End-User Testing for the Lyee Methodology Using the Screen Transition Paradigm and WYSIWYT

Darren BROWN, Margaret BURNETT, and Gregg ROTHERMEL

TR#03-60-01

School of Electrical Engineering and Computer Science

Oregon State University, Corvallis, OR 97331

20 March 2003

burnett@cs.orst.edu

Abstract

End-user specification of Lyee programs is one goal envisioned by the Lyee methodology. But with any software development effort comes the possibility of faults. Thus, providing end users a means to enter their own specifications is not enough; they must also be provided with the means to find faults in their specifications, in a manner that is appropriate not only for the end user's programming environment but also for his or her background. In this paper, we present an approach to solving this problem that marries two proven technologies for end users. One methodology for enabling end users to program is the screen transition paradigm. One useful visual testing methodology is "What you see is what you test (WYSIWYT)". In this paper, we show that WYSIWYT test adequacy criteria can be used with the screen transition paradigm, and present a systematic translation from this paradigm to the formal model underlying WYSIWYT.

Key words: WYSIWYT, screen transition diagram, end-user software testing

PACS:

1 Introduction

One goal envisioned by the inventors of the Lyee methodology [1] is that even an end user will someday be able to enter a set of software requirements—without the assistance of a professional software developer—and that the methodology will be able to generate software that conforms to these requirements. (Following convention in the literature, the term "end users" as used here refers to people who are not professional programmers.) Indeed, other collaborators in the Lyee project have been working on important fundamentals that will contribute to this goal (e.g., [2], [3]). For this goal to become a reality, 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 [4]. These partial results allow optimism that the questions above can be resolved. With this in mind, we will make the further assumption that these and related issues can be solved for the Lyee methodology. But once 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 Lyee environment?

Addressing this question is our role in the Lyee collaboration project. This paper makes the following specific contributions:

(1) We present a structure of Lyee requirements with two important attributes: it is based on an approach of proven usefulness to end users, and it is highly compatible with the Lyee methodology's underlying structures.
(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.

In Section 2 we present a summary of the WYSIWYT methodology which has previously been devised for the spreadsheet paradigm. Section 3 presents the structure for end users of item (1) above. Section 4 brings up some specific

issues that must be resolved to ensure the viability of testing the structure. Section 5 discusses the choice of adequacy criteria (item 2 above) and presents the translation from the structure to the WYSIWYT methodology's formal model (item 3). Section 6 concludes.

2 Background: The WYSIWYT Testing Methodology

In previous work [5], [6], [7], 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 programmers test more effectively and efficiently using WYSIWYT than they do unaided by WYSIWYT (e.g., [8], [9], [5], [10]).

This proven effectiveness of WYSIWYT for spreadsheets suggests WYSIWYT as a possibility for testing in the Lyee methodology, provided that it can be effectively adapted to Lyee. In this section, we summarize the WYSIWYT methodology.

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 [11], [12]. 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) to tested (blue).

Figure 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 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 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 is aware of 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) [6]. 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 [13], [14] for discussions of how complex spreadsheets are treated.) For example, Figure 2 shows a portion of the CRG for the cells in Figure 1, delimited by dotted rectangles. The process of translating an abstract syntax tree representation of an expression into its control flow graph representation is well known [15]; a similar translation applied to the abstract syntax tree for each formula in a spreadsheet yields that formula's formula graph. 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 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.

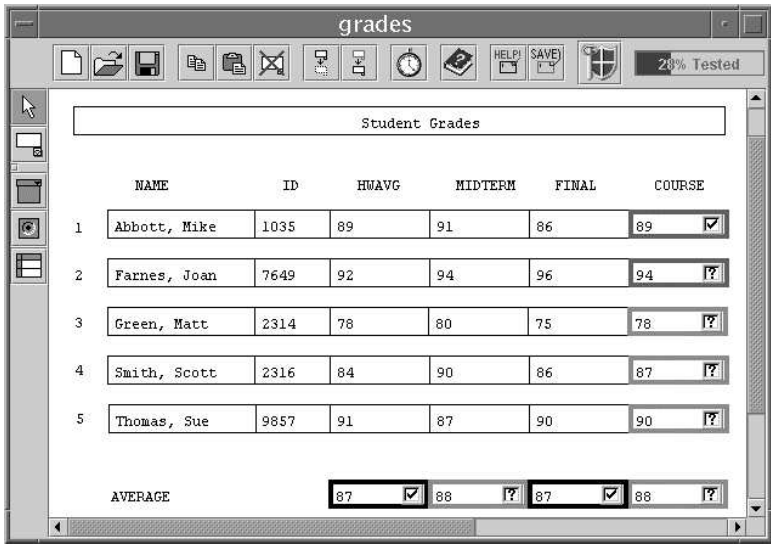

Fig. 1. Visual depiction of testedness of a student grades spreadsheet. Blue-bordered cells (black in this paper) such as the first cell in the Average row are tested, red-bordered cells (light gray in this paper) such as the second cell in the Average row are untested, and shades between such as the top cell in the Course column are partially tested. The upper right corner of each spreadsheet reports a spreadsheet's overall testedness percentage. 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.

We used the cell relation graph model to define several test adequacy criteria for spreadsheets [6]. (A test adequacy criterion [16] is a definition of what it means for a program to be tested "enough.") The strongest criterion we defined, du-adequacy, is the criterion we have chosen for our work. The du-adequacy criterion is a type of dataflow adequacy criterion [17], [18], [19], [20]. 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 de-

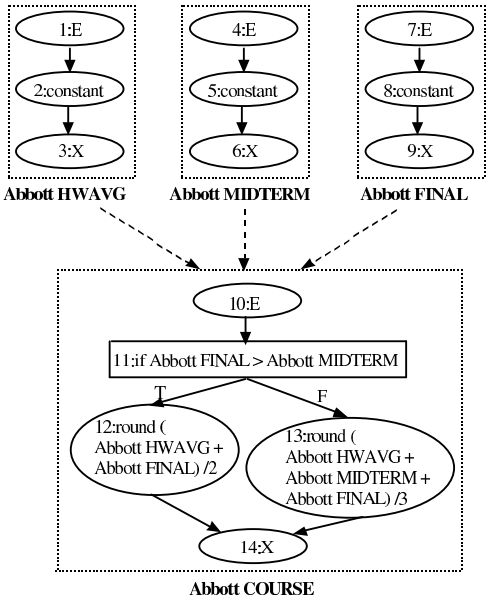

Fig. 2. A partial cell relation graph of Figure 1. These are the formula graphs for the top row ("Abbott, Mike"). Dashed arrows indicate dataflow edges between cells' formula nodes. For clarity in this figure, we preface cell names with "Abbott" instead of with the internal IDs actually used.

fines 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 out-edge from a predicate node that refers to C). For example, in Figure 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 definition-use associations have been covered (executed) by at least one test. In this model, a test is a user decision as to whether 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 significantly more likely to achieve du-adequate coverage and do so efficiently using the WYSIWYT methodology than those not using it [9], [10], du-adequate test suites have frequently been significantly more effective at fault detection than random test suites [5], and subjects have been significantly more likely to correctly eliminate faults using the WYSIWYT methodology than those not using it [8]. Both programmer [8], [10] and end-user [9] audiences have been studied.

3 Attributes of End-User Requirements Specification

To define an end-user methodology for testing Lyee specifications, basic attributes of the user's paradigm must be established for working with Lyee. Our strategy was to consider end-user programming specification paradigms that already exist, that also have strong compatibility with the Lyee methodology. Our literature search revealed a paradigm that has been proven empirically to be successful with one class of end users (namely, interface designers) [21], [22]. This paradigm is called the "screen transition paradigm". The general idea is

that a user can design an interface by explicitly sketching how the intended interface is to be used. See Figure 3.

The screen transition paradigm is an extremely good fit with the Lyee methodology, because much of it is already a part of the Lyee methodology. That is, the Lyee methodology begins with similar diagrams that are currently manually simulated by developers. The Lyee objects such as words, formulas, and conditions are extracted from these diagrams. Thus, this is a natural fit to connect the users with the underlying Lyee structures. In this paper, we will assume that this idea can be applied to end-user programming for the Lyee methodology.

We are also 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; a later collaboration may design exactly how this will work.)

The question that we consider in this paper is this: Given the assumption that end users will provide complete specifications via screen transition diagrams, how can testing be supported? The first issue to consider is how testing support will be presented to users. We do not develop this issue here, but we do require that its solution must satisfy two constraints: (1) the presentation of testedness

will be integrated with the screen transition diagrams, as in the WYSIWYT methodology, and (2) any update or test made by the user will be immediately and visually reflected in the presentation.

The second issue to consider is, what is the user going to test? In order to consider this issue, it is necessary to define further conditions for the user's requirements specification elements.

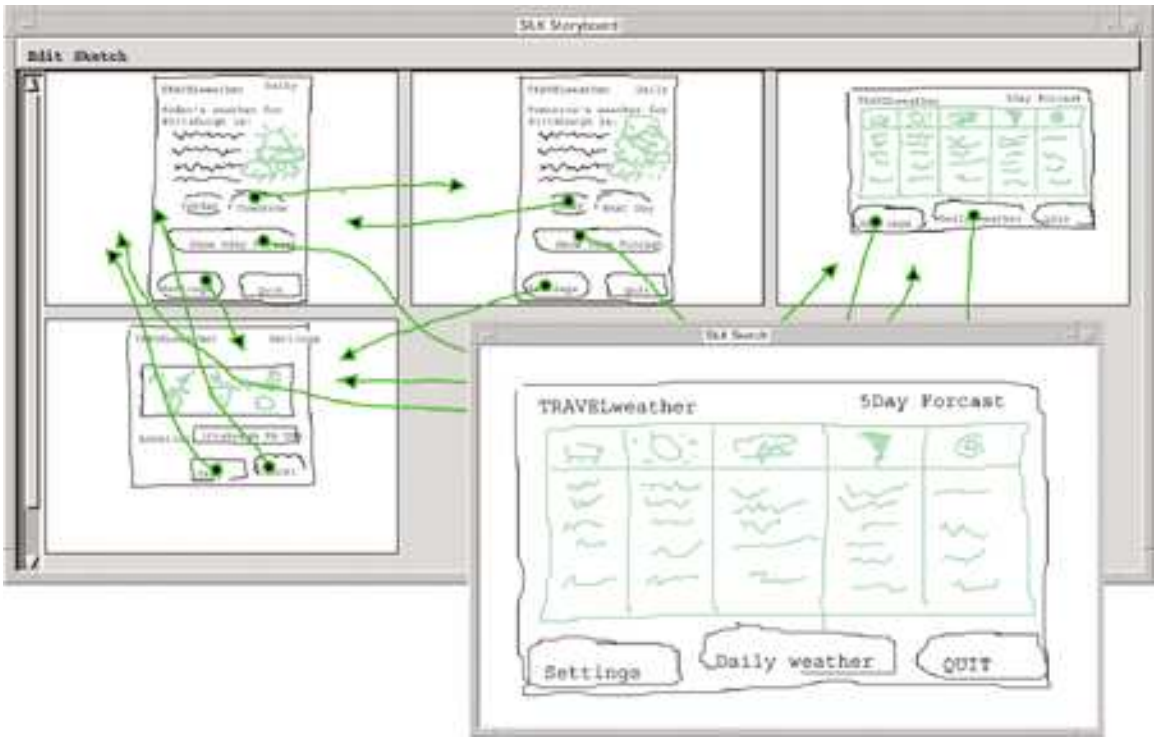

Fig. 3. A SILK sketch (front) of a five-day weather forecast and storyboard (rear) [22]. An experienced user-interface designer created the sketch. The designer has also created buttons and drawn arrows to show the screen transitions that occur when the user presses the buttons.

With screen transition diagrams, the user communicates a program using screens, objects on those screens, and transition-actions. (It should be understood that we are attempting to relate only what the user communicates, not how.) Thus, a screen is a collection of objects, which will be used and/or

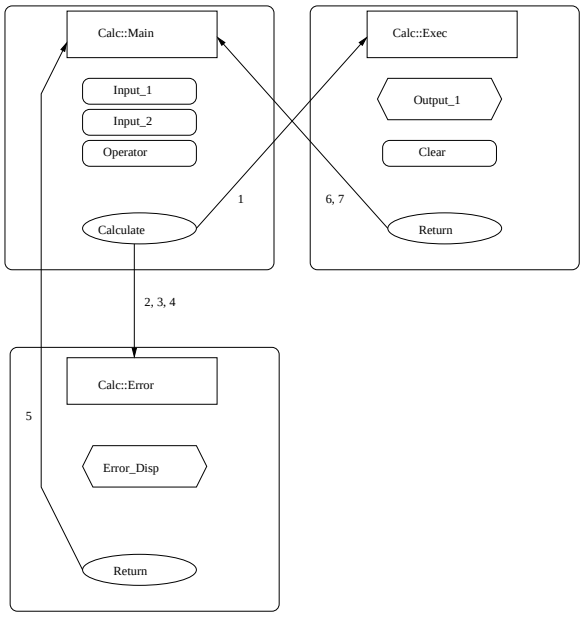

Fig. 4. An example screen transition diagram.

produced by computations. In Figure 4, a screen is denoted as a large rounded rectangle containing the screen name and the objects that are on the screen. The general idea of how this technology can come together with the Lyee methodology is illustrated in Figure 5.

The diagram of Figure 4 depicts requirements for a program named Calc, which takes two numbers and an operator, and then returns the operator applied to the two numbers as long as the numbers and the operator are valid. (At the moment, this is just a hand-drawn depiction of the idea; in practice the style of the diagram is expected to be similar to the style of Figure 3.) In the diagram, input objects are denoted by soft-cornered rectangles within the screens. For example, in the screen Calc::Main, there are the three input objects: Input_1, Input_2, and Operator. An event object (an object capable of generating events) is shown with an oval as with Calculate. Transitions to

screens are shown by connecting an event object to the title of the screen that the event yields with a directed arrow, such as (Calculate, Calc::Exec). Finally, output objects are denoted with a hexagon shape, as with Output_1. Transitions are defined in Table 1 and referred to by the numbers in the table.

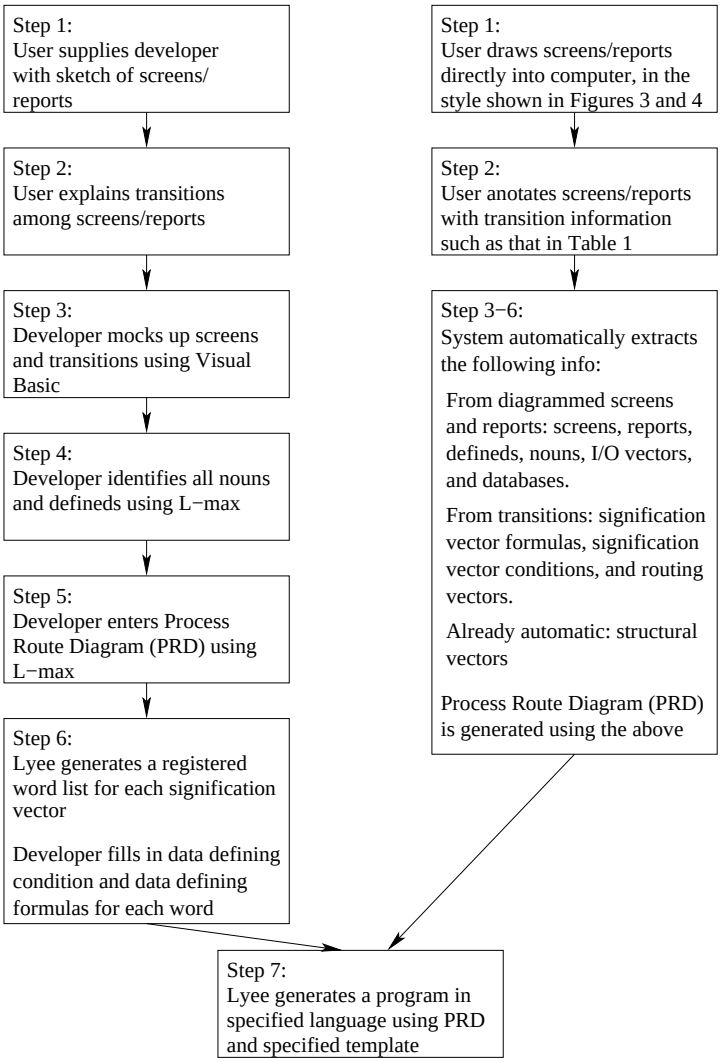

Fig. 5. A strategy (right side) for applying end-user programming to the current (left side) Lyee methodology (also assumes incorporation of work by other collaborators in relevant ways).

The attributes of action objects are as follows. Action objects are what de-

Transition number	Screen	Event	Condition(s)	Destination Screen	Action(s)
1	Calc::Main	Pressed calculate object	AND ((input_1 != NAN) (input_2 != NAN) (OR (operator != /) (AND (operator == /) (input_2 != 0))))	Calc::Exec	output_1 = input_1 operator input_2
2	Calc::Main	Pressed calculate object	(input_1 == NAN)	Calc::Error	Error_Disp =“input 1 is not a number”
3	Calc::Main	Pressed calculate object	(input_2 == NAN)	Calc::Error	Error_Disp =“input 2 is not a number”
4	Calc::Main	Pressed calculate object	AND ((operator == /) (input_2 == 0))	Calc::Error	Error_Disp =“divide by zero”
5	Calc::Error	Pressed return object		Calc::Main	
6	Calc::Exec	Pressed return object	(clear ==“yes”)	Calc::Main	input_1 =“null” input_2 =“null” operator =“null”
7	Calc::Exec	Pressed return object	OR ((clear ==“no”) (clear ==“null”))	Calc::Main	

Table 1
Transitions/actions for screen transition diagram depicted in Figure 4. (“NAN” stands for“not a number”).

scribe and cause computations to occur. They consist of an event, a set of conditions, a destination screen, and a set of actions to be taken. Actions include Lyee’s notion of formulas, and can reference both input and output

objects; they can also affect both input and output objects.

We emphasize that the format shown in Table 1 is only for precision of this discussion, and is not suitable for end users. As Pane and Myers showed empirically [23], 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 [24]. In addition to their suggested alternatives, other possibilities include the demonstrational rewrite rules of Cocoa [25] or Visual AgentTalk [26], which have been demonstrated to be usable by end users.

As the above example illustrates in part, there are three types of objects: input, output, and action. A specialized kind of input object is an event object, which generates a user event if the user interacts with it. A condition on input objects is that they allow the user to enter input, via the keyboard or the mouse, but are also updatable by the program. A condition on output objects is that they cannot receive user inputs. Instead, their purpose is to receive the results of computations, but they can also provide input values to computations.

Action objects are a slightly more powerful form of transition than is traditional, but in this document we use the terms interchangeably. The difference is that the event and conditions that are required to execute the actions are only a partial specification of state; however, the destination screen and the

actions to execute upon taking the transition completely specify a new state. A transition from and to the same screen is legal.

Once an event occurs, the actions are performed and the destination screen is displayed. Those actions performed are the ones in the action object that matches the satisfied conditions. Since this is a deterministic machine, only one transition can be allowed to be possible given an event and conditions. Some possible ways to accomplish this are to require that conditions for a given event are mutually exclusive, that transitions must be prioritized, or that the first transition that meets the criterion for transition will be taken without regard for further applicable transitions.

Events need not require user intervention. For example, an event may be defined as a certain output cell reaching some threshold, or simply the current screen being reached on a given transition.

To summarize this section, we have identified a set of attributes that we assume to be present in a future approach to allowing end users to enter their requirements into the Lyee set of tools. Resting upon these assumptions allows consideration into how end users might test such requirements.

4 Testing End-User Requirements Specifications

Given the above attributes of end users' requirements specifications via screen transition mechanisms, we now consider end-user testing of Lyee requirements specifications using WYSIWYT as a basis.

4.1 WYSIWYT-Based Testing

To make use of the WYSIWYT methodology for testing end-user spreadsheets, a screen transition diagram can be modeled as a set of du-associations associated with each transition. This model will be used to support testing of all of the cases possible under the assumptions stated in this paper, except for transitions into screens that contain no uses.

The following define the definitions and uses in this model:

Definition 1: Definition of input object

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 2: Definition of output object

A *definition* of output object A is:

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

Definition 3: Use

A *use* of object A is:

- a reference to A the right-hand side of an action, or
- a reference to A in a condition.

Building upon these definitions in the same manner as in Section 2, du-associations are interactions between definitions and uses, and the other definitions follow. Using this model, WYSIWYT's dataflow-based testing techniques can be employed to try to cover each du-association. As with WYSIWYT, the system will treat the user as an oracle and allow the user to state whether, given a particular scenario of inputs, results are correct. Figure 6 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 algorithm in Figure 6 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) [27].

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. We 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.

(1) Pick a transition T; let O be the set of objects with uses in T or in T's destination screen.
(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.

Fig. 6. Algorithm sketch for one run of a WYSIWYT test generator for screen transition diagrams.

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 [28]. 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

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, 17, 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 2
The beginning of one possible run of a WYSIWYT-based test generation.

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.

For example, the screen transition diagram in Figure 4 has 27 du-associations,

which are listed in Table 3. The algorithm of Figure 6 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 2.

4.2 Issues Introduced by WYSIWYT for Screen Transition Diagrams

There are two additional important attributes that must be present in order to ensure the properties that are necessary to translate between screen transition diagrams and cell relation graphs. First, spreadsheets require data flow to be preserved locally at the cell level. This is accomplished by nesting "if expressions". To ensure this property translates to the screen transition diagram, there must be a way to control the order of transitions. Second, aiding testing in the spreadsheet paradigm, there is a distinguished value "undefined" that is assigned to cells in which no predicate is satisfied and there is no "else" clause that is reachable. There must also be a similar mechanism in the screen transition paradigm if coverage equivalence is to be ensured in forward and backward translation. Transition 4 in Table 4 is an example of a transition

DU-assoc. number	Definition	Use
1	Input_1 (as program input)	Transition 1 action
2	Input_1 (resulting from Transition 6 action)	Transition 1 action
3	Input_2 (as program input)	Transition 1 action
4	Input_2 (resulting from Transition 6 action)	Transition 1 action
5	Operator (as program input)	Transition 1 action
6	Operator (resulting from Transition 6 action)	Transition 1 action
7	Input_1 (as program input)	Transition 2 condition
8	Input_1 (resulting from Transition 6 action)	Transition 2 condition
9	Input_2 (as program input)	Transition 3 condition
10	Input_2 (resulting from Transition 6 action)	Transition 3 condition
11	Operator (as program input)	Transition 4 condition
12	Operator (resulting from Transition 6 action)	Transition 4 condition
13	Input_2 (as program input)	Transition 4 condition
14	Input_2 (resulting from Transition 6 action)	Transition 4 condition
15	Input_1 (as program input)	Transition 1 condition
16	Input_1 (resulting from Transition 6 action)	Transition 1 condition
17	Input_2 (as program input)	Transition 1 condition (1st occurrence)
18	Input_2 (resulting from Transition 6 action)	Transition 1 condition (1st occurrence)
19	Operator (as program input)	Transition 1 condition (1st occurrence)
20	Operator (resulting from Transition 6 action)	Transition 1 condition (1st occurrence)
21	Operator (as program input)	Transition 1 condition (2nd occurrence)
22	Operator (resulting from Transition 6 action)	Transition 1 condition (2nd occurrence)
23	Input_2 (as program input)	Transition 1 condition (2nd occurrence)
24	Input_2 (resulting from Transition 6 action)	Transition 1 condition (2nd occurrence)
25	Clear (as program input)	Transition 6 condition (1st occurrence)
26	Clear (as program input)	Transition 6 condition (2nd occurrence)
27	Clear (as program input)	Transition 7 condition

Table 3
Definition-use associations in the Calc example.

that would be built-in in some fashion for all output cells as a fall-through to ensure that all values are well defined.

5 Translating Screen Transition Diagrams to CRGs

To show that there is validity in applying WYSIWYT to screen transition diagrams, a loss-less translation method is needed between screen transition diagrams and cell relation graphs (CRGs). CRGs are the formal model used to define testing in the spreadsheet paradigm in WYSIWYT.

Consider the diagram in Figure 7 and the associated Table 4. These represent a program in the screen transition paradigm. (We emphasize that the figure and table are not in a format we would expect an end user to create.) For this example program, there are four input cells EC, Asgn0, Asgn1, AsgnEC and four derived transitions which are triggered any time an input cell is altered. All of these transitions are self transitions because there is one screen in the program. This will serve as a running example in the next few sections of the paper.

#	Screen	Event	Predicate	Dest	Actions
1	Main	Edited any cell	(EC==0 AND !(Error? Asgn0) AND !(Error? Asgn1))	Main Main	Asgn = Asgn0 + Asgn1 Total = Asgn
2	Main	Edited any cell	(EC==1 AND !(Error? Asgn0) AND !(Error? Asgn1))	Main Main	Asgn = Asgn0 + Asgn1 Total = Asgn + AsgnEC
3	Main	Edited any cell	(Error? Asgn0) OR (Error? Asgn1)	Main Main	Asgn = Error Total = Error
4	Main	Edited any cell	Else	Main Main	Asgn = Undefined Total = Undefined

Table 4
Transitions for example screen transition diagram grades

The translation method proceeds as follows: First translate the screen tran-

sition diagram to a set of FAR rules via the method of Section 5.1. Next, translate the FAR rules to a spreadsheet via the method of Section 5.2. Finally, translate the spreadsheet to a CRG via the method in Section 5.3.

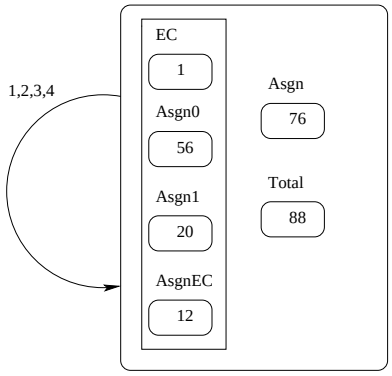

Fig. 7. An example of what the only screen of the grades screen transition diagram might look like if EC were true and Asgn0 and Asgn1 contained no errors. There is a box around user editable cells.

5.1 Screen Transition Diagrams to FAR rules

The screen transition paradigm is highly related to the rule-based paradigm [29]. 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 had side (RHS) of the same expression. In Table 4, 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. Using this idea, our method translates the screen transition paradigm to the rule-based paradigm.

(1) Define a FAR rule for each transition action definition of the form (*C, predicate, consequence*), where C is the LHS of the assignment statement, *predicate* is simply the predicate of the transition and *consequence* is the RHS of the transition action definition.
(2) Define a FAR rule for each input variable of the form (*C, predicate, consequence*), where C is the name of the variable, *predicate* is "always" and *consequence* is the constant which has previously been input.

Fig. 8. Algorithm for translating a screen transition diagram to FAR rules.

This is where FAR (Formulas and Rules) comes in. FAR is an end-user programming language that allows the user to program in either the spreadsheet or rule-based paradigm by representing in both paradigms the program entered [30]. Figure 8 presents an algorithm to translate a screen transition diagram to a collection of FAR rules. FAR's translation rules will then provide the vehicle needed to translate FAR rules to the spreadsheet paradigm and hence to WYSIWYT's formal model, the CRG.

Example 1 : *Derived FAR Rules*

By applying the algorithm in Figure 8 to the screen transition diagram composed of Figure 7 and Table 4, these action and input rules are derived:

- Transition Action Rules (one for each action in Table 4):
  - (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)

(1) For each FAR rule of the form (*C, predicate, consequence expression*), translate to cell C with formula "If *predicate* then *consequence expression*" preserving required nestedness information.
(2) For each FAR rule of the form (*C, "always", consequence expression*), translate to cell C with formula "consequence expression".

Fig. 9. Algorithm for translating FAR rules to a spreadsheet.

- (Total, ((Error? Asgn0) OR (Error? Asgn1)), Error)
- (Total, Else, Undefined)

- Input Rules (one for each input in Figure 7):
  - (EC, Always, 1)
  - (Asgn0, Always, 56)
  - (Asgn1, Always, 20)
  - (AsgnEC, Always, 12)

5.2 FAR Rules to Spreadsheet

The derived FAR rules can now be translated to a spreadsheet via the algorithm in Figure 9.

Example 2 : *Spreadsheet*

Applying the algorithm in Figure 9 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 in Figure 8. Figure 10 sketches what this might look like in a spreadsheet language.

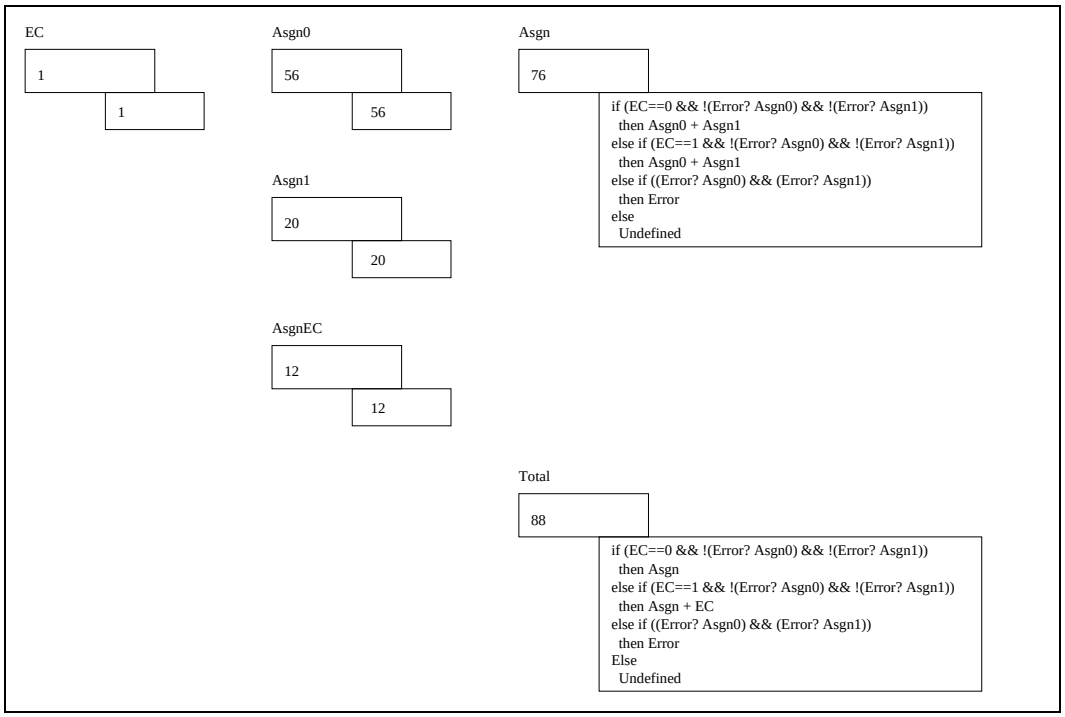

Fig. 10. Grades translated to a spreadsheet via FAR rules. The format in the figure is the cell name above the cell showing its value, with the formula attached to the lower right of the cell.

5.3 Spreadsheet to CRG

Translating a spreadsheet to a CRG is taken directly from [7]. The input cells are translated to three nodes. The first corresponding to an entrance node, the second is a read node and the third is an exit node. Output cells are translated with the entrance and exit nodes as well as control logic exactly mapping the if-then formulas. The algorithm is given in Figure 11

(1) For each constant cell create a collection of three nodes connecting them in the order listed:
- Entry Node
- Read Node
- Exit Node

(2) For each formula cell create a collection of nodes
- Entry Node
- Flow now continues either to a computation node or a predicate node
  - · Predicate Node: each branch (T or F) continuing flow to either another predicate node or to a computation node. In the case of a predicate node the definition is recursive.
  - · Computation Node: the flow continues to the exit node.
- Exit Node

Fig. 11. Algorithm for translating a spreadsheet to a CRG.

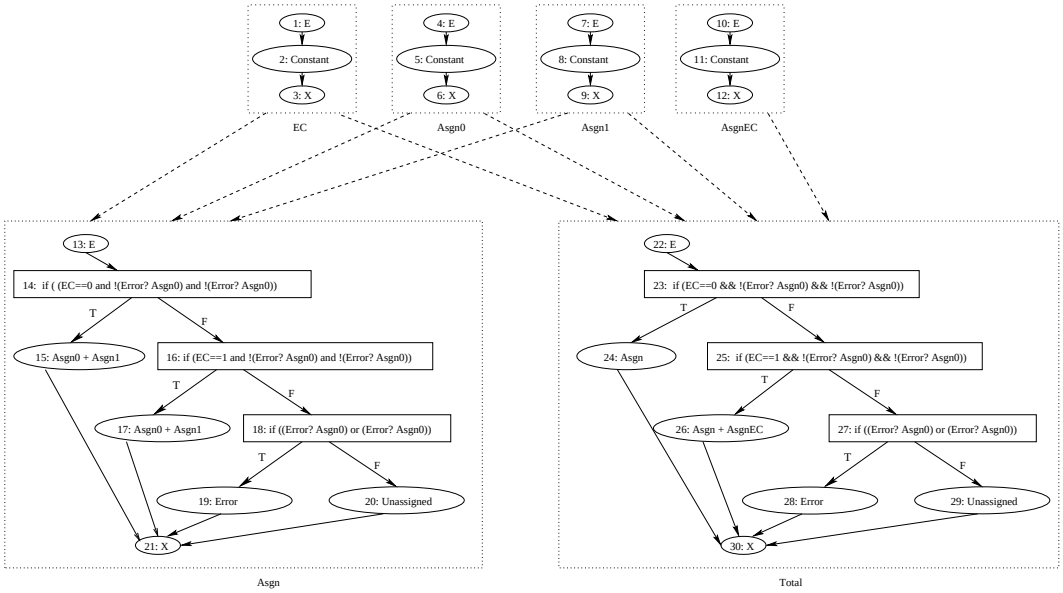

Fig. 12. Grades represented and a CRG.

5.4 Comparing Du-Associations in CRGs and Screen Transition Diagrams

The methodology presented in [5] can be used to find all of the du-associations that are associated with the sample program from the cell relation graph. The following examples start with the sample program that has been the running

example in this paper. Figure 7 and Table 4 showed this program in the screen transition paradigm, and Figure 12 now shows the CRG of the same program. Examples 3 and 4 detail the definitions and du-associations corresponding to the CRG. Examples 5 and 6 then detail the definitions and du-associations in the screen transition version. These examples provide a concrete vehicle for comparison.

Example 3 : *Definitions in CRG*

Definitions in a CRG are all the computations nodes connected to an exit node. Thus, all of the definitions in Figure 12 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
(9) 24,Total
(10) 26,Total
(11) 28,Total
(12) 29,Total

Example 4 : *Du-associations in CRG*

(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

(25) 2,23f,EC
(26) 5,23f,Asgn0
(27) 8,23f,Asgn1
(28) 2,25f,EC
(29) 5,25f,Asgn0
(30) 8,25f,Asgn1
(31) 5,27f,Asgn0
(32) 8,27f,Asgn1
(33) 2,15,Asgn0
(34) 2,15,Asgn1
(35) 2,16,Asgn0
(36) 2,16,Asgn1
(37) 15,24,Asgn
(38) 17,24,Asgn
(39) 19,24,Asgn
(40) 20,24,Asgn
(41) 15,26,Asgn
(42) 17,26,Asgn
(43) 19,26,Asgn
(44) 20,26,Asgn
(45) 11,26,AsgnEC

Example 5 : *Definitions in screen transition diagram*

In Table 4, definitions all occur in the action portion of a transition, or in editing cells. Thus, all of the definitions in Figure 7 and Table 4 are:

(1) Edit EC
(2) Edit Asgn0
(3) Edit Asgn1
(4) Edit AsgnEC
(5) Transition 1 Asgn
(6) Transition 1 Total
(7) Transition 2 Asgn
(8) Transition 2 Total
(9) Transition 3 Asgn
(10) Transition 3 Total
(11) Transition 4 Asgn
(12) Transition 4 Total

Note the one-to-one correspondence of these definitions with those of Example 3.

Example 6 : *Du-associations in 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

In comparing the definitions and definition-use pairs between the CRG and the screen transition diagram, note two points. First, there are an equal number of definitions, and upon closer inspection, each definition in the screen transition diagram maps to a definition in the CRG. There are 13 c-use pairs in each diagram and they map 1:1 between the two representations. Second, while there are 45 du-associations at the CRG level, there are only 21 at the screen transition diagram level.

To understand this difference, consider exactly where it is. There are 8 p-uses at the screen transition level and 32 p-uses at the CRG level. Comparing the two, note that the CRG *repeats* predicates for each cell affected. For example, each transition in Grades affects two cells. Thus, the predicate that is only evaluated once in the screen transition diagram will be repeated twice in the CRG. Also, screen transition diagrams may only have true uses on predicate

branches. Thus, having p-uses on both true and false branches as in the CRGs are irrelevant in screen transition diagrams. Both of these reasons combined explain why there are four times as many p-uses in the CRG as there are in the screen transition diagram. We view this reduction as a potential asset, if it means that equivalent testing can be accomplished in the screen transition paradigm with fewer du-associations than in original WYSIWYT, thus requiring less input from the end user.

6 Conclusion

In this paper we introduced a Lyee requirements model that is both well-suited with the Lyee methodology's underlying structures and has been shown empirically to be useful to end users. We showed that WYSIWYT test adequacy criteria can be used with such a model and presented the necessary conditions for which this is the case. Finally, we presented a systematic translation method from the structure above to the formal model underlying WYSIWYT. Thus, the translation method provides a formal model to define end-user testing for the Lyee methodology using the screen transition paradigm.

We have presented this principle framework in order to show the feasibility of applying the WYSIWYT methodology to the screen transition paradigm as a front end to Lyee end users. The next steps will be to formally show coverage equivalence between the two paradigms, and to consider visualization

techniques for WYSIWYT testing in the screen transition paradigm.

Acknowledgments

We thank the Institute of Computer Based Software Methodology and Technology, Catena Corporation, and Iwate Prefectural University for their support of this project.

References

[1] F. Negoro, I. Hamid, A proposal for intention engineering, in: Int'l. Conf. Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, 2001.

[2] C. Salinesi, M. B. Ayed, S. Nurcan, Development using LYEE: A case study with LYEEALL, Tech. Rep. TR1-2, Institute of Computer Based Software Methodology and Technology (Oct. 2001).

[3] C. Salinesi, C. Souveyet, R. Kla, Requirements modeling in Lyee, Tech. Rep. TR2-1, Institute of Computer Based Software Methodology and Technology (Mar. 2001).

[4] M. Burnett, Bringing HCI research to bear upon end-user requirement specification, in: International Workshop on the Lyee Methodology, Paris, France, 2002, pp. 227–236.

[5] G. Rothermel, M. Burnett, L. Li, C. DuPuis, A.Sheretov, A methodology for testing spreadsheets, ACM Trans. Softw. Eng. Meth. 10 (1).

[6] G. Rothermel, L. Li, M. Burnett, Testing strategies for form-based visual programs, in: International Symp. Software Reliability Engineering, 1997, pp. 96–107.

[7] G. Rothermel, L. Li, C. DuPuis, M. Burnett, What you see is what you test: A methodology for testing form-based visual programs, in: Int'l. Conf. Softw. Eng., 1998, pp. 198–207.

[8] C. Cook, K. Rothermel, M. Burnett, T. Adams, G. Rothermel, A. Sheretov, F. Cort, J. Reichwein, Does a visual 'testedness' methodology aid debugging, Tech. Rep. 99-60-07, Oregon State Univ., Corvallis, OR (Mar. 2001).
URL
`ftp://ftp.cs.orst.edu/pub/burnett/TR.EmpiricalTestingDebug.ps`

[9] V. Krishna, C. Cook, D. Keller, J. Cantrell, C. Wallace, M. Burnett, G. Rothermel, Incorporating incremental validation and impact analysis into spreadsheet maintenance: An empirical study, in: Int'l. Conf. Softw. Maint., 2001, pp. 72–81.

[10] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. Green, G. Rothermel, An empirical evaluation of a methodology for testing spreadsheets, in: Int'l. Conf. Softw. Eng., 2000, pp. 198–207.

[11] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, S. Yang, Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm, J. Func. Prog. 11 (2) (2001) 155–206.

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

[13] M. Burnett, B. Ren, A. Ko, C. Cook, G. Rothermel, Visually testing recursive programs in spreadsheet languages, in: IEEE Symp. Human-Centric Comp. Lang. and Env., 2001, pp. 288–295.

[14] M. Burnett, A. Sheretov, B. Ren, G. Rothermel, Testing homogeneous spreadsheet grids with the 'what you see is what you test' methodology, IEEE Trans. Softw. Eng. 28.

[15] A. Aho, R. Sethi, J. Ullman, Compilers, Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.

[16] E. Weyuker, Axiomatizing software test data adequacy, IEEE Trans. Softw. Eng. 12 (1986) 1128–1138.

[17] P. Frankl, E. Weyuker, An applicable family of data flow criteria, IEEE Trans. Softw. Eng. 14 (1988) 1483–1498.

[18] J. Laski, B. Korel, A data flow oriented program testing strategy, IEEE Trans. Softw. Eng. 9 (1993) 347–354.

[19] S. Rapps, E. Weyuker, Selecting software test data using data flow information, IEEE Trans. Softw. Eng. 11 (1985) 367–375.

[20] E. Weyuker, More experience with dataflow testing, IEEE Trans. Softw. Eng. 19 (1993) 912–919.

[21] J. Landay, Interactive sketching for the early stages of user interface design, Ph.D. thesis, Carnegie Mellon Univ., Pittsburgh, PA (Dec. 1996).

[22] J. Landay, B. Myers, Sketching interfaces: Toward more human interface design, Computer 34 (2001) 56–64.

[23] J. Pane, B. Myers, Tabular and textual methods for selecting objects from a group, in: IEEE Symp. Vis. Lang., 2000, pp. 157–164.

[24] J. Pane, B. Myers, L. Miller, Using HCI techniques to design a more usable programming system, in: IEEE Symp. Vis. Lang., 2002, pp. 199–206.

[25] N. Heger, A. Cypher, D. Smith, Cocoa at the visual programming challenge 1997, J. Vis. Lang. and Computing 9 (1998) 151–169.

[26] A. Ioannidou, A. Repenning, End-user programmable simulations, Dr. Dobb's Journal 24 (1999) 40–48.

[27] M. Fisher II, G. Rothermel, C. Cook, M. Burnett, Automated Test Generation for Spreadsheets, in: Int'l. Conf. on Software Engineering, Orlando, FL, 2002, pp. 141–151.

[28] A. Blackwell, T. Green, Investment of Attention as an Analytic Approach to Cognitive Dimensions., in: T. Green, R. Abdullah and P. Brna (Eds.) Collected Papers Workshop. Psychology of Programming Interest Group, 1999, pp. 24–35.

[29] M. Lin, J. Malec, S. Nadjm-Tehrani, On semantics and correctness of reactive rule-based systems, in: Proc. Andrei Ershov Third Int'l. Conf. Perspectives of System Informatics, 1999, pp. 235–246.

[30] M. Burnett, S. Chekka, R. Pandey, FAR: An end-user language to support cottage e-services, in: IEEE Symp. Human-Centric Comp. Lang. and Env., 2001.