from a node in $M$ must end in $N$.

**Inductive Hypothesis.** Let $G(V,E)$ be a connected DAG with depth $n$. Let $N$ be the set of sink nodes in $G$. Some sink node in $G$ is reachable from every node in $G$.

**Inductive Step.** Let $G'(V \cup N', E \cup A')$ be a connected DAG with depth $n+1$. (Note that $N$ from the inductive hypothesis is a subset of $V$.) Let $N'$ be a set of sink nodes, each connected to a node in $N$ of graph $G$ with an edge in $A'$. In removing $N'$ and $A'$ from the graph, the inductive hypothesis can be used to show that every node in $G$ can reach some sink in $G$. Now add back $N'$ and $A'$. As every edge in $A'$ is connected from some node in $N$ to some node in $N'$, all nodes in $N'$ are reachable from $N$. As all nodes in $G$ can reach $N$, and all nodes in $N$ which are not sinks can reach $N'$, all nodes in $G'$ can reach some sink.

Thus, a subsuming set of a spreadsheet without control flow altering constructs is the set of du-associations whose uses are contained in the sinks of a program.

### 5.1.2 Coverage Equivalence with Conditionals

Now we show the construction of a subsuming set for programs with conditionals. For precision, we define the du-association types that conditionals introduce. Figure 5.2 illustrates these types of du-associations.
Definition 5.1.4 A CDU-out is a computation-use du-association whose definition is contained within a conditional cell.

Definition 5.1.5 A CDU-in is a computation-use du-association whose use is contained within a conditional cell.

Definition 5.1.6 A PDU-in is a predicate-use du-association whose use is contained within a conditional cell.

Conditionals are the only control-flow-altering constructs in the spreadsheet paradigm. If we can show that the set created by adding CDUs-out to the subsuming set subsumes all du-associations introduced with conditionals, then we can show that the set created by unioning the set of CDUs-out with the subsuming set of the spreadsheet's portions without conditionals will subsume the spreadsheet.

Theorem 5.1.3 The subsuming set for spreadsheets with zero or more conditionals is the set of du-associations whose uses are in sink cells and the set of du-associations whose uses reference conditional cells.

Proof. Proof proceeds by construction (four cases):

Du-associations not participating in conditionals: As shown earlier, our subsuming set first consists of the du-associations with uses in the sinks. As shown in Theorem 5.1.2, coverage is still guaranteed for all du-associations not involving conditionals.

CDUs-out: We add all CDUs-out to the subsuming set. That is, we add all du-associations in which the definition is a computation use in a conditional cell.
FIGURE 5.2: Definition-use association types: This is a modified CRG to highlight the definition-use types added by conditionals. We began with a cell relation graph, removing the cell dependency arrows. We then removed the entry and exit nodes. Finally, we added explicit depictions of definition-uses with dashed arrows.
PDU\textsuperscript{sin}: No action is required, because of the requirement for well-defined cells in Section 3.2. By this requirement, all predicate branches are followed by CDUs-out. By including all CDUs-out in the subsuming set, both true and false PDU\textsuperscript{sin} will be reached as well.

CDU\textsuperscript{sin}: Again, no action is required. All CDU\textsuperscript{sin} for a cell either end in sinks or are connected to the use cell’s CDU\textsuperscript{out}. This guarantees exercising the CDU\textsuperscript{in} by the definition of what it is to exercise a du-association in the context of WYSIWYG\cite{32}. This is also because of the requirement for well defined cells in Section 3.2. Since all predicate branches must lead to computation nodes, and all references to CDU\textsuperscript{out} computation nodes contained in conditional cells are included in the subsuming set, the du-associations in which the computation use resides in conditional cells are guaranteed to be subsumed as well.

5.2 Subsumption in the Screen Transition Paradigm

We have shown that there exists a subsuming set construction for the spreadsheet paradigm. Now we show that given our translation algorithms in Chapter 4, the du-associations in the screen transition paradigm are a superset of the subsuming set in the spreadsheet paradigm. Thus, all-du-association testing in the screen transition paradigm guarantees all-du-association testing in the spreadsheet produced by the translation, as modeled by the CRG. We sketch the proof here.

Theorem 5.2.1 The du-associations in the screen transition paradigm are a superset of the subsuming set in the spreadsheet paradigm.

Proof. There are three cases. The first case is trivial in that CDUs are translated directly according to the algorithms in Section 4. The second case deals
with PDUs that are directly translated and again this is trivial. The final case shows that the PDUs that are created are not members of the subsuming set in the spreadsheet.

**Case 1: Computation-use du-associations:** By Theorem 5.1.3, a subsuming set $S$ for a spreadsheet consists of the conditional cells’ CDUs-out and all du-associations in sink cells. Obviously, all computation-use du-associations in $S$ are also in the screen transition diagram, since the algorithms in Chapter 4 map all computation-use du-associations 1-to-1.

**Case 2: Non-sink predicate-use du-associations:** The translation algorithms preserve all predicate-use du-associations, and hence any of those in $S$ are also in the screen transition diagram. The translation algorithms can also add new predicate-use du-associations to the spreadsheet, which can either be in non-sink conditional cells or in sink conditional cells. Let $C$ be the set of all CDU-outs from a non-sink conditional cell. The set $C$ together subsumes all predicate du-associations in the cell containing $C$’s definitions, by Theorem 5.1.3, and we just showed that these CDU-outs are also in the screen transition diagram.

**Case 3: Sink predicate-use du-associations:** Recall that one kind of these additions came from duplicating du-associations in the translation algorithms, but since $S$ is a set, such duplicates cannot affect $S$’s contents. The other kind of addition came from adding a false predicate-use to the spreadsheet for each (true) predicate in the screen transition paradigm. If this addition is in a sink, it will be in $S$. However, as discussed in Chapter 3, the final else in a cell always exists in the screen transition diagram, and since it subsumes all previous else’s
in the cell, the previous else's, whether additions or not, are not needed in $S$.

We have shown that the set of du-associations in the screen transition diagram is a subsuming set in the translated spreadsheet. Thus, all du-associations testing in the screen transition paradigm guarantees all du-associations testing in the corresponding spreadsheet.
CHAPTER 6
CONCLUSIONS AND FUTURE WORK

In this work we considered the question of whether the WYSIWYT visual testing methodology was general enough to serve as a testing methodology for the screen transition paradigm. We showed that it is by showing that there is a visual adaptation of WYSIWYT for a (hypothetical) screen transition language, and by presenting a translation from an arbitrary screen transition diagram to a coverage-equivalent CRG.

Thus, the CRG formalism defining WYSIWYT for spreadsheets also generalizes to define WYSIWYT for screen transition diagrams. This result has two practical implications. First, the screen transition paradigm is emerging for several purposes, including for teaching (as in the Kara system), for interface design by end users (as in the SILK system), and for real-world communication about software needs (as in the Lyee methodology); the ability to use WYSIWYT in this paradigm immediately provides support for testing programs in this paradigm, even when the testers are end-user programmers without formal software engineering training. Second, the paradigm's strong relationship to rule-based programming and to more traditional state transition diagrams suggests that WYSIWYT may be general enough to support these paradigms as well.

The next target of our research will be to study further how to set up visualization of WYSIWYT in the screen transition paradigm. How should the user
interface with the testing methodology? Both ease of use and clarity for user decisions are to be kept in mind. Also, how can the portions of the program that are relevant be shown to allow for the best understanding by the end user? Dealing with a finite screen size has often been a problem in testing the spreadsheet paradigm, and we expect that this problem will be similar in the screen transition paradigm. These questions and many more must be asked to see if we can devise a way to effectively communicate with the user about our testing methodology's reasoning. From literature on on-line trust and its impact on the usefulness of on-line systems, it is clear that users need to understand the system's logic to trust it, and that if they do not trust it, they will not bother to provide the system with the information (validations) needed by the system to help users test [2, 14]. The most important principle underlying this research is that the purpose of a testing methodology is to support end users' ability to problem solve about the correctness of their requirements specifications; any design choices that we make must above all uphold this principle, even if it means sacrificing other desirable attributes. After all, if end users do not find it to be helpful, they will not use it or gain from it. Evaluating our approach will eventually require empirically testing a fully functional implementation of our testing methodology with end-user programmers of various experience levels.

Finally, we hope to look into properties of the set of du-associations in the screen transition paradigm in order to answer some important questions about testing in this paradigm. For example we are interested in finding sets of du-associations in programs for which the maximum ratio possible of these du-associations furthers testedness upon validation. This is another way of describing minimum subsumption sets [26]. We are interested to see whether this would keep the user interested in testing by always making progress upon val-
idation or if this would make programs too difficult to test in that validations would be on compound du-associations. Minimum subsumption sets certainly have the danger of convoluting testing by forcing the user to validate a large portion of the program with one decision. Future research will deal with discovering a balance between these two extremes.

In a nutshell, we now have a formal model to support testing in the screen transition paradigm. We have designed one possible approach to how an end user would interface with our testing methodology. Designing, implementing and empirically testing an interface which utilizes this model is the next logical step. We hope that this paradigm and testing methodology will empower end users, thus bridging the gap between the innovations of end users and the software engineering expertise of professional developers.
BIBLIOGRAPHY


