Dynamic, Incremental Assertion Propagation in End-User Programming

Eugene Creswick, Jay Summet, Margaret Burnett, Christine Wallace, Curtis Cook, Gregg Rothermel

Oregon State University

School of Electrical Engineering and Computer Science

{creswick, burnett, wallacch, cook}@cs.orst.edu, grother@cse.unl.edu,summetj@cc.gatech.edu

Abstract

End-user programming is growing at a rapid rate, but there has been only a little in the way of tools or environments to improve the correctness of programs created by end users. We describe an approach to dynamic assertions in one of the most widely used end-user programming paradigms—the spreadsheet paradigm. Our approach does not assume any formal knowledge of, or interest in, software engineering practices. Dynamic assertions, which can be entered incrementally, feature deductive propagation from user-entered assertions through spreadsheet formulas. These propagated dynamic assertions can then be compared with other user-entered dynamic assertions and—in the event of a conflict—can alert the user to the possibility of a bug in the spreadsheet formulas.

Deductive propagation, however, is not viable in all situations. We present algorithms for propagating dynamic assertions through spreadsheets that obey a particular set of restrictions, and we evaluate these algorithms in regard to four properties: reliability, correctness, responsiveness and usefulness. We present lower bounds on the classification of the propagation problem for the variants where deductive propagation is not viable. As part of our evaluation of the reliability property, we empirically examine the occurrence of these variants in a corpus of real-world spreadsheets, both demonstrating that our approach does improve the reliability of assertion propagation and identifying directions for further research.

1 Introduction

End-user programming has spread widely, in such varied manifestations as CAD applications, e-mail filtering rules, spreadsheets, and multimedia authoring. Many people using these applications do not have any programming background, and it seems unlikely that they will want to learn programming techniques, but the need for software engineering support in these environments is real. Boehm and Basili observe that 40-50% of the software created by end users contains non-trivial faults [8]. This is further supported by Panko's reports of spreadsheet errors [33, 34, 35].

Typical software engineering approaches, however, are not directly applicable to end-user programming for several reasons: it is not reasonable to expect end-user programmers to climb the learning curve required for tools based on these approaches, end users do not usually create the formal specifications needed by many of these tools, and many software engineering approaches are designed for batch operation whereas most end-user programming environments are interactive in nature. To help address this problem, we have developed a paradigm of software engineering for end-user programmers, termed *end-user software engineering*, and have implemented our work to date in Forms/3 [10, 12], a research spreadsheet language. One of the techniques we have examined within the spreadsheet paradigm involves the use of assertions.

When creating a spreadsheet, the user has a mental model of how it should operate. The formulas he or she enters provide one approximation of this model, but unfortunately these formulas may contain inconsistencies or faults. These formulas are only one representation of the user's model of the problem and its solution; they contain information on how to generate the desired result, but they do not provide ways for the user to communicate other properties. Traditionally, assertions have provided a means for professional programmers to make these expectations explicit and catch exceptions. Our dynamic approach to assertions attempts to provide these same advantages to end-user programmers.

Assertions are a potentially attractive addition to end-user programming for three reasons. First, as we have just explained, assertions provide a way to make explicit a user's mental model of the program, essentially integrating "specifications" with that program. Second, assertions can be used in an incremental manner; it is possible to enter one or two assertions and gain some value without the commitment to enter a purportedly complete set of specifications. Third, assertions in the form of preconditions, postconditions and invariants have a proven track record of increasing effectiveness in software maintenance and debugging by professional programmers [37].

In this paper we use the term *dynamic assertions* to describe initial assertions (such as assertions entered by the user) that are deductively propagated through the program, as well as the assertions derived through propagation. In contrast to dynamic assertions, the term *static assertions* will be used to describe assertions that are not propagated. When multiple dynamic assertions meet on the same cell, the assertions can be compared to cross-check the program with the specifications (the user-entered assertions). If there is a conflict between these assertions, there are only two possible sources for this inconsistency: the spreadsheet formulas that were used to derive the assertions and the initial user-entered assertions from which the derived assertions were propagated. For this reason, dynamic assertion conflicts often identify faults in the program or faults in the user's mental model of how the program should operate.

In this paper we describe an approach to dynamic assertions for end-user programmers, we identify variants of the assertion propagation problem, we present algorithms that propagate dynamic assertions through some of these variants, and we present open problems regarding propagation.

2 Related Work

Dynamic assertions touch on many areas in the field of Computer Science. Of these, four primary areas have generated contributions that are closely related to our approach for dynamic assertions. These areas are:

- assertions for professional software developers,
- assertions for end-user programmers,
- constraint programming, and
- automatic invariant generation.

Assertions for Professional Software Developers: For professional software developers, the only widely used language that natively supports assertions is Eiffel [27]. To allow programs in other languages to share at least some of the benefits of assertions, methods to add support for assertions to languages such as C, C++ and Awk have been developed [3, 50]. All these approaches have provided *static* assertions. Despite the static nature of these assertions, the application of such assertions to software engineering problems has proven promising. For example, there has been research on deriving runtime consistency checks for Ada programs [38, 42]. Rosenblum has shown that these assertions can be effective at detecting runtime errors [37]. However, these traditional approaches are not geared toward end users.

Assertions for End-User Programmers: Microsoft Excel, which is currently the most widely used end-user programming environment, provides users with a "data validation" tool, which is similar to static assertions. Users are able to specify a list or range of acceptable values for a cell. However, the value in that cell is checked against these limits only when the user edits that cell or when the user explicitly requests that all violations be shown. It is therefore possible for a visible cell to contain a value that is out of range without the system notifying the user. Excel also does not reason in any form about the user-specified limits. Reasoning about assertions deductively is an important aspect of our approach to dynamic assertions.

Ayalew et al. have also studied the application of assertions to debugging in the spreadsheet paradigm [4]. Their approach, "interval testing", takes user-entered ranges and calculates intervals on cells that are dependent on those with specified ranges. These intervals are then compared and, if they conflict, heuristics are used to determine the "most influential faulty cell". This cell is then shaded to indicate that it may be the source of an error. Our work differs from that of Ayalew in a number of ways. From a human perspective, our approach includes a strategy that takes into consideration users' curiosity and attention, and also reduces the need for memorization. From an algorithmic perspective, interval testing relies entirely on interval arithmetic to propagate assertions, which is not sufficient for many real-world spreadsheets [4]. (In Section 6.2 we present an examination of real-world spreadsheets which demonstrates

this.) Later in this paper, we identify specific situations in which interval arithmetic is not sufficient and we present an algorithm for dynamic assertion propagation through some of these situations.

Constraint Programming: Assertions are similar to the constraints used in Constraint Programming [26] in that they define a set of valid values that a program can output. When programming with constraints, a query is specified as a set of constraints that are solved by a constraint solver [25]. These constraints limit the search space to a set of answers that obey the constraints. Myriad problems can be solved in this manner; however, the way the problem is stated can affect the solver's ability to find a solution. An example of this is the problem of finding the roots of a quadratic equation such as $x^2+6x+10=0$. A typical constraint solver will not be able to "discover" the quadratic formula, and therefore cannot find the roots of all quadratic equations.

Information such as the quadratic formula in this example is based on background knowledge about the problem. In constraint programming, this information is provided by a domain expert. In the quadratic equation example, the quadratic formula $\left(x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}\right)$ would be provided by a domain expert. Since many end-user programming environments, such as Excel, are used for a very wide range of tasks, our approach cannot predict the domains that will be relevant, and the approach does not have access to an expert to convey this knowledge to the system. Given all the needed background knowledge, a constraint program often can blur the boundary between inputs and outputs. In ThingLab [9], for example, constraints are placed on objects through a visual interface. The user can then modify any mutable values to see the effect on other values in the program.

Constraints have been used for programming tools that target specific domains as well. Carlson et al. have applied constraints to generate algorithm animations that can be run forward and backward to facilitate algorithm education [13]. Griebel et al. incorporated constraints with Pictorial Janus to create animations [20]. Myers et al. have also addressed animation generation for interfaces through the use of constraints in Amulet [30]. In the past constraints have been applied to spreadsheets, resulting in multi-paradigm environments [24, 43]. However in contrast to assertions, which critique values that have been generated by independent logic, constraints actively generate values themselves.

Automatic Invariant Generation: In addition to methods that allow the programmer to define assertions, methods have been developed that generate invariants automatically. Ernst has examined the use of statistical methods to detect likely program invariants by extensive examination of a program's behavior over a test suite [16, 31]; these techniques have been implemented in Daikon [15]. Our approach differs from Ernst's invariants in that our dynamic assertions are created directly from user input and from *deductive* propagation. We have chosen deduction because we believe it is important that an approach targeted at end users generate tight and relevant assertions on outputs. The assertions generated by our approach fit this requirement, although there are some situations through which our approach is unable to propagate. Because of the statistical nature of Ernst's work, generated invariants can sometimes be incorrect. Our work also differs in that Ernst's invariants are designed to be used in batch by professional programmers, while

our approach is interactive.

DIDUCE [21] deduces invariant assertions and uses them to check program correctness. DIDUCE has a training phase, in which it considers all behaviors to be correct and relaxes invariants to encompass these behaviors. A checking phase then reports violations to the invariants inferred in the training phase. Raz et al.'s approach to semantic anomaly detection [36] uses off-the-shelf unsupervised learning and statistical techniques, including a variant of Daikon, to infer invariants about incoming data from online data feeds. Raz et al. have conducted empirical work that shows effectiveness. Recent work that can be described as inferring assertions related to correctness of end-user programs involves automatic detection of errors through outlier analysis [28]. This approach is similar to that of Raz et al., but it has been developed in the domain of programming-by-demonstration for text processing.

Jeffords and Heitmeyer have presented a system which automatically generates state invariants from an operational (model based) requirements specification expressed in SRC (Software Cost Reduction) tabular notation [22]. Bjørner et al. presented a formal proof system that attempts to prove a given goal by generating intermediate assertions [7]. This system is implemented in STeP, the Stanford Temporal Prover [6]. These approaches require their users to have some knowledge of proof systems and requirements specifications, and it does not seem reasonable to expect end-user programmers to have such background knowledge.

3 Dynamic Assertions for End Users

Two factors motivated our research on dynamic assertions: First, end-user programming languages are often interactive and incremental, and therefore dynamic assertions should also possess these properties. Second, end users should not be expected to have knowledge about, or interest in, software engineering practices. Therefore dynamic assertions should not require such knowledge or interest.

We have prototyped our approach to dynamic assertions for end-user programmers in Forms/3 [10, 12], a research spreadsheet language. Figure 1 shows a spreadsheet in Forms/3 that a user is in the course of changing. (The user's changes are discussed in Section 3.2.) One obvious difference from commercial spreadsheet systems is that in Forms/3, cells are not locked to a grid. Cells can also be given meaningful names (by the user), such as `input_temp`, which represents a user-entered temperature for the spreadsheet to convert.

Currently our prototype supports three sources of assertions. *User assertions* are entered by the user directly, and are displayed next to a stick figure. User assertions can be seen in Figure 1, on the `input_temp` cell. User assertions provide a base for *system-generated assertions*. System-generated assertions result from deductively propagating assertions through spreadsheet formulas in the direction of data-flow, and are dynamically updated whenever the user edits a formula or an assertion. System-generated assertions are displayed next to a computer icon. (The methods used to calculate these assertions are presented in Sections 4 and 5.) The `output_temp` cell in Figure 1 has both user and system-generated assertions. *Help Me Test assertions (HMT assertions)* are automatically generated assertions based

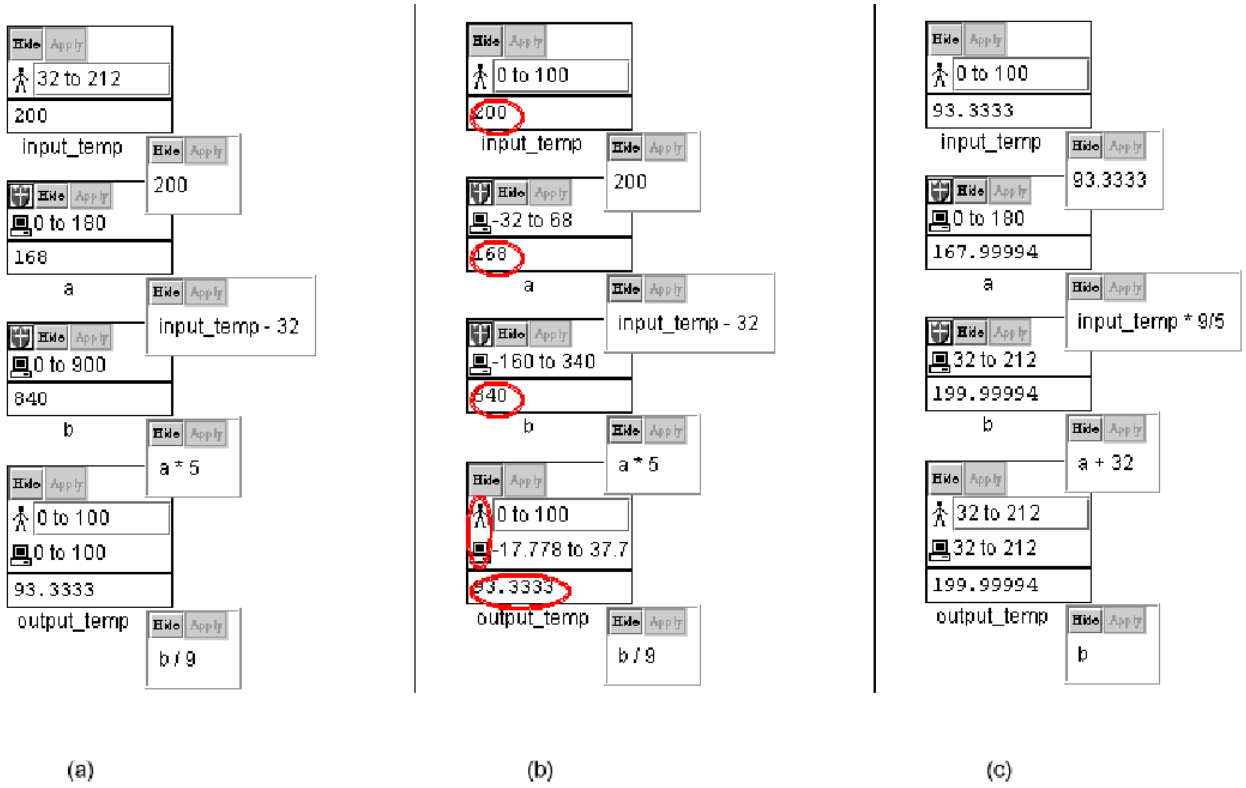

Figure 1: A simple temperature conversion spreadsheet shown at three stages as an end user changes the spreadsheet to convert from Celsius to Fahrenheit. Assertions are shown above each cell, and formulas are in the boxes at the lower right of each cell.

on possible input values found by Help Me Test [18], an automatic test-case generation tool[1]. HMT assertions are propagated in the same manner as user assertions, and are briefly discussed in Section 6.1.

3.1 Dynamic Assertions from the User's Perspective

In the field of professional software engineering, specifications play a large part in the software engineering process. Similarly, in our vision of end-user software engineering, the user's mental model of the program plays a large part in the development of their program. Although some of the user's model can be expressed via spreadsheet formulas, some aspects cannot, such as constraints on values. User assertions alone, even in a static approach to assertions, have the benefit of providing a means for the user to inform the system of these constraints in their mental model. A dynamic approach to assertions brings even more benefits. Dynamic assertions add three ways of potentially helping end users:

- First, dynamic assertions act as "guards" for the values in a cell, just as static assertions do. If the value in a cell falls outside any of the assertions on that cell, a *value violation* occurs and a red violation oval is drawn around the value in the cell.

- Second, because dynamic assertions allow for multiple sources of assertions, the assertions from each source can be compared with the other assertions on the cell. If any of these assertions are not in complete agreement, an *assertion conflict* occurs and a red assertion conflict oval is drawn over the disagreeing assertions (`output_temp` in Figure 1(b) has such an oval), alerting the user to the possible failure in the program. If a user assertion is ever in conflict with a system-generated assertion there is either a bug in the spreadsheet formulas or an incorrect user assertion. This guarantee is a key aspect of the advantages of dynamic assertions.

- Third, when a formula is incorrect, the system-generated assertion may not fit with the user's mental model of the spreadsheet. Dynamic assertions provide this alternate view of a program through deductive propagation. This can result in a system-generated assertion that may not "look right" to the user. Section 3.2 discusses a study in which a user's program contained an extra division operator. The resulting system assertion of `3.5556 to 23.5556` alerted the user to the fault.

End users interact with assertions in our current prototype through two interfaces: a graphical interface designed to assist the first-time user and a more concise textual interface for the more experienced user. The graphical interface is presented as a separate window (shown in Figure 2) which is opened by the user double-clicking on a cell's assertion tab. This window presents a graphical depiction of the assertion. Currently this interface in the prototype supports only numerical ranges, and displays a graphical number-line on which the user can place, remove, add, and move points and ranges. After specifying an assertion through this interface, the user applies the assertion which is then displayed

[1] Forms/3 presents users with the ability to visually test their spreadsheet [39, 40, 41]; however, coming up with test cases is sometimes difficult. Help Me Test generates values to assist users with this.

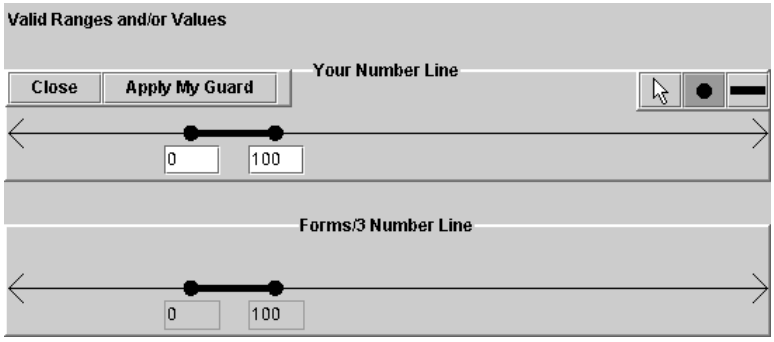

Figure 2: The graphical interface to assertions in Forms/3.

in a more compact syntax above the cell. This is the second interface—the *assertion bar*. Assertion bars exist above all cells, although they are hidden if they are empty (empty assertions represent *null* assertions, which accept all values). Users can display assertion bars for a cell by clicking on its assertion tab. Once an assertion bar is visible, the user may edit the text on the bar directly (if the assertion is a user assertion or HMT assertion), and then click "apply" to change the assertion. All of the assertions in Figure 1 are displayed through assertion bars.

3.2 Example

This example demonstrates dynamic assertions by stepping through a simple task for an end user. Figure 1(a) shows a Forms/3 spreadsheet that converts temperatures in degrees Fahrenheit to degrees Celsius. The `input_temp` cell has a constant value of 200 in its formula and is displaying the same value. There is a user assertion on this cell that alerts the user (with a violation oval) if the value of the cell is below 32 or above 212. (The assertion does not prevent such values from being entered.) The formulas of the `a`, `b`, and `output_temp` cells each perform one step in the conversion, first subtracting 32 from the original value, then multiplying by five, and finally dividing by nine. Cells `a` and `b` have assertions generated by the system (as indicated by the computer icon), which reflect the propagation of the user assertion on the `input_temp` cell through their formulas. The spreadsheet's creator has told the system that the `output_temp` cell should range from `0 to 100`, and the system has agreed with this assertion. This agreement was determined by dynamically propagating the user assertion on the `input_temp` cell through the formulas in cells `a`, `b`, and `output_temp`, and comparing it with the user assertion on the `output_temp` cell.

Suppose a user has decided to change the direction of the conversion to make the spreadsheet convert from degrees Celsius to degrees Fahrenheit. We conducted a study in which users were asked to perform this task. What follows is a summary of one user's behavior [46]. The quotes are from a recording of the subject's commentary.

First, the user changed the assertion on `input_temp` to range from `0 to 100`. This caused several red violation ovals to appear, as in Figure 1(b), because the values in `input_temp`, `a`, `b`, and `output_temp` were out of range and the

assertion on output_temp was in conflict with the previously specified assertion for that cell. The user decided "that's OK for now," and changed the value in input_temp from 200 to 75 ("something between zero and 100"), and then set the formula in cell a to "input_temp * 9/5" and the formula in cell b to "a + 32".

At this point, the assertion on cell b was 32 to 212. Because the user combined two computation steps in cell a's formula (multiplication and division), the correct value appeared in cell b, but not in output_temp (which still had the formula "b / 9"). The user now chose to deal with the assertion conflict on output_temp, and double-clicked on the guard icon to view the details graphically. Seeing that the system-generated assertion specified 3.5556 to 23.5556, the user stated "There's got to be something wrong with the formula" and edited output_temp's formula, removing the division operator and making it a reference to cell b. This corrected the value in output_temp, although an assertion conflict still existed because the previous user assertion remained at 0 to 100, which no longer matched the system assertion of 32 to 212. Comparing the system-generated assertion with the user assertion, the user saw the discrepancy and changed the user assertion to agree, which removed the final conflict. Finally, the user tested by setting input_temp to 93.3333, the original output value, to see if it resulted in approximately 200, the original input value. The results were as desired, and from this the user (correctly) determined that the task was complete.

3.3 Assertions Concepts and Definitions

The example in Section 3.2 motivates dynamic assertions as a useful tool for an end user. This section describes these dynamic assertions from a theoretical point of view. To show the expressive power of assertions, we consider them in terms of an abstract syntax. An assertion on a cell guarded_cell is in the form of a tuple:

(guarded_cell, {*and-assertions*})

Where:

- each *and-assertion* is a set of *or-sub-assertions*
- each *or-sub-assertion* is a set of (*unary-relation*, *value-expression*) and (*binary-relation*, *value-expression-pair*) tuples
- each *unary-relation* $\in \{=, <, \leq, >, \geq\}$
- each *binary-relation* $\in \{$*to-closed*, *to-open*, *to-openleft*, *to-openright*$\}$
- each *value-expression* is a valid formula expression in the spreadsheet language
- each *value-expression-pair* is simply a pair of *value-expressions*

Despite the apparent simplicity of the concrete syntax described in Section 3.1, it is just as powerful as the abstract syntax above. Each assertion bar (and-assertion) is AND'ed with all other assertion bars; each assertion bar contains a

comma-delimited list of sub-assertions (or-sub-assertions) that are OR'ed together. Unary and binary expressions can be expressed within each sub-assertion. Each sub-assertion represents a range (a binary-relation) defining allowable values for the cell. Unary relations can be specified by various combinations of sub-assertions (binary-relations). For example, the relation "$< n$" can be expressed as "-infinity to n" where n is a value-expression. (Our prototype currently supports only constant value-expressions.)

In this document, sub-assertions are represented with the standard notation for intervals: $[a\ b]$ represents the inclusive range from a to b, including the end points, expressed in the abstract syntax as (*to-closed, a, b*); $(a\ b)$ represents the exclusive (asymptotic) range from a to b, excluding the endpoints, expressed in the abstract syntax as (*to-open, a, b*). For ease of reasoning, single points are represented by degenerate ranges such as $[b\ b]$. The absence of an assertion, also called a *null* assertion, is interpreted as the all-encompassing assertion $[-\infty\ \infty]$, which cannot be violated.

4 Propagation

The ability to propagate assertions through program logic to determine the valid outputs for a spreadsheet program is at the core of the empirically demonstrated benefits of our dynamic approach to assertions. In this section we present methods of deductively propagating assertions through program logic. We begin with a precise definition of the propagation problem:

> Given the spreadsheet as a graph of cells, with formulas and the assertions on those cells, for each *relevant cell* `C`, compute a system-generated assertion for `C`, where a *relevant cell* is any cell in the forward static slice [2] of a cell whose formula or assertion has been modified.

We further identify four properties by which we will evaluate solutions to the propagation problem:

Property 1: Reliability. Given a cell `C`, `C`'s assertion can be computed if all `C`'s ancestors have assertions[3].

Property 2: Correctness. A system-generated assertion will be termed *correct* if it accepts every output that can be produced by the cell's formula given inputs accepted by assertions on the cell's parents, and rejects all other values. We evaluate Correctness later in this section.

Property 3: Responsiveness. Assertions are responsive if they do not interfere with the property of immediate visual feedback which is characteristic of the spreadsheet paradigm.

Property 4: Usefulness. Assertions must be useful to their intended audience: end users creating spreadsheets.

[2] A cell `C` is in the forward static slice of a modified cell `A` if `C` is dependent on `A`—meaning the value of `A` possibly can affect the output of `C`. Slicing was introduced by Weiser as a technique for analyzing program dependencies [47].

[3] In most cases this requirement is overly conservative. For many common situations, propagation is possible if all of `C`'s parents have assertions.

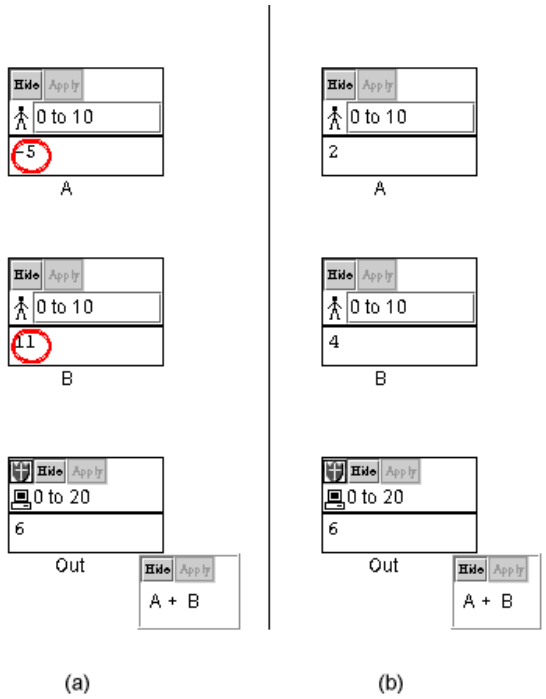

Figure 3: In (a) both input assertions are violated. However, the resulting sum is within the range of the output. In (b) no assertions are violated, and `Out` has the same value as in (a).

To complete the description of Property 2, we need to define "acceptance". Given an and-assertion, A, consider one of A's sub-assertions $A_k = [A_{k_l}\ A_{k_u}]$ where A_{k_l} and A_{k_u} are the lower and upper bounds on the sub-assertion, respectively, with the property: $A_{k_l} \leq A_{k_u}$. Recall from Section 3.3 that A_{k_l} and A_{k_u} constitute a *value-expression-pair*, and A_k represents an *or-sub-assertion*, that is part of A. A *accepts* a value v if and only if $A_{i_l} \leq v \leq A_{i_u}$, for at least one of A's sub-assertions A_i. More concisely we say that $A(v)$ is true if there exists an assertion A_i such that $A_{i_l} \leq v \leq A_{i_u}$ and false otherwise.

One subtlety of the definition of Correctness bears closer examination. Given a cell O, it is possible that some or all of O's parents contain values that violate their assertions even though O's value is within the propagated assertion on O. Figure 3(a) demonstrates this with cell `Out`. The absence of a violation oval on cell `Out` in Figure 3(a) is because it is indeed possible to generate `Out`'s value from a valid set of inputs, as shown in Figure 3(b).

Given these definitions, we now turn our attention to deductive propagation. It turns out that propagating dynamic assertions through arbitrarily complex spreadsheets is a difficult problem. One of the contributions of this paper is the identification of several variations of the propagation problem and the restrictions required to propagate assertions when these variations occur. Recall from Section 3.1 the guarantee that in the event of an assertion conflict, either a formula or a user assertion is incorrect. To maintain this guarantee, the system must be able to calculate *correct*

assertions (as defined by Property 2). Without the Correctness property, users may lose trust in the system, which could result in the user losing interest in providing any assertions at all [14].

We have identified three variants of the propagation problem that each require different approaches to propagate assertions. These three variants are described in the following sections:

- Propagation through trees (Section 4.1).
- Propagation through shared data-flow dependencies (Section 4.2).
- Propagation through control-flow dependencies (Section 4.3).

In the following sections we examine these variants, present situations that pose difficulties for propagation, provide algorithms to propagate dynamic assertions through some of these situations, and evaluate these algorithms with regard to Properties 1-3. Section 6 evaluates our approach to dynamic assertions regarding Property 4.

4.1 Propagation Through Trees: α-Propagation

The first variant of the assertion propagation problem occurs when, given a cell *C*, the following three conditions hold:

1. The cell `C` has no shared dependencies. (The data-flow graph is a tree.)
2. All of `C`'s parents have assertions.
3. Assertion-specific operators have been provided for each formula operator in `C`'s formula.

Our solution to this variant is termed α-propagation.

We discuss the first condition, shared dependencies, in Section 4.1.3 where we address the limitations of α-propagation. Recall from Section 3.3 that the absence of an assertion is treated as an all-encompassing assertion, $[-\infty\ \infty]$, which cannot be violated. Thus the second condition is always satisfied, since all cells have an assertion of some sort (either a user assertion, a system-generated assertion, or an all-encompassing assertion). Regarding the third condition, in our α-propagation approach, each formula operator is paired with an assertion-specific operator that performs a function on the appropriate sub-assertion using interval arithmetic. For example, the assertion+ operator adds two sub-assertions together.

When a formula or assertion changes, the changes must be propagated forward through the downstream cells. α-Propagation propagates assertions through data-flow trees by applying the procedure described in Algorithm 1 to each cell in the spreadsheet. Here we will examine Algorithm 1 by stepping through an example using the assertion+ operator, referring the reader to [44] for details of other assertion-specific operators.

In lines 1-3 of Algorithm 1, the system replaces all constants and cell references in `D`'s formula with the assertions on `D`'s parents, which are termed *input assertions*. For example, if `D`'s formula is "2+`A`", the conversion results in "[2 2] + $[A_l\ A_u]$" where [2 2] is a degenerate range representing the constant 2, and $[A_l\ A_u]$ is a sub-assertion on the cell

Algorithm 1 Overview of the α-propagation algorithm.

Input: Cell D, Spreadsheet S.

1: Build the assertion-specific formula from D's formula:
2: Replace operands with assertions.
3: Replace formula operators with assertion-specific operators.
4: Evaluate the assertion-specific formula.
5: Merge sub-assertions.

$$
\begin{aligned}
i_1 &= A_l \; op \; B_l \\
i_2 &= A_l \; op \; B_u \\
i_3 &= A_u \; op \; B_l \\
i_4 &= A_u \; op \; B_u
\end{aligned}
$$

$$O_r = [min(i_1, i_2, i_3, i_4) \; max(i_1, i_2, i_3, i_4)]$$

Figure 4: The five equations here represent the general form of interval arithmetic.

A. The system then replaces each operator with its assertion-specific version. In our example, this generates "[2 2] assertion+ $[A_l \; A_u]$".

Once this assertion-specific formula is built, the formula is evaluated (line 4 of Algorithm 1), resulting in an assertion for cell D. In our example, the evaluation performs the following calculations:

$$
\begin{aligned}
i_1 &= 2 + A_l \\
i_2 &= 2 + A_u \\
i_3 &= 2 + A_l \\
i_4 &= 2 + A_u
\end{aligned}
$$

More generally, an assertion is calculated using the equations in Figure 4, which calculate an output assertion O_r given the input sub-assertions $A_r = [A_l \; A_u]$ and $B_r = [B_l \; B_u]$, and the formula operator op. While it is not strictly necessary to calculate the *min* and *max* values for addition (since i_1 is always the minimum, and i_4 is always the maximum), we do so here because the other arithmetic[4] operators (-,*,/) require this additional step, and thus it gives a general idea of how they all work.

In the simplest case of assertion-specific formula evaluation, only two input sub-assertions are involved. In the more complex case, where one or both input assertions contain multiple sub-assertions, the substitution and evaluation steps are performed on all combinations of input sub-assertions, generating a set of output sub-assertions. In this case each sub-assertion A_i is evaluated with each sub-assertion B_j, to generate a set of sub-assertions O_{ij}.

[4] Assertion-specific operators for non-arithmetic operators, such as trigonometric operators, can be implemented using first and second derivative tests to generate assertions by calculating the minimum and maximum values for the operator in the specified range.

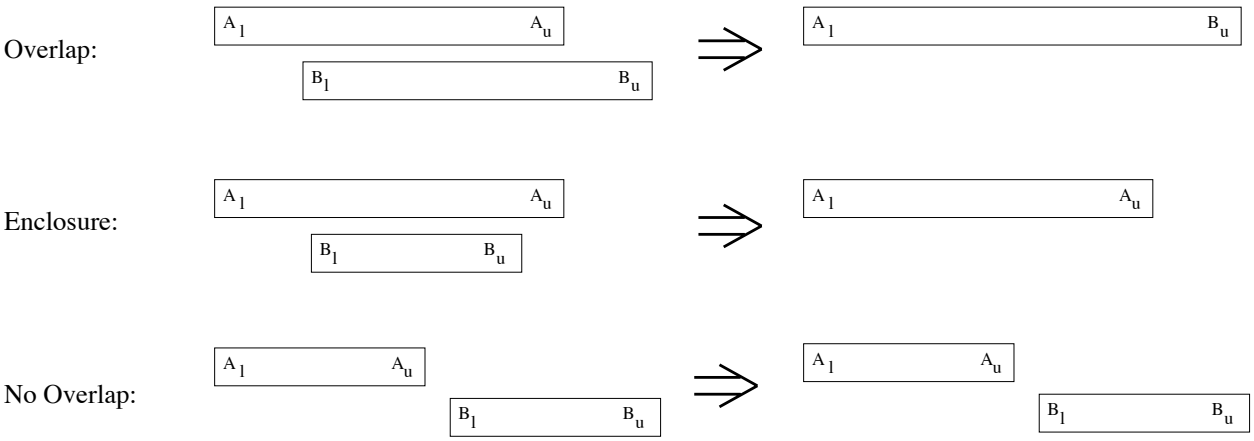

Figure 5: The three possible situations when merging sub-assertions.

α-Propagation deals with complex formulas by using a divide-and-conquer approach. Each operator is addressed individually, and in the order specified by the order of operations in the host language. For example, consider a cell `D` with the formula "`A` − `B` × `C`". Assuming "×" has higher precedence than "-" in the host language, an assertion for "`B` × `C`" is calculated first, and then that assertion is used with the assertion on `A` to find the final assertion for `D`.

In some cases, the propagation algorithm may generate multiple sub-assertions for a cell. This may occur because input assertions have multiple sub-assertions, or a formula may split a sub-assertion by introducing discontinuities. In this case, α-propagation merges the sub-assertions by OR'ing them together (line 5 of Algorithm 1) with an assertion-merging algorithm.

Given two sub-assertions, $A_r = [A_l\ A_u]$ and $B_r = [B_l\ B_u]$ where $A_l \leq B_l$, an assertion-merging algorithm considers three situations to when merging assertions (Figure 5):

Overlap If $A_l \leq B_l \leq A_u \leq B_u$ the sub-assertions overlap, and a new sub-assertion $[A_l\ B_u]$ is generated.

Enclosure If $A_l \leq B_l \leq B_u \leq A_u$, one sub-assertion encloses the other, and the larger sub-assertion is used, discarding the smaller.

No Overlap If $A_u < B_l$ the sub-assertions do not intersect, and they cannot be merged. These sub-assertions are left as individual or-sub-assertions.

4.1.1 Correctness of α-Propagation

Theorem 1 *α-Propagation correctly propagates assertions, given that assertion-specific operators for all relevant formula operators have been provided, and that none of the cells providing input assertions have shared dependencies.*

Proof of Theorem 1: α-Propagation works by first replacing operators and operands with assertion-specific operators and assertions, respectively, and then evaluating the result with standard evaluation rules (eg: β-reduction in

lambda calculus), and finally merging sub-assertions with the merging algorithm. The proof, therefore, proceeds by first proving the correctness of the replacement step, and then proving the correctness of the evaluation step using the assertion-specific operators. The correctness of the merging algorithm is trivially proven by Figure 5.

Lemma 1 α*-Propagation correctly generates an assertion-specific formula by replacing formula operators and formula operands with assertion-specific operators and assertions.*

Proof: For the replacement step we define a correct assertion-specific formula to be one that generates the correct assertion for the given cell. Operator replacement is done in a one-to-one mapping from the standard language operators to their counterpart assertion-specific operators. Given that assertion-specific operators have been provided, operator replacement is obviously correct.

There are four situations to consider when performing operand replacement: If the operand is a sub-expression, it is handled recursively. If the operand is a cell reference, it is replaced with the referenced cell's assertion. Given that there are no shared dependencies, the assertion on each input cell precisely defines the set of values that the input cell can contribute to the formula. If the referenced cell does not have an explicit assertion, a *null* assertion is used. If the operand is a constant value, it is replaced with a degenerate range of $[k\ k]$ where k is the constant in question. For all $k \in \mathbb{R}$ this is obviously correct, since $\forall k, k \in [k\ k]$ and no other number has this property. ■

Regarding correctness of assertion-specific formula evaluation (the evaluation of a formula with assertion-specific operators), we provide the proof of correct evaluation of the assertion+ operator here, and refer the reader to the similar proofs of evaluation for other assertion-specific operators given by Summet and Burnett [44].

The assertion+ operator takes two assertions, A and B as input, and produces an output assertion O. The assertion+ operator is correct if, when applied to any arbitrary input assertions, the output assertion it produces is correct. Recall from Property 2 earlier in this section that an output assertion O is considered to be correct if it accepts every output value that can be produced by the cell's formula given inputs accepted by assertions on the cell's parents, and rejects all other values.

Lemma 2 *The assertion-specific formula evaluates correctly.*

Proof: There are two situations to consider. The first situation is that each input assertion contains only one sub-assertion. These assertions are propagated by the algorithm described by Figure 4, which is correct by definition of interval arithmetic [2, 29].

Next, consider the situation where one or more of the input assertions (A and B) contain multiple sub-assertions.

First we show that O accepts all values it should. Choose two numbers (c and d) where c is accepted by A and d is accepted by B. Because c is accepted by A, c is accepted by at least one sub-assertion of A, A_x. Likewise, d is accepted by at least one sub-assertion of B, B_y. The output assertion O contains a sub-assertion O_{xy} (OR'd with all

other sub-assertions of O) which was generated via the algorithm in Figure 4 using A_x and B_y, so O_{xy} accepts q where $q = c + d$.

Now we show that O rejects all other values. Assume that O accepts some value $r = c + d$ which cannot be generated by values that are both accepted by A and B. If c is accepted by A, then $d = r - c$. However, by interval arithmetic theory, d must then also be accepted and we reach a contradiction. If d is accepted by B, then $c = r - d$, and similarly by interval arithmetic theory c must be accepted and another contradiction is reached. If neither c or d are accepted, then one value (say c) must exceed the maximum range of A or B by some value k_1 while the other value (d) is below the minimum value of B or A by k_2 (respectively). In this case the larger of $|k_1|$ and $|k_2|$ can be added to d and subtracted from c, resulting in two new values c' and d' such that $c' + d' = r$ which is a contradiction, since c' is accepted by A, d' is accepted by B and r is generated by c' and d'. Since we know that at least one of the values used to generate r must be rejected by either A or B, then our assumption must be incorrect, and O rejects r. ■

By Lemma 1 the assertion-specific formula is created correctly (incorporating all parents, and therefore supporting Property 1) , and by Lemma 2 this formula generates a correct output assertion (as defined by Property 2) when evaluated. While the output assertion O may contain redundant or overlapping sub-assertions, the sub-assertion merging algorithm finds the minimum set of sub-assertions that accept the same set of values. Therefore given that assertion-specific operators for all relevant formula operators have been provided, and that none of the cells providing input assertions have shared dependencies, α-propagation correctly propagates assertions and Theorem 1 is true.

4.1.2 Complexity of α-Propagation

Property 3 (Responsiveness) states that dynamic assertions should not interfere with the property of immediate visual feedback that is characteristic of the spreadsheet paradigm. In this section we examine the complexity of α-propagation to determine if it fulfills Property 3.

α-Propagation's task is simply to evaluate the formula of a cell with assertion-specific operators. However, since assertion-specific operators are dealing with ranges rather than points, evaluation of a formula for assertion propagation may require more work than evaluation for a value.

The additional work performed by assertion-specific operators is dependent on the size of the input assertions; thus input assertions with many sub-assertions would greatly increase the time required to calculate an output assertion. However, in our experience, users rarely enter more than one sub-assertion on any given cell. Analysis of the assertions entered by users during a debugging task [51] showed that, of the 279 assertions they entered, only three contained more than one sub-assertion[5].

It is possible, however, that a small number of sub-assertions can generate large numbers of sub-assertions through propagation. Figure 6 gives an example of such a pathological situation. Propagation through a formula is done one

[5]The users went on to replace two of these three assertions with assertions containing only one sub-assertion within 30 seconds.

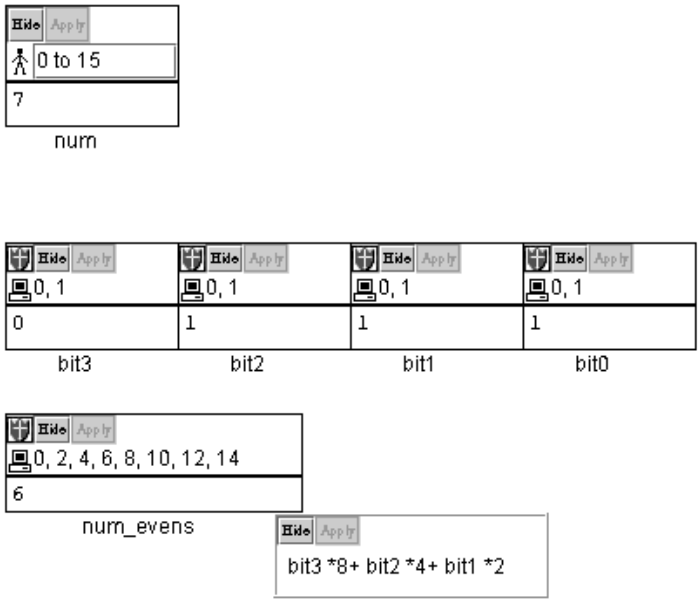

Figure 6: This spreadsheet rounds a value (`num`) down to the nearest even number by removing the least significant bit from the binary representation (`bit0`-`bit3`), and printing the result in decimal. This procedure causes an explosion of sub-assertions, as seen on the `num_evens` cell.

operator at a time, generating intermediate results that are potentially large. Since each operand may have up to A sub-assertions, the evaluation of the first assertion-specific operator in a formula may result in A^2 sub-assertions because every pair of sub-assertions must be evaluated with each assertion-specific operator in the assertion-specific formula, and there are A^2 pairs given A sub-assertions in each input assertion. The output of the first operation is used as the input to other operations in the formula (these are the intermediate results mentioned above), and therefore the second assertion-specific operator may generate up to A^3 sub-assertions. In the worst case, this may happen for every operator, generating A^F sub-assertions, where F is the number of operators in the largest formula in the spreadsheet. (This assumes that each operator is evaluated in constant time. Aggregate operators, such as *sum*, must be broken down into their component parts to determine F.) Asymptoticly, this results in $O(FA^F)$ time to evaluate a formula with F operators. But, as we have already explained, it seems reasonable to assume that $A = 1$ for input cells. Further, it turns out that assertions are rarely split by the propagation engine. Of the supported operators, only "if" and division (which splits assertions only when a divide-by-zero is possible) can split single sub-assertions into multiple sub-assertions. (The special requirements of "if" and division will be returned to several times in later sections.)

Under the assumptions that $A = 1$ and assertions are not split by the propagation engine, the time complexity of α-propagating assertions through a formula becomes $O(F)$. The system may also need to propagate through the entire spreadsheet, adding a factor of N to the complexity—where N is the number of cells in the spreadsheet—raising the complexity to $O(NFA^F)$, or $O(NF)$ under the assumption that $A = 1$. Assertions must then be merged with the sub-assertion merging algorithm, taking $O(A^F \log A^F)$ which is added to the worst case complexity. This does not change the asymptotic complexity however, because $O(A^F \log A^F)$ is equivalent to $O(FA^F)$ for $A > 1$, which is smaller than NFA^F. Under the assumption that $A = 1$ the merging step takes constant time.

The propagation of dynamic assertions occurs in two cases: Assertions are propagated when the user edits an assertion—in which case the operation takes $O(NFA^F)$ time as described above—and assertions are propagated when the user edits a formula of a cell with an assertion. In the second case, assertion propagation is done in parallel with standard spreadsheet evaluation (to generate new values based on a new formula). Without assertions, it takes $O(NF)$ time to populate a spreadsheet with new values (using the same interpretations of N and F as above). Thus, when $A = 1$, α-propagation adds only constant overhead to the work that is done already, and under these usage patterns, α-propagation maintains Property 3 (Responsiveness).

4.1.3 Limitations of α-Propagation

Earlier in this section we stated that α-propagation cannot handle shared dependencies. There are two interrelated reasons for this: α-propagation examines only one formula at a time—therefore preventing the detection of shared dependencies more than one cell up the chain of data flow—and cell references are treated as independent "interval constants" rather than interval variables. The problem is that shared dependencies need to be detected, as they require treatment as interval variables for correctness.

To see why this is true, consider the formula "$X - X$". The appropriate assertion for this formula is 0. This result can be obtained by treating X as an interval variable, which forces both instances of X to be bound to the same value for any one evaluation. Treating X as an interval constant, as α-propagation does, generates a much different result. For example, if X represents the range [0 1], the interval expression [0 1] - [0 1] would be evaluated, resulting in [-1 1] which is not correct. Interval variables would maintain the dependency between instances of a variable (in our case, a cell reference) explicitly [29]. Situations like "$X - X$" arise because of shared dependencies (called *diamonds* because the data flow arcs create a diamond-shaped structure), such as in the example given in Figure 7.

4.2 Propagation Through Shared Data-flow Dependencies

The complications caused by shared dependency diamonds stem from the way interval arithmetic treats intervals as constants, ignoring dependencies between multiple instances of variables. Overcoming these complications requires addressing both the problem of identifying the dependencies and the problem of maintaining the dependencies during the evaluation step. The existence of shared dependencies is the defining characteristic of the second variant of the propagation problem.

Identification of shared dependencies can be accomplished by locating (1) the diamond *head*[6]—the top-most cell in the diamond—and (2) the diamond *sources*—those cells that contribute to the diamond, but are not in the diamond themselves. The diamond heads and diamond sources together are termed the diamond *inputs*. Once the diamond inputs are identified, an expanded formula can be built that makes explicit all the shared dependencies. Section 5.1 discusses an algorithm to locate diamond heads and diamond sources, and then to use these identified cells to build an expanded formula.

Once the expanded formula has been constructed and all dependencies are explicit, the formula must be evaluated while maintaining the dependencies. This problem can also be stated as the problem of calculating the precise set of outputs for the expanded formula given the inputs specified by the assertions on diamond inputs. Specifically, given a function $f(x_0, x_1, \ldots, x_n)$ and f's domain, find the range of f (the output assertion). In the general case this is an unsolvable problem since it is not possible to deterministically locate all discontinuities in an arbitrary function.

In Section 5.1 we present an algorithm termed β-propagation, which is capable of propagating assertions through shared dependencies given a reduced set of formulas and a restriction on the data-flow graph complexity. We discuss the necessary restrictions below. In Section 6.2 we examine a set of real-world spreadsheets to determine the practical impact of the restrictions on the scope of our algorithm.

4.2.1 Required Formula Restrictions

By restricting the set of formulas supported by our approach to propagation it is possible to maintain Property 3 (Responsiveness) while calculating assertions with shared dependencies during typical usage patterns. (To maintain

[6] Note that it is possible for a cell to be affected by more than one diamond head.

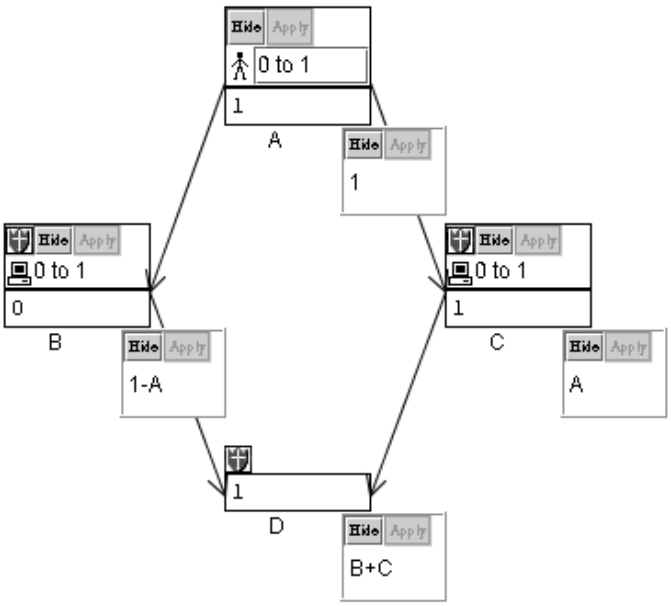

Figure 7: A simple “diamond”. Dataflow follows the arrows between cells. Assertions are shown above each cell, and formulas are in the boxes at the lower right of each cell. Cell `A` has an assertion which has been α-propagated to `B` and `C`; however the shared dependency prevents valid α-propagation to `D`.

consistency with related work in the field of mathematics, we use the terms "formula" and "function" interchangeably.) Restrictions are required not only for operators, but also for some operands.

Our approach relies heavily on the use of derivatives to propagate assertions through shared dependencies. The *critical points* (maxima, minima, and saddle points) of a function f can be calculated by using the first derivative test. (This is done by setting the derivative of f equal to 0 and solving the resulting equation.) The largest and smallest critical points define the endpoints of the assertion; however, derivative tests cannot be applied to arbitrarily complex functions. Since there are multiple ways functions can become complex, we address these restrictions individually.

Discontinuous functions: When calculating an assertion for a function it is important to remember that the exact range must be represented (Property 2, Correctness). However, it is not possible to find all discontinuities of an arbitrary function in general, even when given a set of inputs. From this we place our first restriction on the acceptable functions. Functions must be continuous and have a first derivative. (This set of functions is called c^1 [48].)

High-order polynomials: All polynomials are c^1 [45]. However the derivative of a polynomial must also be a function that can be solved algebraically when set equal to 0. Abel's impossibility theorem [1] states that this is only possible for polynomials of degree four or less. (The general quintic can be solved in terms of Jacobi Theta Functions [23, 49] but this method does not scale to higher degree polynomials either.) Because the first derivative of a polynomial of degree n results in another polynomial of degree $n-1$, functions must be of degree five or less.

Functions of multiple variables: As the number of independent parameters of a function (the number of unique cell references) increases, the derivative test used with one variable is no longer sufficient. Gradients can be used in place of derivatives[7] but this is still not adequate. The reason is that each input assertion introduces bounds on one dimension of the range of the given function. Thus, as the number of cell references increases, these bounds define a cube of interest that has dimensions equivalent to the dimensionality of the function (for example, a function containing four cell references will have sub-assertions that specify a hypercube of interest). The values inside this cube can be generated by the function given values accepted by the inputs. The challenge then is to find the maximum and minimum values within this cube, or on the cube's border. The bounds of this cube, however, increase exponentially with the number of cell references and all bounds would need to be checked explicitly.

In addition, there is another complication with multivariate functions. Recall from Section 4.1 that all pairs of sub-assertions on the inputs were evaluated with the assertion-specific operators. In α-propagation, the cardinality of the assertion-specific operators was low. However, since β-propagation must maintain the dependencies

[7] Gradients are required because derivatives are only applicable to functions of one variable. Gradients provide a multi-dimensional derivative in the form of a vector, with each element in the vector equal to the partial derivative with respect to one variable.

between interval variables in the expanded formula, the number of input cells considered simultaneously is bounded only by the number of cells in the spreadsheet. Given N input cells, and A sub-assertions on each cell, the expanded formula would have to be evaluated A^N times.

The solution to this problem is to use divide and conquer. As we indicated above, it is viable to propagate dynamic assertions through functions of degree five or less. This capability can be leveraged to propagate dynamic assertions through some multivariate functions. Since polynomials consist of addition and multiplication, both of which are commutative, it is possible to divide multivariate functions into single variable functions that are combined with addition and multiplication. However, in order to maintain the dependencies between instances of the same variable, propagation is possible only when the expanded formula can be divided into single-variable polynomials such that no cell reference occurs in more than one polynomial. The expanded formula can be split in such a way if no more than one variable occurs in more than one term with other variables. With this restriction, each variable can be treated individually and the combinatorial explosion of sub-assertions mentioned above is avoided.

The intuition behind this restriction is that any function that can be divided in this way can be represented by a spreadsheet in which no shared dependency has more than one diamond input (the diamond head). Section 5.1 describes in detail an approach, which we term β-*propagation*, to dynamic assertion propagation using divide and conquer.

Our prototype of dynamic assertions, implemented in the Forms/3 spreadsheet environment, supports propagation through the following operations: $+, -, *, <, \geq, >, \leq, =, if$ (in Section 4.3 we examine certain instances of if that cannot be propagated through) and $/$ (for α-propagation).

4.3 Propagation Through Control-flow Dependencies

In addition to the complications presented by pure data-flow constructs, control-flow can create a new source of dependencies. This situation is still caused by a diamond of sorts, but the diamond is not caused entirely by data flow (although it appears this way when viewed at the granularity of cells). An example of this is shown in Figure 8.

In Figure 8 the formula for cell `B` has two basic blocks[8] and a predicate. Figure 9 shows the control-flow graph for cell `B`. Notice that the control-flow arcs in cell `B` complete a diamond with the data-flow arcs from cell `A`. Propagating through diamonds of this sort is the third variant of the propagation problem, which we term the *if-problem*.

By virtue of control-flow, the "then" and "else" nodes are executed only for a specific set of inputs to cell `B`. Specifically, the "then" node is executed precisely when the inputs to `B` cause the predicate to evaluate to true, and the "else" node executes precisely when the predicate evaluates to false. Given a formula of the form:

[8] A basic block is a portion of a program that is run in its entirety every time it is executed. Formally, a basic block is a maximal single-entry, single exit sequence of atomic statements.

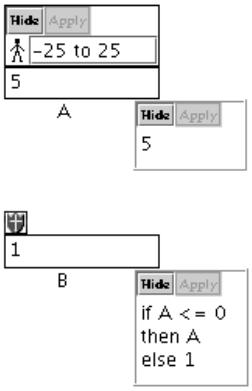

Figure 8: A simple spreadsheet demonstrating the if-problem.

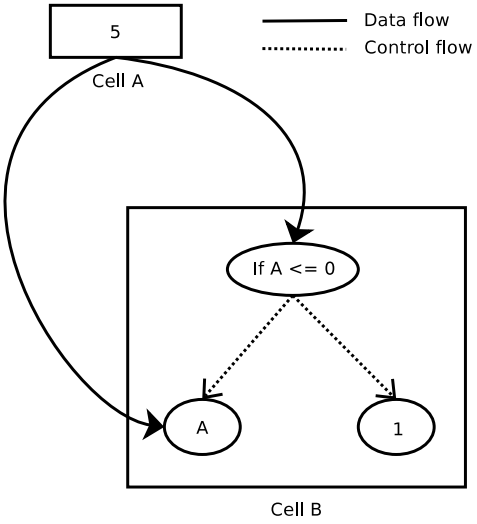

Figure 9: The control flow arc from the predicate to the "then" node completes a (3-arc) diamond with the two data-flow arcs into cell B.

if *pred* **then**
  then_expr
else
  else_expr
end if

we define the following terms:

- cellRefs(*pred*) is the set of all cells referenced by *pred*
- cellRefs(*then_expr*) is the set of all cells referenced by *then_expr*
- cellRefs(*else_expr*) is the set of all cells referenced by *else_expr*
- sat(*refs*,*pred*) is the set of assignments to cell references in *refs* that can cause *pred* to evaluate to true.
- fail(*refs*,*pred*) is the set of assignments to cell references in *refs* that can cause *pred* to evaluate to false.

The if-problem occurs when the following condition is true:

$$\text{cellRefs}(\textit{pred}) \cap [\text{cellRefs}(\textit{then\_expr}) \cup \text{cellRefs}(\textit{else\_expr})] \neq \emptyset. \quad (1)$$

When an if expression does not have shared dependencies between the predicate and either the "then" or "else" expressions, the if-problem does not occur. In this case, sub-expression assertions are propagated with the assertion-if operator (using α-propagation or β-propagation as appropriate), which first propagates a boolean assertion[9] to the predicate, and then calculates the correct assertion for the if based on the possible truth values of the predicate. The assertion-if operator returns the assertion on the "then" expression if the predicate is asserted to be true and returns the assertion on the "else" expression if the predicate is asserted to be false. If the input to the predicate can be either true or false, the sub-expression assertions on the "then" and "else" expressions are unioned and returned.

A precise definition of the if-problem follows from Equation 1 above and from the fact that an assertion on a cell with an "if" is simply the union of the assertions on the "then" and "else" clauses of the formula. The if-problem is then defined as the problem of calculating precise sub-expression assertions for the "then" and "else" clauses, given the subset of variable assignments that can cause the predicate to evaluate to true (for the "then" clause) or false (for the "else" clause). In general, these subsets are:

$$\text{sat}(\text{cellRefs}(\textit{pred}) \cap \text{cellRefs}(\textit{then\_expr}), \textit{pred})$$

for the *then_expr*, and

$$\text{fail}(\text{cellRefs}(\textit{pred}) \cap \text{cellRefs}(\textit{else\_expr}), \textit{pred})$$

for the *else_expr*.

[9] Propagation of boolean assertions follows from the propagation of ranges, and is discussed in detail by the previous work [44].

If the sat and fail sets can be found, they can then be used to generate sub-expression assertions for the *then_expr* and *else_expr* as follows. If either expression is a constant, then the sub-expression assertion is simply that constant. Since all possible paths of execution must pass through one of these two expressions, the sub-expression assertions can be unioned to give a cell-level assertion. Given a nested "if", this procedure is repeated recursively.

In the case where there is no intersection of variables between the predicate and either the *then_expr* or *else_expr* the respective sub-expression assertion is calculated with α-propagation or β-propagation, as needed. By the definition of the if-problem, at least one variable in the predicate is also used in either the *then_expr* or *else_expr*.

4.3.1 Classification of the If-problem

The general case of the if-problem, with no restrictions on the predicate complexity is NP-Hard. This is shown through a Turing reduction from Satisfiability, which is known to be NP-Complete. A Turing reduction proves NP-Hardness by demonstrating that all instances of a known problem (Satisfiability) can be converted into an instance of the unknown problem (the if-problem) through a polynomial number of constant-time steps[10].

The Satisfiability problem is defined as follows:

> Given a boolean expression *S* over *N* variables, is there an assignment of truth values to each of the *N* variables such that *S* evaluates to true?

Any instance of Satisfiability can be transformed into an instance of the if-problem in linear time through the following five steps:

1. Create a cell `C` with an if expression.
2. Set the expression *S* to be the predicate.
3. Set the "then" clause to be True.
4. Set the "else" clause to be False.
5. Place an assertion of "True, False" on each variable.

Steps 1, 3 and 4 all take constant time. Steps 2 and 5 each take time linear in the number of variables.

If *S* is unsatisfiable, the assertion on `C` will be "False", since only the "else" clause is reached. If *S* is a tautology, the assertion on `C` will be "True" since only the "then" clause is reached. Otherwise the assertion on `C` will be "True, False", indicating that both branches of the if can be exercised. Thus, any instance of Satisfiability can be transformed into an instance of the if-problem in linear time. Therefore the if-problem is at least as hard as Satisfiability.

[10] NP-Hard is a more general set of problems than NP-Complete, and therefore a Turing reduction does not require the additional step of proving membership in NP[19].

Note that, since the precision of values in the if-problem is theoretically unbounded (variables in the if-problem can all be mapped to the Reals), it may not be possible to solve any instance of the if-problem by solving an instance of Satisfiability.

Section 6.2 considers the occurrence of the if-problem in a corpus of real-world spreadsheets. The empirical evidence from this corpus shows that the if-problem is rare in real-world spreadsheets. Because of this evidence, we have decided not to support assertion propagation through instances of the if-problem; instead a *null* assertion is generated. This decision violates Property 1 (Reliability, from Section 4) because of the inherent difficulty of the if-problem. It may be possible to restrict the set of predicates such that assertions can be propagated through some instances of the if-problem efficiently, but this would violate Property 1 as well. Another trade off would be to use an approach such as Ernst's statistical methods of invariant detection [16], which may support Property 1, but such methods would most likely violate Property 2 (Correctness). Our decision to not support propagation through instances of the if-problem maintains Properties 2 and 3 (Responsiveness).

5 Solving Shared Data-flow Dependencies: β-Propagation

In Section 4.2 we discussed complications with propagation of assertions through certain spreadsheets. Despite these complications with general spreadsheets, it is still viable to propagate assertions if certain restrictions are placed on the spreadsheets supported. Section 4.2.1 presented a set of restrictions on the functions our approach is able to propagate assertions through, given spreadsheets with shared data-flow dependencies. In this section we present algorithms for dynamic assertion propagation through spreadsheets that obey these restrictions.

5.1 β-Propagation: Propagating Through Low-Degree, c^1 Spreadsheets

β-Propagation is a generalization of α-propagation[11] that provides a solution to α-propagation's issues with shared dependency diamonds by maximizing and minimizing formulas that represent spreadsheet calculations. Because of the inherent difficulties of calculating minimum and maximum values of complex functions, β-propagation can be applied only to the portions of spreadsheets that obey the restrictions stated in Section 4.2.

Algorithm 2 gives an overview of the dynamic assertion propagation algorithm with both α-propagation and β-propagation. Algorithm 2 is applied to one cell at a time, propagating an assertion to each cell in turn, if possible. The first step (line 1 in Algorithm 2) determines if β-propagation is needed in order to propagate an assertion to a cell *D*. To determine this, *D*'s formula is checked for shared dependencies. This check is made by performing a breadth-first search of *D*'s backward slice and flagging visited cells. Specifically, the approach detects shared dependencies by checking the flag on each cell before traversing that cell. If the flag is set, the cell represents a shared dependency and

[11] β-Propagation is triggered by the same events as α-propagation. Depending on the presence of shared dependencies, the correct algorithm is applied.

Algorithm 2 Overview of the dynamic assertion propagation algorithm with α-propagation and β-propagation.

Input: Cell D, Spreadsheet S

```
1:  if not(detect_diamonds(D,S,∅)) then
2:     Perform α-propagation (Algorithm 1)
3:  else
4:     ;; Perform β-propagation:
5:     Find the diamond inputs.
6:     Expand D's formula.
7:     if D's formula violates any restrictions from Section 4.2.1 then
8:        return null
9:     end if
10:    Build sub-assertions from critical points.
11:    Merge sub-assertions.
12: end if
```

the search halts. Algorithm 3 (*detect_diamonds*) performs this check, returning the first shared dependency (diamond head) found. For example, if Algorithm 3 were applied to cell D in the spreadsheet in Figure 7, the algorithm would return cell A. If a diamond head is found, β-propagation is required. Once shared dependencies have been detected, all the dependencies must be found.

Without keeping track of which diamond heads have been found, β-propagation would not halt—rather, the same diamond head would be located at each iteration (Algorithm 4). This is dealt with in Algorithm 3 by ignoring certain edges in the data-flow graph because they have already been determined to be within diamonds. The *untraversableEdges* set defines these edges which are not to be traversed. *UntraversableEdges* is initially empty (line 1 of Algorithm 2 and line 3 of Algorithm 4), and the full graph is traversable. As β-propagation locates diamond heads, *untraversableEdges* is maintained by adding to *untraversableEdges* all but one edge from each diamond head to flagged children of that diamond head.

In order to perform β-propagation, all of D's shared dependencies must be made explicit in D's formula. First, our approach identifies all the ancestors of D that participate in diamonds for which D is a sink. Algorithm 4 shows one way of doing this. Once the cells that participate in these diamonds have been found, our approach uses set difference to find the diamond sources (cells that are parents of cells participating in a diamond, but that are not in diamonds themselves).

$$\textit{diamond sources} = \text{parents}(\textit{diamonds}) - \textit{diamonds}$$

The set of diamond sources is combined with the diamond head(s) to yield the set of *diamond inputs* (line 5 of Algorithm 2).

Once the diamond inputs have been identified, a recursive formula expansion algorithm is applied to D's formula (line 6 of Algorithm 2). Recursive formula expansion recursively replaces each cell reference with that cell's formula, enclosed in a construct from the host language that maintains any implicit order of evaluation created by data flow. In the language of arithmetic, parentheses are used for this. This expansion repeats recursively until each diamond input

Algorithm 3 A function used to locate shared dependencies with breadth-first search and flagging. *parents(x,untraversableEdges)* returns only the parents of x accessible through traversable edges.

Require: Cell D, Spreadsheet S, Set untraversableEdges

```
1:  function detect_diamonds(D, S, untraversableEdges)
2:      head ⇐ null
3:      UpFrontier ⇐ parents(D, untraversableEdges)
4:      while UpFrontier not empty do
5:          if c, flag = True then
6:              head ⇐ c
7:              return head
8:          else
9:              c, flag ⇐ True
10:             UpFrontier ⇐ UpFrontier ∪ parents(c, untraversableEdges)
11:             Remove c from UpFrontier
12:         end if
13:     end while
14:     return null
15: end function
```

Algorithm 4 This procedure finds all cells between a given cell D and its shared dependencies.

Require: Cell D, Spreadsheet S

```
1:  diamondHeads ⇐ ∅
2:  untraversableEdges ⇐ ∅                    ;; The set of edges not to be traversed by parents
3:  head ⇐ detect_diamonds(D,S,untraversableEdges)
4:  downstreamCells ⇐ ∅
5:  while head ≠ null do
6:      diamondHeads ⇐ diamondHeads ∪ {head}
7:      Add all but one edge from head to flagged children of head to untraversableEdges
8:      Set all flags to False
9:      head ⇐ detect_diamonds(D,S,untraversableEdges)
10: end while
11: for all c ∈ diamondHeads do
12:     downstreamCells ⇐ downstreamCells ∪ forwardSlice(c)
13: end for
14: return backwardSlice(D) ∩ downstreamCells
```

is reached[12]. At this point the expansion stops, and the expanded formula is returned. The expanded formula represents all the shared dependencies together in one formula, so that they can effectively be treated as interval variables. Recall from Section 4.1.3 that input assertions must be treated as interval variables for correctness.

For example, the formula for the cell D, $B+C$ in Figure 7 is expanded to:

$$(1-A)+A$$

The expanded formula is now checked against the restrictions stated in Section 4.2.1 (line 7 of Algorithm 2). This is done by scanning the expanded function for operators and operands that are not allowed. The restrictions on functions of multiple variables are checked by converting the expanded formula into a canonical representation [13]. This canonical representation is then searched by checking for non-linear terms, and if any are found, checking again for terms with multiple variables. If any of the restrictions in Section 4.2.1 are violated, a *null* assertion is generated for the diamond sink.

First derivative tests are used to find maximum and minimum values. Due to the restrictions from Section 4.2.1, all non-linear expanded functions can be broken into polynomials of one variable, and these polynomials are of degree five or less. For example, Figure 10 shows a spreadsheet with the expanded formula X^2-X-Y^2, which meets the restrictions. The canonical representation stores polynomials as a main polynomial over one variable which is added and multiplied with polynomials of other variables. If no polynomials in the canonical form have the same main variable, then the restriction is satisfied and each polynomial can then be treated independently. This is the divide step of the divide and conquer algorithm. Continuing with the example from Figure 10, our approach breaks the expanded formula into X^2-X and $-Y^2$. (The canonical representation of X^2-X-Y^2 is $<X,1,-1,<Y,-1,0,0>>$ where the first element of the vector is the variable for that polynomial, and the other entries are the coefficients. As this example shows, polynomials over additional variables are stored as coefficients of the constant term. This format is explained in detail by Norvig [32].)

First derivative tests are applied to these polynomials to find critical points (line 10 of Algorithm 2). The boundary conditions specified by the sub-assertion on the relevant variable are then added to the set of critical points. (Recall that each variable is a cell reference.) The critical points within the boundary conditions are then evaluated, along with the boundary conditions, generating a set of values. The largest and smallest of these values are used to generate a sub-assertion for the relevant part of the expanded formula. Since the dependencies in the expanded formula have now been dealt with, the resulting intervals can be treated as interval constants. These interval constants are combined with standard interval arithmetic to generate a set of output sub-assertions.

These output sub-assertions are finally merged (line 11 of Algorithm 2) using the sub-assertion merging algorithm discussed in Section 4.1.

[12] Because it is possible for an edge to be traversed multiple times, due to nested data-flow diamonds, recursive formula expansion is implemented with dynamic programming.

[13] This is possible because the expanded formula is a polynomial at this point [32].

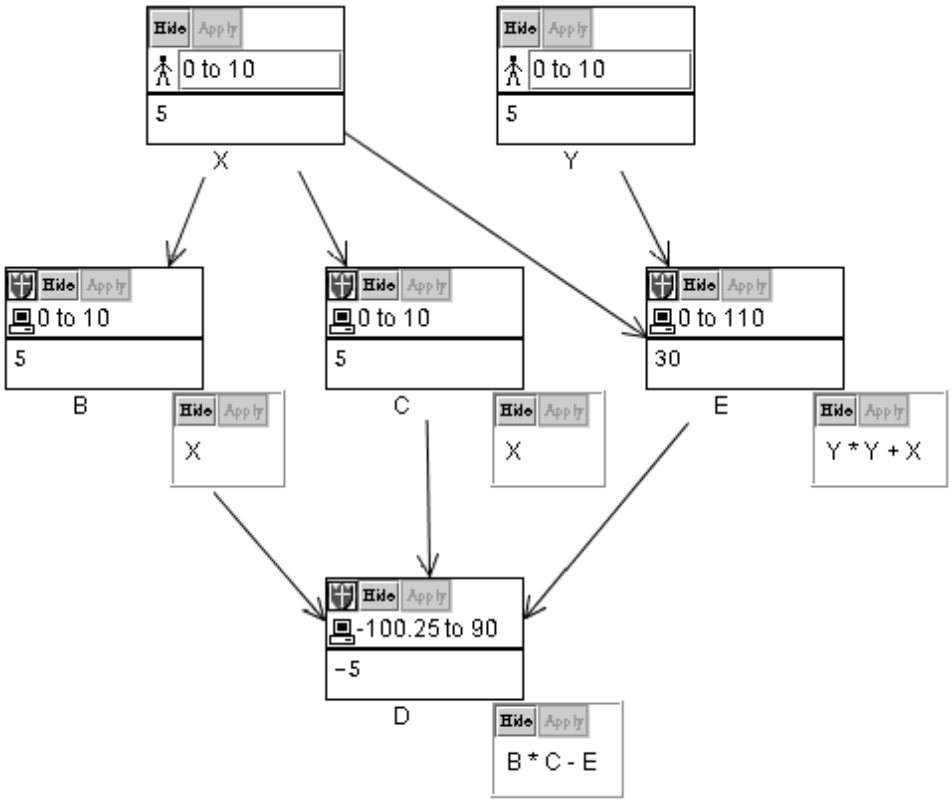

Figure 10: Multiple data-flow diamonds end in Cell D, but propagation is still possible because all terms in the expanded formula ($X^2 - X - Y^2$) depend on at most one variable.

Returning to our example from Figure 7, the expanded formula $((1-A)+A)$ is evaluated with $A=0$ (since this is a constant function, 0 is chosen arbitrarily), resulting in 1. Since only one critical point was found (and therefore only one value is returned) the output assertion is the degenerate range [1 1]. In the example from Figure 10, the interval constant [-0.25 90] is generated for the polynomial $X^2 - X$, from the critical point (0.5) and the boundary point(10). The interval constant [-100 0] is then calculated for the polynomial $-Y^2$. These constants are added together with the assertion+ operator, yielding the final sub-assertion [-100.25 90] for cell D.

5.1.1 Correctness of β-Propagation

β-Propagation is applied to data-flow diamonds that do not involve instances of the if-problem. Such diamonds are termed *pure* data-flow diamonds.

Theorem 2 β-*Propagation correctly propagates assertions through all pure data-flow diamonds*.

Proof of Theorem 2: Once the shared dependencies have been found, β-propagation generates assertions through two steps: a formula expansion step, which generates a formula with explicit dependencies; and a calculation step, which maintains these dependencies when calculating a final assertion. The proof of correctness reflects these steps by proving the correctness (as defined in Section 4, Property 2) of each step as a lemma.

Lemma 3 *Recursive formula expansion creates a formula that generates the same value as the original set of cells traversed with recursive formula expansion*.

Proof: The proof of Lemma 3 follows directly from the definition of formula expansion. A formula is expanded by replacing each instance of a cell reference with that cell's formula, enclosed in parentheses (or other grouping construct). This process is repeated as needed. Grouping constructs, such as parenthesis, preserve the order of evaluation; hence formula expansion is simply the replacement of left-hand sides (cell references) with their right-handed sides (those cells' formulas). ■

Lemma 4 *The calculation step of* β*-propagation generates correct assertions with first derivative tests*.

Proof: The expanded formula is broken into polynomials of one variable, and first derivative tests are applied to each of these polynomials to find critical points. By definition, all critical points of a function are found by first derivative tests, but the bounds on the inputs might not fall on one of these critical points. However, recall that the bounds from the input assertions are added to the set of critical points, resolving this discrepancy. By definition, all maximum and minimum points of a continuous function must be on boundaries or at critical points. Thus, correct sub-assertions are generated for each polynomial of one variable. These assertions are then combined with interval arithmetic using the assertion-specific operators as described in Section 4.1.

Finally, the resulting output sub-assertions are merged with the sub-assertion merging algorithm described in Section 4.1. Because derivative tests, the assertion-specific operators involved (for example, assertion+), and the sub-assertion merging algorithm have all been proven to be correct, the generated output assertion is also correct. ■

5.1.2 Complexity of β-Propagation

In this section we examine the time complexity of β-propagation. Based on this analysis we are able to evaluate β-propagation with regard to Property 3 (Responsiveness) from Section 4.

β-Propagation consists of six primary steps:

1. First, breadth-first search is used to locate the diamond head(s).
2. Second, the cells within the diamond are identified by traversing the data-flow graph from the diamond heads to the diamond sink.
3. Third, the expanded formula is created with recursive formula expansion.
4. Fourth, the expanded formulas are checked against the restrictions in Section 4.2.1.
5. Fifth, output sub-assertions are created by finding and comparing critical points.
6. Finally, the output sub-assertions are merged with the sub-assertion merging algorithm.

The complexity of each of these steps is addressed individually using the notation of Table 1. β-Propagation is triggered by the same user actions as α-propagation: either a formula or assertion edit by the user. Since an edit may cause the entire spreadsheet to be updated in the worst case, it is possible that every cell may need an assertion propagated to it with β-propagation. For simplicity, we examine the complexity of each of the steps of β-propagation for one cell, then add a factor of N to accommodate the case where every cell on the spreadsheet must be updated, which is possible in the worst case.

For the first step, location of the diamond heads, repeated breadth-first search with flags is used. Each search returns one diamond head. In the worst case, there may be N diamond heads, and each instance of breadth-first search may have to traverse the entire spreadsheet, taking $O(N+E)$ time for each search. Therefore this step takes $O(N^2+NE)$ time in the worst case.

For the second step, identifying the cells in the diamond, the graph is traversed from each diamond head to the diamond sink. In the worst case, there are N diamond heads and there may be N cells and E edges between each diamond head and the diamond sink. Since this traversal is linear in the number of edges (in the worst case) this step takes $O(E)$ time for each diamond head, and $O(NE)$ time for all heads.

Variable	Description
N	The number of cells in the spreadsheet.
E	The number of edges in the spreadsheet, each edge representing a reference to a cell in another cell's formula.
R	The number of references in the expanded formula.
A	The greatest number of sub-assertions on any cell in the spreadsheet.
X	A convenience variable. By definition $X > N, X > E, X > R$ and $X > A$.

Table 1: The notation used in the discussion of the complexity of β-propagation. E is represented separately because it is not bounded by N^2 since there can be more than one edge between any two cells.

For the third step, creating the expanded formula, recursive formula expansion traverses every edge in the diamond. This procedure recursively calls itself each time an edge is traversed. Recursive formula expansion performs one operation at each recursive step: The formula of the current cell is copied into the expanded formula. Assuming a constant bound on the spreadsheet formula length, this single copy operation takes $O(1)$ time[14]. Since recursive formula expansion is implemented with dynamic programming, each edge is only traversed once and therefore the copy operation happens only once for each edge. In the worst case recursive formula expansion traverses the entire spreadsheet (because the diamond may span the entire spreadsheet), taking $O(1)$ time at each of E edges, resulting in an asymptotic time complexity of $O(E)$.

As the fourth step, the expanded formula is now checked against the restrictions in Section 4.2.1. The expanded formula is searched for operators that are not allowed in polynomials and for exponents greater than five. This requires a pass through the expanded formula, taking $O(R)$ time. Recall from Table 1 that R represents the number of cell references in the expanded formula. A second pass puts the formula in canonical form. In the worst case, conversion to canonical form may take $O(2^R)$ time, since the expanded formula may be a product of R polynomials, each of which is a sum of terms. This case, however, is filtered in advance because it always generates a polynomial which cannot be split, and therefore cannot be propagated through. Given this, conversion to canonical form takes $O(R \log R)$ time, since it may require polynomial multiplication to obtain the simplest (canonical) form [32]. (This is supported by the empirical evaluation of spreadsheets in Section 6.1.) A final $O(R)$ pass ensures that the formula can be split into sub-polynomials of one variable.

In step five, the expanded formula is divided into sub-polynomials in $O(R)$ time. This generates (at most) R polynomials of constant size. Each of these sub-polynomials is differentiated in constant time, taking a total of $O(R)$ steps. The resulting derivatives are solved to find critical points, taking constant time for each sub-polynomial [5, 17], and $O(R)$ steps for all sub-polynomials. Evaluating all critical points for R polynomials takes $O(R)$ time, since each poly-

[14]For example, in Excel the maximum formula length is 1024 characters.

nomial has a constant number of operators and only a constant number of critical points can exist for polynomials with constant bounded degree. In total, the asymptotic time for this step is $O(R)$.

For each of A sub-assertions on the diamond inputs, the only sub-polynomial depending on that input is evaluated at the end points of the sub-assertion. The resulting points are compared with the set of values from critical points to find sub-assertions for the sub-polynomials. This takes A time to evaluate one sub-polynomial, and A time to find the new boundary points. This is repeated for each of the R sub-polynomials, resulting in $O(AR)$ time. Notice that by separating the diamond inputs (in step five), this approach is able to avoid the combinatorial explosion discussed in Section 4.2.1. Since the sub-polynomials are only dependent on one variable, and they are continuous, this step generates at most A sub-assertions.

Finally, step six combines these resulting intervals with assertion-specific operators, which take $O(A^2R)$ time given A sub-assertions on each of R operands. As these operators are evaluated, the sub-assertion merging algorithm from Section 4.1 is applied to the resulting output sub-assertions. The merging algorithm takes $O(A^2 \log A^2)$ time, since the output sub-assertions may need to be sorted. It is possible, however to concoct a situation in which each combination step generates A^2 assertions that cannot be merged, and that are then combined with the next operand to generate A^3 output sub-assertions that cannot be merged, and so on, generating A^R output sub-assertions in total.

In total, the time required for β-propagation is dominated by the combination and merging of sub-assertions in step six, which takes $O(A^R)$ time in the worst case. The exponential complexity of this step is due to the same reasons discussed in Section 4.1.2. Similarly, we anticipate that it is safe to assume that in most cases $A = 1$. As the number of generated sub-assertions increases, it becomes statistically less likely that they will not overlap, in which case they will be combined by the sub-assertion merging algorithm. In addition, in our empirical work, users rarely entered multiple sub-assertions on a cell, further reducing the chances of this combinatorial explosion[15]. Section 6.1 discusses a study in which users entered assertions that support this assumption. Under this assumption, the time required for β-propagation is dominated by steps one, locating the diamond heads $O(N^2 + NE)$; and four, checking the formula restrictions $O(R \log(R))$. These steps are additive, resulting in $O(N^2 + NE + R \log(R))$. Recall that this time complexity represents the time required to propagate an assertion onto one cell, it may be necessary to perform these calculations for every cell on the spreadsheet, adding a factor of N. Adding this factor, and noting that R is bounded by E (because each edge represents a cell reference), brings the complexity to $O(N^3 + N^2E + NE \log E)$. By introducing X, a variable larger than N, E, R and A, this complexity can be re-written as $O(X^3 + X^3 + X^2 \log X)$, which is dominated by $O(X^3)$.

Under the simplifying assumptions that the number of sub-assertions stays small and users enter continuous assertions, β-propagation adds time linear in the size of X to the work already done by standard spreadsheet evaluation mechanisms. Therefore β-propagation is a viable algorithm for assertion propagation in spreadsheets with shared data-flow diamonds, and in particular β-propagation maintains Property 1 (Reliability) and Property 3 (Responsiveness).

[15] Of the functions our approach supports, only if-expressions can split individual sub-assertions.

6 Dynamic Assertions: Evaluations and Applicability

6.1 Summary of Empirical Evaluations with Users

Although the focus of this paper is propagation, the motivation for propagation is usefulness of the propagated assertions (Property 4 in Section 3). Therefore, in this section we briefly summarize four empirical studies from our previous work that address the usefulness of dynamic assertions in end-user programming environments from the perspectives of the users themselves.

The initial think-aloud study alluded to in Section 3.2 provided insights into five users' abilities to reason about and use assertions. Results and observations from this initial study led to the following questions:

1. *Will users with assertions be more effective at debugging than users without assertions?*

2. *Will users understand assertions?*

3. *Will assertions help users judge the correctness of their spreadsheets?*

These questions were then addressed by a controlled experiment with 59 participants (end users with little or no programming experience) [11]. These users performed two debugging tasks. The participants were split into two groups: a control group without access to dynamic assertions, and a treatment group with specific dynamic assertions available, if they chose to display them. (Participants were not able to enter assertions other than those provided.) This study revealed that the treatment participants (participants with assertions) both identified and corrected significantly more faults than the control group, answering Question 1. Questions 2 and 3 were addressed by a post-session questionnaire. This questionnaire revealed that users understood assertions and that users with assertions were able to more accurately judge the correctness of their spreadsheets than users without assertions.

Questions 1, 2 and 3 demonstrated the usefulness of dynamic assertions once they have already somehow been entered. However the benefits exhibited by the studies above can be achieved only if users actually enter assertions, from which more can be propagated. Since the use of assertions is not ubiquitous among professional programmers, it seemed likely that a similar reluctance may exist with end-user programmers. To address this issue, we developed a strategy called Surprise-Explain-Reward [51]. This strategy leverages the user's curiosity (triggered by a surprise) to educate and entice him or her (through an unintrusive negotiated explanation system) into using dynamic assertions. Once assertions are present, various rewards such as violation ovals and testing assistance improvements continue to provide rewards for entering more dynamic assertions. In our prototype, surprises are generated by Help-Me-Test, an automatic test-case generation tool that may be invoked by the user [18]. These surprises take the form of "guessed" HMT-assertions that are based on the tool's attempts to find suitable test values for each cell. By exploring the HMT-assertions with the mouse, users discover—through tool tips—that they can enter better assertions and reap rewards.

Wilson et al. performed an experiment to test the effectiveness of Surprise-Explain-Reward with dynamic assertions [51] and to answer the following questions related to assertions. (Other questions regarding Surprise-Explain-Reward were also addressed.)

4. Will users enter assertions?

5. Do users understand assertions without any training?

6. Will user-entered assertions be accurate?

In this study 16 participants (business students) conducted the same debugging tasks as in [11]. In contrast to the earlier assertions studies, in this study the participants were not trained to use assertions. In fact, assertions were never mentioned by the experimenters. Participants initially did not use assertions (on average, they entered their first assertion 14 minutes into the first task); however once they did use assertions, they continued to use them. Every participant who entered an assertion entered more assertions later. During this experiment the participants entered 279 assertions, of which 95% were correct. By "correct" we mean that the assertions were identical to the assertions created by the design team for the previous experiment. This indicates that, given suitable support, not only will users enter assertions without any previous knowledge of the feature, users are also capable of creating accurate assertions on their own (answering Questions 4 and 6). Through questionnaire-based comprehension testing we were also able to determine that the participants were able to understand assertions based entirely on the explanations provided by the system and their experimentation (answering Question 5). The empirical results of Questions 1-6 provide encouraging evidence that dynamic assertions satisfy the Usefulness property (Property 4).

6.2 Spreadsheet Evaluations

Sections 4.2 and 4.3 describe situations in which deductive propagation of dynamic assertions is not viable. Recall, however, that Property 1 from Section 4 requires that given a cell `C`, an assertion for `C` can be computed if all `C`'s ancestors have assertions. The inability to propagate dynamic assertions in all situations violates this property. This raises the following questions:

1. How often do the situations arise in which it is not viable to propagate dynamic assertions?
2. When these situations arise, how do they arise, and how might they be addressed?

To answer these questions we conducted an empirical examination of 40 real-world spreadsheets. This corpus of spreadsheets was collected from spreadsheets used for real-world tasks that were either donated to us or were obtained from Google search results for the terms "database", "grades" and "financial". Of the spreadsheets obtained, we considered only macro-free spreadsheets with at least one formula involving one or more cell references. The spreadsheets ranged in size from 15 to 7,773 cells.

	Spreadsheets affected	Total instances
Data-flow diamonds	13	311
if-problems	2	25

Table 2: Data-flow diamonds and instances of the if-problem were found in 15 of the 40 spreadsheets in our corpus.

6.2.1 How often do these situations occur?

Each spreadsheet was examined by instrumenting our prototype to record all diamonds found, and then applying our prototype to each spreadsheet as if every input cell had an assertion. As Table 2 shows, of the 40 spreadsheets examined, 15 contained either data-flow or control-flow diamonds, creating 336 diamond sinks. The remaining 25 spreadsheets did not contain diamonds, and could be processed with α-propagation.

6.2.2 When do the situations occur, and how might they be addressed?

In the 15 spreadsheets with diamonds, 336 diamonds were detected, 25 of which were instances of the if-problem. Of the remaining 311 diamonds, 244 (78.5%) were represented by expanded formulas that violated one of the restrictions in Section 4.2.1. (Our prototype stops processing the expanded formula as soon as a violation is found.) Table 3 presents the violations detected. Examination of this table reveals that, aside from six instances of log, the only restrictions that had an effect on β-propagation's ability to propagate dynamic assertions were the restrictions on discontinuous functions.

7 Open Problems

The examination of real-world spreadsheets revealed four common functions that were responsible for all except two of the propagation problems encountered (Table 3). The number of spreadsheets that our approach to dynamic assertions supports would greatly increase to 38 out of 40 if these functions were supported. The functions identified by our experiment were *log*, *division*, *countif* and *round*. This section considers the open problem of propagation through the function classes represented by these four functions. β-Propagation provides the foundation for managing shared dependencies upon which potential solutions can built.

One spreadsheet in our corpus contained six diamonds involving log. Although log is not currently supported because it is not a polynomial, this function could be easily supported because it is a unary function (therefore shared dependencies are not an issue) and it can be differentiated. Other functions with similar properties (such as periodic functions) could also be handled in a similar manner. We propose that it is viable to remove the restriction that a function must be polynomial if it is unary and differentiable.

Shared dependencies with division appeared in five spreadsheets, but division violates the restriction on discon-

Invalid formulas	Spreadsheets in violation	Diamonds in violation
High degree polynomials	0	0
Indivisible multivariate polynomials	0	0
Unsupported functions:		
log	1	6
Discontinuous functions:		
countif	1	3
/ (division)	5	92
round	3	119
Sub Total:	10	220
if-problem	2	22
Total affected:	12	244
Percentage of all samples:	30% of all spreadsheets	78.5% of all diamonds

Table 3: The number of spreadsheets containing situations through which our approach to dynamic assertions cannot propagate, grouped by the reasons. A total number of 40 spreadsheets involving 336 diamonds were examined.

tinuous functions. It may be possible to support division, however, because it is only discontinuous at one point. (α-Propagation takes advantage of this property to propagate dynamic assertions through division when shared dependencies are not present.) Division with shared dependencies could be approached separately with two special cases: in the case in which shared dependencies only occur in either the expanded formula for the dividend or divisor (but no dependencies are shared by both), β-propagation could be applied to the operand(s) with shared dependencies to generate intermediate assertions which are combined with the assertion-specific division operator as interval constants. In the case in which shared dependencies do span operands (a reference to the same cell appears both in the expanded formula for the divisor and dividend), a canonical representation for rational functions may lead to a viable algorithm for propagation through division with shared dependencies. However, a more general solution would be the development of algorithms that are able to propagate assertions through general functions with constant, or periodic, discontinuities (such as division and tangent).

Countif and round are representative of functions that present difficulties because they are discontinuous, mapping elements of the Reals to elements of the Integers (or the Natural numbers, in the case of countif). In order to maintain Property 2 (Correctness), each integer in the resulting range would be represented as a degenerate sub-assertion, potentially generating an infinite set of sub-assertions on a cell. By "smoothing over" these discontinuities (as opposed to accepting only exact matches), and propagating approximate assertions that span discontinuities smaller than a certain size, it may be viable to propagate assertions through these functions. For example, instead of a tight assertion such as `0.5, 1.5, 2.5, 3.5, 4.5` on a cell with a formula such as "round(X)+0.5", the assertion `[0 5]` could be generated, which accepts values such as 2.75. This behavior offers a tradeoff between the Reliability and Correctness Properties. In our approach, we have elected to maintain Correctness, although further investigation into approximations is warranted. This tradeoff leads to understandability issues as well. Can the Correctness of an asser-

tion be communicated to the user in a useful way? When assertions of various "strengths" are present, how can these assertions be differentiated?

Future examination of batch algorithms for assertion propagation is also warranted. Batch algorithms, perhaps running in the background, may be able to process a large portion of a spreadsheet that is off screen more quickly than an interactive approach, ultimately allowing more computationally intensive approaches to dynamic assertion propagation to be applied to the spreadsheet paradigm.

8 Conclusions

We have presented an incremental and interactive approach to dynamic assertions to improve program correctness in end-user programming environments, and we have evaluated this approach in regard to four properties: Reliability, Correctness, Responsiveness and Usefulness.

Reliability: In order to provide an approach that is reliable, every cell whose ancestors have assertions must also have an assertion. We have identified two situations that pose difficulties for propagation—shared data-flow dependencies, and the if-problem. We have provided two algorithms, α-propagation and β-propagation, that are capable of propagating dynamic assertions through a variety of spreadsheets, some with instances of the shared data-flow problem. We have also proven a lower bound on the classification of the general if-problem, demonstrating that it is at least as difficult as the problem of Satisfiability.

Correctness and Responsiveness: For each algorithm presented (α-propagation and β-propagation) we have proven the correctness of the algorithm, ensuring that when assertions are propagated, they are correct. Thus, we have shown that our approach to dynamic assertions is correct, fulfilling Property 2. To fulfill the third property, Responsiveness, we placed restrictions on the types of formulas through which we would attempt to propagate assertions. Based on previous studies we have made a number of key simplifying assumptions about the type of assertions entered, and we have shown that under these assumptions α-propagation and β-propagation calculate correct assertions without violating the property of immediate visual feedback, which is characteristic of the spreadsheet paradigm. We then conducted an examination of real-world spreadsheets to determine the impact of these restrictions. In this examination, propagation succeeded in 70% of the spreadsheets considered.

Usefulness: Dynamic assertions were evaluated with regard to Usefulness through empirical evaluations. Empirical results show that not only do dynamic assertions improve correctness, but end users are able to make use of dynamic assertions with no prior training.

We have also presented open questions regarding the area of dynamic assertion propagation. These questions were suggested by the findings of our examination of real-world spreadsheets. These questions also lead to the possibility of approximations that "smooth over" discontinuities in assertions. This would involve adjusting the tradeoff between

Correctness, Reliability and Responsiveness to further increase the ability to generate useful dynamic assertions for end-user programmers.

References

[1] N.H. Able. Beweis der unmöglichkeit, algebraische gleichungen von höheren graden als dem vierten allgemein aufzulösen. In L. Sylow and S. Lie, editors, *Oeuvres Completes*, pages 66–87. Johnson Reprint Corp., New York, 1988.

[2] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, New York, NY, 1983.

[3] M. Auguston, S. Banerjee, M. Mamnani, G. Nabi, J. Reinfelds, U. Sarkans, and I. Strnad. A debugger and assertion checker for the awk programming language. In *International Conference on Software Engineering*, 1996.

[4] Y. Ayalew and R. Mittermeir. Spreadsheet debugging. In *Proceedings of the European Spreadsheet Risks Interest Group*, Dublin, Ireland, July 24–25, 2003.

[5] G. Birkhoff and S. MacLane. *A Survey of Modern Algebra*. The Macmillan Company, New York, NY, 1941.

[6] N. Bjørner, A. Browne, E. Chang, A. Colon, A. Kapur, H.B. Sipma, T.E. Uribe, and Z. Manna. *STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562*. Computer Science Departemnt, Stanford University, November 1995.

[7] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997.

[8] B. Boehm and V. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

[9] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation library. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

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

[11] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the* 25^{th} *International Conference on Software Engineering*, pages 93–103, Portland, OR, May 3–10, 2003.

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

[13] Paul Carlson, Margaret Burnett, and Jonathan Cadiz. A seamless integration of algorithm animation into a visual programming language. In *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, pages 194–202, Gubbio, Italy, May 1996.

[14] C. Corritore, B. Kracher, and S. Wiedenbeck. Trust in the online environment. In *HCI International*, volume 1, pages 1548–1552, New Orleans, LA, August 2001.

[15] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D. Thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[16] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[17] W. M Faucette. A geometric interpretation of the solution of the general quartic polynomial. *American Mathmatical Monthly*, 103(1):51–57, January 1996.

[18] M. Fisher, M. Cao, G. Rothermel, C.R. Cook, and M.M. Burnett. Automated test case generation for spreadsheets. In *Proceedings of the* 24^{th} *International Conference on Software Engineering*, pages 141–151, Orlando, Florida, May 19–25, 2002.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, N.Y., 1979.

[20] P. Griebel, G. Lehrenfeld, W. Mueller, C. Tahedl, and H. Uhr. Integrating a constraint solver into a real-time animation environment. In *IEEE Symposium on Visual Languages*, pages 12–19, Boulder, Colorado, September 1996.

[21] S. Hangal and M. Lam. Tracking down software bugs using automatic anomoly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, Orlando, FL, May 2002.

[22] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, pages 56–69, Orlando, Florida, November 1998.

[23] R. B. King and E. R. Cranfield. An algorithm for calculating the roots of a general quintic equation from its coefficients. *Journal of Math and Physics*, (32):823–825, 1991.

[24] Thomas Kunstmann, Martin Frisch, and Robert Muller. A declarative programming environment based on constraints. In *IEEE Symposium on Visual Languages*, pages 120–121, Darmstadt, Germany, September 1995.

[25] W. Leler. *Constraint Programming Languages*. Addison-Wesley, 1988.

[26] K. Marriott and P. J. Stuckey. *Programming With Constraints: An Introduction*. The MIT Press, Cambridge, Massachusetts, 1998.

[27] B. Meyer. Design by contract. *Computer*, 25:40–51, October 1992.

[28] R.C. Miller and B.A. Myers. Outlier finding: Focusing user attention on possible errors. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 81–90, November 2001.

[29] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.

[30] Brad A. Myers, Robert C. Miller, Rich McDaniel, and Alan Ferrency. Easily adding animations to interfaces using constraints. In *ACM Symposium on User Interface Software and Technology*, pages 119–128, Seattle, Washington, November 1996.

[31] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[32] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Francisco, CA, 1992.

[33] R. Panko. Finding spreadsheet errors: Most spreadsheet errors have design flaws that may lead to long-term miscalculation. *Information Week*, page 100, May 1995.

[34] R. Panko. What we know about spreadsheet errors. *Journal on End User Computing*, pages 15–21, Spring 1998.

[35] R. Panko and R. Halverson. Spreadsheets on trial: A survey of research on spreadsheet risks. In *Proceedings of the* 29^{th} *Hawaii International Conference on System Sciences*, January 1996.

[36] O. Raz, P. Koopman, and Shaw M. Semantic anomaly detection in online data sources. In *Proceedings of the International Conference on Software Engineering*, pages 302–312, Orlando, FL, May 2002.

[37] D. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Soft. Eng.*, pages 19–31, Janurary 1995.

[38] D. Rosenblum, S. Sankar, and D. Luckham. Concurrent runtime checking of annotated ada programs. In *Conf. Foundations of Software Technology and Theoretical Computer Science*, pages 10–35, December 1986.

[39] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10(1):110–147, January 2001.

[40] G. Rothermel, L. Li, C. Dupuis, and M. Burnett. What You See Is What You Test: A methodology for testing form-based visual programs. In *Proceedings of the* 20^{th} *International Conference on Software Engineering*, pages 198–207, June 1998.

[41] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. Green, and G. Rothermel. An empirical evaluation of a methodology for testing spreadsheets. In *Proceedings of the* 22^{nd} *International Conference on Software Engineering*, pages 230–239, June 2000.

[42] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. *Computer*, pages 32–41, March 1993.

[43] Mark Stadelmann. A spreadsheet based on constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 217–224, Atlanta, Georgia, November 1993.

[44] J. Summet and M. Burnett. End-user assertions: Propagating their implications. Technical Report 02-60-04, Oregon State University, Corvallis, OR, August 2002.

[45] W. Wade. *An Introduction to Analysis*. Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[46] C. Wallace and C. Cook. End-user assertions in Forms/3: and empirical study. Technical Report TR 01-60-11, Computer Science Department, Oregon State University, September 2001.

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

[48] E. W. Weisstein. C-k function. From MathWorld–A Wolfram Web Resource, 1999. `http://mathworld.wolfram.com/C-kFunction.html`, Last accessed: May 26, 2004.

[49] E. W. Weisstein. Quintic equation. MathWorld–A Wolfram Web Resource, 1999. `http://mathworld.wolfram.com/QuinticEquation.html`, Last accessed: May 26, 2004.

[50] D. Welch and S. String. An exception-based assertion mechanism for C++. *Journal of Object Oriented Programming*, 11(4):50–60, 1998.

[51] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 305–312, Fort Lauderdale, FL, April 5–10, 2003.