END-USER ASSERTIONS: PROPAGATING THEIR IMPLICATIONS

1. INTRODUCTION

1.1 SPREADSHEET ERRORS AND END-USER SOFTWARE ENGINEERING

Spreadsheet languages are the most commonly used end-user programming paradigm. In fact, their popularity is so great that they may be the most widely used of all programming languages. However, spreadsheets created by end-users are subject to the same faults and errors as programs created by professional programmers. In one empirical study, 44% of spreadsheets created by experienced users contained faults [Brown and Gould 1987]. Other spreadsheet model building experiments back up this result, finding that 20-40% of the created spreadsheets contained errors [Teo and Tan 1997; Panko 1995; Panko 1998].

One factor that may cause these errors is that spreadsheets are generally created in an ad-hoc manner, without a clear design plan or formal specification [Brown and Gould 1987; Cragg and King 1993]. Additionally, once created, many spreadsheets continue to grow in size and undergo constant maintenance. In a survey of spreadsheets from 10 companies, sizes ranged from 150 to 10,000 cells, and one spreadsheet was in its 60th version [Cragg and King 1993]. Once developed, even experienced spreadsheet users have difficulty comprehending and debugging spreadsheets [Hendry and Green 1994]. To make matters worse, spreadsheet creators are overconfident in the reliability of their spreadsheets [Brown and Gould 1987; Wilcox et al. 1997; Rothermel et al. 2000].

In an effort to improve spreadsheet reliability, the Forms/3 research group at Oregon State University has been developing the concept of end-user software engineering. In end-user software engineering, we recognize that while end-users cannot be expected to learn and use traditional software engineering practices (e.g., formal specifications, code reviews, white and black box testing, formal correctness proofs),

they can benefit from devices that embed these techniques (behind the scenes) in the environment to assist them in producing more reliable spreadsheets.

We have already developed and begun preliminary evaluation of (1) methods that encourage spreadsheet testing based upon code coverage theory [Rothermel et al. 1998, 1999], (2) fault localization methods [Reichwein et al. 1999], and (3) automatic test-case generation techniques [Cao 2000]. The results of empirical studies show that users of our testing methods were able to detect more errors [Cook et al. 1999], and were less overconfident about their spreadsheets [Rothermel et al. 2000].

Due to the success of this previous work, we have continued to explore ways in which the benefits of formal software engineering techniques can be brought to end-users. Our goal is to develop methodologies that allow end-users to receive the benefits of the use of formal software engineering techniques without requiring formal training in these techniques. In this thesis, we move toward this goal by exploring how assertions can be incorporated into a spreadsheet language.

1.2 ASSERTIONS AND FORMAL TECHNIQUES

Assertions commonly take the form of preconditions, (conditions which must hold before executing a logical block of code), post-conditions (conditions which must hold after executing a logical block of code), and invariants (conditions which must always be true). Programmers may use assertions to detect exceptional conditions, insure the integrity of inputs, enforce requirements, or document their assumptions.

1.2.1 Assertions as a representation of the user's mental model

When creating a spreadsheet the user has a mental model of how it should operate. One approximation of this model is the cells and formulas they enter, but unfortunately the formulas may contain inconsistencies or errors. These formulas and cell relations are only one way of representing the user's model of the problem and its solution. They are a declarative representation of the user's model, which convey information on how to generate the desired result. However, the assertion mechanism allows the user to convey and specify to the computer other beliefs they may hold about their spreadsheet. For example, in some cases a user may have a clear understanding of the appropriate range for an input or output cell (a property of their mental model).

The assertion mechanism allows users to more fully communicate their mental model of the problem, both to the computer, and other users who may use the spreadsheet (including themselves in the future). By receiving this assertion information, in addition to formulas, the system gains the ability to cross-check the user's mental model as represented by the assertions they create against the way it is represented by the actual formulas and cell references.

The system does this cross-checking by reasoning about the assertions and their interactions with the formulas and references of the spreadsheet. The propagation of assertions (covered in Chapter 3) through the spreadsheet's formulas assists the user in understanding the behavior of the spreadsheet, and allows the system to identify conflicts between the assertions and the formulas of the spreadsheet. This form of checks and balances is especially important in the process of end-user programming, where most programs are written without a formal written specification. Indeed, the user's mental model is usually the only specification that exists, and it is often subject to major revisions and possible errors throughout the development process.

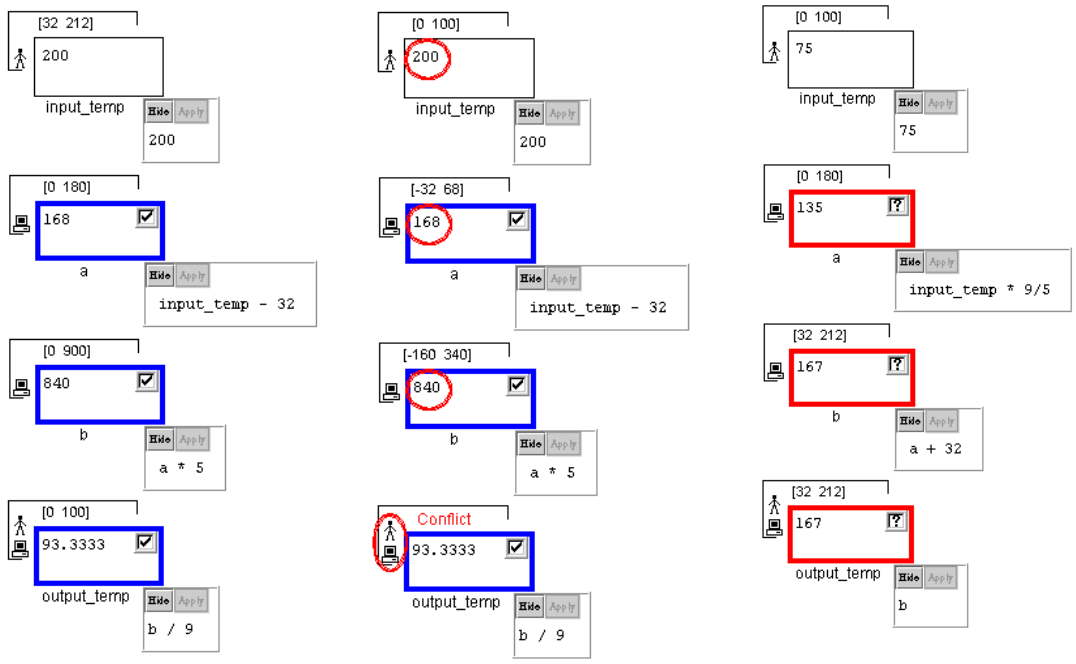
1.2.2 Cross checking of specifications

A formal method used in real-time programming is the *dual language approach*, in which two different specifications are developed, one model-based and one property-based [Ostroff 1989]. The model-based specification specifies how the system will work, and the property-based specification specifies properties (invariants) which will be true about the system. The existence of both specifications allows them to be checked against each other to detect inconsistencies even before the implementation is begun. We do not expect end-users to develop formal specifications before beginning work on their spreadsheet. However, by propagating user specified assertions through the spreadsheet's formulas, and then comparing the results against other user specified assertions, we hope to use assertions, in addition to the formulas of the spreadsheet, to perform a similar cross-checking.

1.3 THE USERS VIEW OF ASSERTIONS

We will now present a simple scenario that introduces our prototype assertion propagation engine in the context of the Forms/3 research language [Burnett et al. 2001a;

Burnett and Gottfried 1998]. Figure 1-a shows a portion of a spreadsheet which 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 specified "guard" on this cell which limits the value of the cell to between 32 and 212. 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. The a and b cells have guards generated by the system (as indicated by the computer icon) which reflect the propagation of the user guard 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 range (as indicated by the co-existence of the computer and the person icon in the guard). This agreement was determined by propagating the user specified guard on the input_temp cell through the formulas and comparing it with the user specified guard on the output_temp cell.



(a) (b) (c)

Figure 1: A temperature conversion (°F to °C) spreadsheet at three points in a modification task of reversing the conversion (°C to °F). As initially given to the user (a), showing the system's response (b) to the modifications of the guard on cell input_temp to range from zero to 100, and the final spreadsheet after all modifications haven been made (c).

An end user is given this spreadsheet and told to change the direction of the conversion and make it convert from degrees Celsius to degrees Fahrenheit. One possible scenario for their actions while making this change follows:

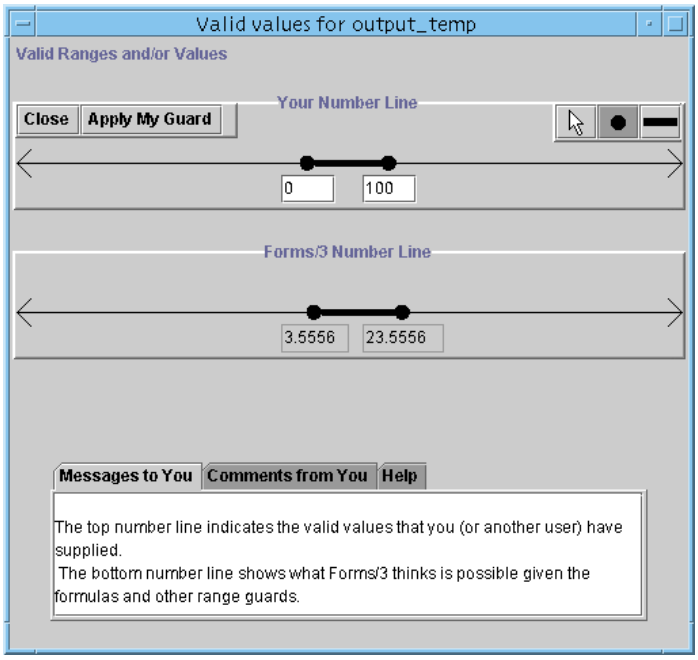


Figure 2: This dialog which displays both the assertion propagated by Forms/3 and the user specified assertion, is viewed by clicking on a guard. At this point in the modification task, the dialog is indicating that Forms/3 does not agree with the user supplied range. The user correctly interprets this to mean that there is a problem with their formula for the output_temp cell.

First, the user changes the guard on the input_cell to range from 0 to 100. This results in the appearance of several red violation ovals (See Figure 1-b), as the values in input_cell, a, b, and output_cell are now out of range and the guard on output_cell is now in conflict with the previously specified guard for that cell. The user decides "that's OK" for now, and changes the value in input_cell from 200 to 75 ("something between zero and 100") and sets the formula in cell a to "input_cell * 9/5" and the formula in cell b to "a + 32".

At this point, the guard on cell b has a range from 32 to 212, and because the user has combined two steps in cell a's formula (multiplication and division), they have obtained the correct value in cell b, as opposed to the output_cell (which still has the

formula “b / 9”). The user now chooses to deal with the conflict message on the output_cell, and clicks on the guard to view the number-line (See figure 2).

Seeing that the Forms/3 number-line ranges from 3.5556 to 23.556 the user mutters “There’s got to be something wrong with the formula” and edits output_cell’s formula, making it a reference to cell b. This results in the value of output_cell being correct, although a conflict still exists because the previous user specified guard remains at zero to 100. The user brings up the number-line again, and seeing that Forms/3’s range is the expected 32 to 212, changes the user guard to agree, which removes the final conflict and completes the modification task. (See Figure 1-c).

This scenario is a summary description of the behavior shown by an end-user participant in a protocol analysis experiment run by Christine Wallace as part of her Masters research. The quotes were taken from a recording of the participant’s “think-aloud” commentary. The scenario demonstrates the type of interaction we wish to foster between the end-user and the programming environment. Additionally, the protocol analysis showed that end-users are able to understand the idea of “guards” and how they are propagated by the system, can understand and solve conflicts, and collaborate with the assertion mechanisms described in this thesis. Further information from the protocol analysis will be presented in Section 3.4.

1.4 OVERVIEW OF THIS WORK

This work uses the idea of an assertion (represented as “guards” to end-users) as the basis for a collaborative system between the user and the programming environment. Users are able to place assertions on cells to ensure specific properties, for example “this cell will contain a number between zero and one hundred”. Assertions provide three main benefits. First, they act as documentation of the user’s mental specification for the program. This documentation conveys the users beliefs to others should the program be distributed, and also provides a reminder for the original user at a later time. Second, they provide protection of the specified properties. Users of the program will receive a warning when violating the assertions. Finally, in addition to providing protection of properties and documentation, the system can deductively reason about the assertions and their interactions with the formulas of the spreadsheet by propagating the assertions

through formulas. The system attempts to leverage the user-specified assertions into extra system-generated assertions. These system-generated assertions serve two purposes. First, they might assist the user in understanding the behavior of their spreadsheet. Second, they allow the system to cross-check the users assertions and formulas, and to detect certain types of program errors and logical errors in the user’s mental model of the problem.

2. RELATED WORK

Traditionally, assertions in the form of preconditions, post-conditions, and invariants, have provided a method for programmers to reason about the integrity of their logic, document their assumptions, and catch exceptions. However, the only widely used language that natively supports assertions is Eiffel. In order to allow programs in other languages to share at least some of the benefits that come from assertions, methods to graft support for assertions onto languages such as C, C++ and Awk have been developed (e.g., [Welch and String 1998; Curcio 1998; Auguston et al. 1996]). These approaches allow professional programmers to manually annotate their programs with assertions. For example, the C language currently provides an "assert" macro that is expanded by the preprocessor into an "if" statement that halts execution if the assertion fails.

In the software engineering community, many applications of assertions to software engineering problems have been investigated, and the use of assertions to help with many of these has proven promising. For example, there has been research on deriving runtime consistency checks for Ada programs, working from the high-level specification language Anna [Luckham 1985; Sankar 1991; Rosenblum et al. 1986; Sankar 1993; Luckham 1990]. Rosenblum has shown that these assertions can be effective at detecting runtime errors [Rosenblum 1995], and has classified various types of assertions that may be effective for detecting faults.

2.1 DERIVING ASSERTIONS COMPARED TO OTHER FORMS OF ANALYSIS AND INTERPRETATION

In addition to allowing programmers to define their own assertions, methods have been developed and implemented that can detect statistically likely program invariants by extensive examination of a program's behavior over a large test suite [Ernst et al. 1999]. Figure 3 shows an example of this system in operation. As this approach uses statistically based inference, it requires a very large number of tests to gain high confidence levels, and can never prove the absolute correctness of its results. Because this approach requires observation of an instrumented program over a large test suite, it would be difficult to

apply during the incremental construction of a spreadsheet. Ernst et al. demonstrated how their method could assist experienced programmers with maintenance tasks when applied

```
15.1.1:::END   100 samples
       N = I = N_orig = size(B)            (7 values)
       B = B_orig                          (100 values)
       S = sum(B)                          (96 values)
       N in [7..13]                        (7 values)
       B                                   (100 values)
          All elements >= -100             (200 values)
```

Figure 3: An example from [Ernst et al. 1999] showing likely invariants inferred by their system at the end of a program which “sums the values in array B (of length N) into result variable S”. The results above were inferred by observing the instrumented program’s behavior on “100 randomly generated arrays of length 7 to 13, in which each element was a random number in the range -100 to 100, inclusive”.

to previously built programs, but did not claim that it would be appropriate for use by end users.

Jeffords and Heitmeyer have presented a system which automatically generates state invariants from an operational (model based) specification expressed in SRC (Software Cost Reduction) tabular notation [Jeffords and Heitmeyer 1998]. Figure 4 presents an example from their paper of the application of their algorithm to the SRC mode transition table specifying an automobile cruise control system.

Old Mode	Event	New Mode
1 Off	@T(IgnOn)	Inactive
2 Inactive	@F(IgnOn)	Off
3 Inactive	@T(Lever = const) WHEN IgnOn AND EngRunning AND NOT Brake	Cruise
4 Cruise	@F(IgnOn)	Off
5 Cruise	@F(EngRunning)	Inactive
6 Cruise	@T(Brake) OR @T(Lever = off)	Override
7 Override	@F(IgnOn)	Off
8 Override	@F(EngRunning)	Inactive
9 Override	@T(Lever = resume) WHEN IgnOn AND EngRunning AND NOT Brake OR @T(Lever = const) WHEN IgnOn AND EngRunning AND NOT Brake	Cruise

Initially: $M = \texttt{Off} \wedge \neg\texttt{IgnOn} \wedge \neg\texttt{EngRunning}$

Table 1: Mode Transition Table for Cruise Control.

i	Mode m	$N_i(m)$	$X_i(m)$	$P_i(m)$	Comments
1	Off	$\overline{I} \vee \overline{I} \vee \overline{I} \vee \overline{I}\wedge\overline{E}$	$\{\overline{I}\}$	$\overline{I}$	ISP gives 4th DJ
	Inactive	$I \vee \overline{E} \vee \overline{E}$	$\{I\}$	$true$	—
	Override	$B \vee O$	$\{I, E\}$	$true$	—
	Cruise	$C\wedge I\wedge E\wedge\overline{B}\wedge\overline{O}\wedge\overline{R}\wedge\overline{L} \vee R\wedge I\wedge E\wedge\overline{B}\wedge\overline{O}\wedge\overline{C}\wedge\overline{L}$	$\{I, E, \overline{B}, \overline{O}\}$	$I\wedge E\wedge\overline{B}\wedge\overline{O}$	Apply OIA, CET
2	Off	$\overline{I} \vee \overline{I} \vee \overline{I} \vee \overline{I}\wedge\overline{E}$	$\{\overline{I}\}$	$\overline{I}$	Fixpoint reached?
	Inactive	$I \vee \overline{E}\wedge I\wedge\overline{O}\wedge\overline{B} \vee \overline{E}$	$\{I\}$	$true$	Apply P_1(Cruise), OIA to 2nd DJ
	Override	$B\wedge I\wedge E\wedge\overline{O} \vee O\wedge I\wedge E\wedge\overline{B}\wedge\overline{C}\wedge\overline{R}\wedge\overline{L}$	$\{I, E\}$	$I\wedge E$	Apply P_1(Cruise), OIA, & CET
	Cruise	$C\wedge I\wedge E\wedge\overline{B}\wedge\overline{O}\wedge\overline{R}\wedge\overline{L} \vee R\wedge I\wedge E\wedge\overline{B}\wedge\overline{O}\wedge\overline{C}\wedge\overline{L}$	$\{I, E, \overline{B}, \overline{O}\}$	$I\wedge E\wedge\overline{B}\wedge\overline{O}$	Fixpoint reached?
3	Off	$\overline{I} \vee \overline{I} \vee \overline{I} \vee \overline{I}\wedge\overline{E}$	$\{\overline{I}\}$	$\overline{I}$	Fixpoint already reached?
	Inactive	$I \vee \overline{E}\wedge I\wedge\overline{O}\wedge\overline{B} \vee \overline{E}\wedge I$	$\{I\}$	I	Apply P_2(Override), OIA to 3rd DJ
	Override	$B\wedge I\wedge E\wedge\overline{O} \vee O\wedge I\wedge E\wedge\overline{B}\wedge\overline{C}\wedge\overline{R}\wedge\overline{L}$	$\{I, E\}$	$I\wedge E$	Fixpoint reached?
	Cruise	$C\wedge I\wedge E\wedge\overline{B}\wedge\overline{O}\wedge\overline{R}\wedge\overline{L} \vee R\wedge I\wedge E\wedge\overline{B}\wedge\overline{O}\wedge\overline{C}\wedge\overline{L}$	$\{I, E, \overline{B}, \overline{O}\}$	$I\wedge E\wedge\overline{B}\wedge\overline{O}$	Fixpoint already reached?
4	Off	$\overline{I} \vee \overline{I} \vee \overline{I} \vee \overline{I}\wedge\overline{E}$	$\{\overline{I}\}$	$\overline{I}$	Fixpoint reached!
	Inactive	$I \vee \overline{E}\wedge I\wedge\overline{O}\wedge\overline{B} \vee \overline{E}\wedge I$	$\{I\}$	I	Fixpoint reached!
	Override	$B\wedge I\wedge E\wedge\overline{O} \vee O\wedge I\wedge E\wedge\overline{B}\wedge\overline{C}\wedge\overline{R}\wedge\overline{L}$	$\{I, E\}$	$I\wedge E$	Fixpoint reached!
	Cruise	$C\wedge I\wedge E\wedge\overline{B}\wedge\overline{O}\wedge\overline{R}\wedge\overline{L} \vee R\wedge I\wedge E\wedge\overline{B}\wedge\overline{O}\wedge\overline{C}\wedge\overline{L}$	$\{I, E, \overline{B}, \overline{O}\}$	$I\wedge E\wedge\overline{B}\wedge\overline{O}$	Fixpoint reached!

Key

ISP:	Initial State Predicate	I:	IgnOn
OIA:	One Input Assumption	E:	EngRunning
CET:	Constraint from Enumerated Type	B:	Brake
$N_i(m)$:	Mode Entry Condition for Mode m at ith pass	O:	Lever =off
$X_i(m)$:	Unconditional Exit Set for Mode m at ith pass	C:	Lever = const
$P_i(m)$:	Invariant computed for Mode m at ith pass	R:	Lever = resume
DJ:	Disjunct of $N_i(m)$	L:	Lever = release

Table 2: Mode Invariant Generation for Cruise Control

Figure 4: These tables, from [Jeffords and Heitmeyer 1998], illustrate the SRC mode transition table in tabular form (top) for an automobile cruse control system which was produced by the system designer or programmer and the corresponding table (bottom) showing assertions in the form of entry conditions, exit sets, and invariants generated by their algorithm over the course of four iterations.

The formal proof system presented in [Bjørner et al. 1997] attempts to prove a given goal by generating intermediate assertions. This system is implemented in STeP, the Stanford Temporal Prover [Bjørner et al. 1995], a computer-aided formal verification system for concurrent and reactive systems. Users specify axioms about their system and

```
SPEC (* Greatest common divisor spec file *)
```

properties they wish to prove in a textual specification file (see Figure 5). Neither of these systems are designed to be used by end users.

The method in which we propagate assertions shares similarities with abstract interpretation. However, abstract interpretation derives information from the static semantics of a program as written, and not based upon specification level information such as assertions provided by the programmer [Abramsky and Hankin 1987]. We deductively propagate user specified assertions through program statements, instead of extracting information directly from the program. See Figure 6. Because these assertions provide abstract information about the program, the result of our reasoning (propagation of assertions through program statements) is also abstract.

Abstract interpretation is commonly used as a program analysis tool to provide optimization information at compile time, and not to enhance programmer understanding. Many programmers might not find information such as the strictness or mode of parameters to be useful for program comprehension, testing or debugging tasks. The method of propagating assertions outlined here is not intended to improve execution speed, but to improve (end-user) programmer understanding, and program correctness. By involving the user in the specification of the initial assertions, we hope to derive information that is useful to the user, as opposed to the computer.

The processes of abstraction and concretization popularized by the Cousot's framework do not directly apply as the abstraction step is unnecessary, its job being performed by the user when generating assertions. There are similarities between the value conflict checking of assertions and concretization [Cousot and Cousot 1977].

This work uses the mathematical foundations of interval arithmetic when propagating range sub-assertions. A complete introduction to interval arithmetic can be found in textbooks such as [Moore 1966; Alefeld and Herzberger 1983].

Earlier work with static type inference for Forms/3 [Djang et al. 2000] used propagation with respect to type guarantees, but made no predictions as to the values of the inferred types, only that they would support certain required operations. When our system is able to propagate assertions, as a side effect we are able to deduce the type(s) of

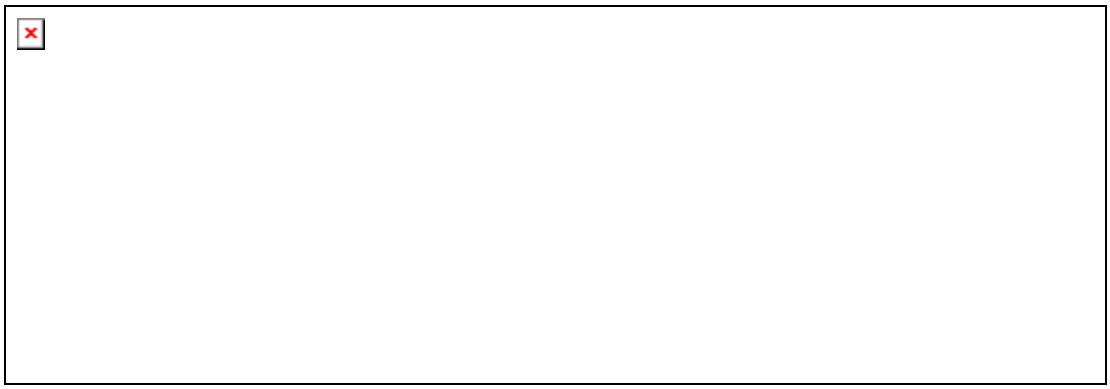


Figure 6: Correspondence between the Cousot's abstract interpretation framework (left) and assertion propagation. While assertion propagation and abstract interpretation share similarities, assertions are not generated by abstraction from the static semantics of the program.

the possible values in addition to providing more specific information about the values themselves. However, we are not guaranteed that assertions will exist to propagate, or that we will be able to propagate them through every formula. Because of this, our system does not replace the static type inference system devised for Forms/3.

2.2 SPREADSHEETS AND OTHER END-USER PROGRAMMING SYSTEMS

End-user programming systems are designed to allow non-programmers to tell a computer how to perform some action. They range from research prototypes which attempt to learn what the user wants by watching them demonstrate how objects should act, to the popular commercial spreadsheet system Microsoft Excel, which is the most widely used end-user programming environment today.

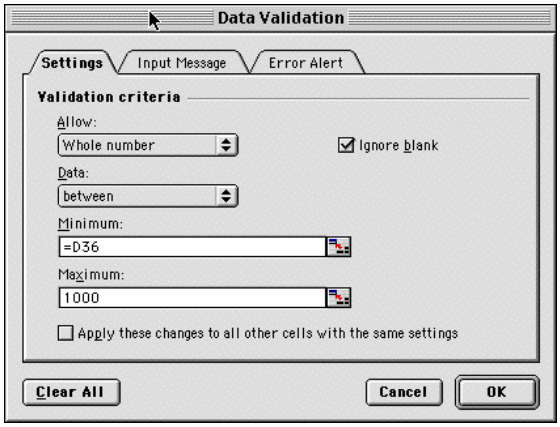


Figure 7: An example of Excel's data validation dialog.

In Excel, data values are stored in cells which are fixed in a physical location by a grid. Excel offers users a "Data Validation" feature through which they can specify limits on the value of a cell. Figure 7 shows the dialog box that can be used to place limits on cell(s). These limits apply only to the specific cell(s) specified by the user, and are checked only when the user edits that specific cell's formula. It is possible for a formula change elsewhere to bring a cell's value outside of a limit without a warning from Excel. If the user edits a cell's formula to produce an invalid value, a dialog will appear telling them they have entered an invalid value, prompting them to retry their formula edit, or cancel and revert back to the previous formula. Users are not allowed to enter an invalid value and fix the problem later as part of a multi-step modification task. Excel does not reason about these limits, propagate them to other cells, or provide a visual display of limits after they are placed.

Forms/3 is a research spreadsheet language. Another such language from the same roots is Formulate [Ambler 1999]. As in Forms/3, objects are placed by the user, and the only use of a grid-like structure is within objects that represent structured data such as arrays, lists or tables. One of the main contributions of Formulate is the ability to work with most structured data (arrays) without having to index data implicitly. Instead, the

data is partitioned into regions. The expressions which define the values of regions exist in the main partition, and multiple "view" partitions can be defined to access various portions of the data without implicit indexing.

Formulate is geared towards users with a high-school algebra background and spreadsheet experience. However, some other end-user programming systems that look very unlike spreadsheets are geared towards children, such as KidSim/Cocoa [Cypher and Smith 1995; Heger et al. 1998], AgentSheets [Perrone and Repenning 1998; Repenning and Summer 1995], and ToonTalk [Kahn 1996]. KidSim/Cocoa and AgentSheets both use graphical rewrite rules, where the user specifies a rule that describes how to modify the objects in the world, going from a before to an after situation. These systems are more like simulations, where the user specifies how objects in the environment should behave. AgentSheets even allows the user to generalize behaviors from one type of object to another ("Cars move on roads like trains move on tracks").

In ToonTalk, the user can either manipulate objects directly, or demonstrate a task to a robot. Once a robot is "taught" how to perform a task, it can be used to automate the task and perform it repeatedly. ToonTalk is an end-user programming systems which uses the concept of "Programming by Demonstration", where the user demonstrates a desired behavior to the system, and the system attempts to develop a program which will exhibit the desired behavior. In ToonTalk, the actions which can be demonstrated are limited, and the user must help the system develop a generalized program by "erasing" extra details with a vacuum cleaner tool.

A more sophisticated programming by demonstration system is Gamut [McDaniel and Myers 1999], an integrated language environment for building software such as interactive 2D board games. The user demonstrates behaviors of game objects, and Gamut attempts to learn and generalize rules or behaviors for the objects. Gamut collaborates with the user during this training process. If it makes a mistake the user presses a "Stop That!" button (to stop an incorrect action) or a "Do Something!" button (to indicate the omission of a desired behavior).

None of these end-user programming systems support assertions to cross-check the program's logic. In systems which allow the user to directly enter formulas or specify calculations such as Forms/3 and Formulate, the user is able to perform tests on values

which could be used to implement ad-hoc assertions (“IF A < 0 THEN **ERROR** ELSE <perform calculations>”). These ad-hoc assertions have no special status within the system, and the system is not able to derive extra meaning from them. A similar technique could be used with rule based systems, by indicating a special action to be taken when an exceptional condition appears. Programming by demonstration systems such as Gamut may infer an assertion from examples provided by the user, but they would be treated as just a learned behavior (“If this box is empty, write ‘ERROR’ over there.”) as opposed to understanding that it represented an exceptional condition (“If this box is empty, abort all further calculations.”).

In all cases, the end-user would have to use other language features to “cobble together” an assertion, which the system would not recognize as a unique construct that could be reasoned about separately from the program itself. To the best of our knowledge, no other end-user programming system supports deductive propagation of assertions as described in this work.

3. DEDUCTIVE PROPAGATION OF ASSERTIONS

3.1 DEFINITIONS

Before introducing the assertion propagation system itself, we define what an assertion is, two subtypes of assertions, and two functions that produce and compare assertions:

3.1.1 General Definitions

Assertion - A function that takes as input the value of a cell, and returns a Boolean value of true or false. A true value indicates that the conditions of the assertion have been met (value is good) and a false value indicates that the conditions of the assertion have not been met (a violation exists between the value of the cell and the assertion). An assertion that is specified by the user is a user specified assertion (USA). An assertion generated deductively by the system (based upon other assertions and cell formulas) is a system generated assertion (SGA). Each assertion is made up of one or more sub-assertions, whose output(s) are combined with a logical OR. For example, we may indicate that a cell can have a numerical value between zero and ten, OR a Boolean value that must be true, OR a divide-by-zero error by ORing together three sub-assertions.

Sub-Assertion - A sub-assertion is an assertion (a function that takes as input the value of a cell and returns a Boolean value of true or false) that accepts a specific class of values that can be contained by a cell. To make our reasoning and implementation easier, we separate various classes of values into separate sub-assertions which we can treat individually. For example, we represent classes of numbers with range sub-assertions, Boolean values with Boolean sub-assertions, and errors with error sub-assertions.

User Specified Assertions - User Specified Assertions (USAs) are assertions that are added, removed, or modified via user interaction with the GUI. The simplest way for a user to add a USA is by deliberately placing one on a cell. Additionally, future versions of the system will attempt to glean USAs by watching the user's actions, and entering into a dialog with the user when appropriate. For example, when the user places an "X" mark in a cell's testing checkbox (an action which indicates the user believes the value in

the cell is incorrect) the GUI will prompt the user to specify why the cell's value is incorrect, and reformat it into a USA.

System Generated Assertions - System Generated Assertions (SGAs) are assertions that are automatically generated by the system, using the forward propagate function (see below), based upon existing assertions and cell formulas. These assertions are updated as formulas or user specified assertions change.

Forward Propagate - A function that takes as input a cell and its formula, and returns either a new assertion for that cell or a null value (if it is unable to propagate an assertion). This new assertion is derived from assertions on cells that are referenced in the formula as described in Section 3.2. The new assertion must return true whenever all the assertions it depends upon return true.

Assertion Consistency Check - A function that takes as input two assertions, and returns a true or false value depending upon a *consistency criterion*. The consistency criterion may vary depending upon language specific factors. In Forms/3, our consistency criterion is that the sub-assertions must be identical with the exception of divide-by-zero error sub-assertions.

Our original consistency criterion required that the assertions be identical. However, we found that it was beneficial to ignore omissions of divide-by-zero errors in User Specified Assertions. Specifically, the user is not required to specify that a divide-by-zero error is one possible outcome of a division. For example: if the system determines that the result of evaluating a formula including a division is between zero and 5000 with a possible divide-by-zero error, while the user only specifies that the result should be between zero and 5000, the system treats the assertions as consistent.

Because the Forms/3 system automatically carries any divide-by-zero errors involved in intermediate calculations to the final output cell(s) during evaluation, the user is not in danger of accidentally using a value that is affected by a divide-by-zero error. Because of the manner in which Forms/3 handles this error, there is no benefit in forcing the user to be explicitly aware of the possibility of such errors. The assertion system does detect the possible error condition, and System Generated Assertions show when a divide-by-zero error is possible. However, the system does not require that a User

Specified Assertion specify the possibility of a divide-by-zero error to be judged consistent with a System Generated Assertion that does. If the assertion propagation system were to be implemented in a language that did not make divide-by-zero errors obvious, it would be beneficial to require that the user acknowledge the possibility of such an error.

One consistency criterion we considered and rejected was a subset-of-values criterion. The subset-of-values criterion would allow the user to specify an assertion which allowed any subset of values accepted by the system generated assertion. Although this would allow the user to omit error conditions (such as the divide-by-zero error case above), it would also allow them to specify ranges such as [0 10] which would not be in conflict with a system generated range of [0 100]. This would undermine one of the major advantages of the assertion system -- namely, the ability to notify the user when their beliefs about the outputs of their program do not match their beliefs about the inputs of the program as propagated through the formulas (statements) which make up the program.

3.1.2 System States

There are several states that a programming system which uses the assertion propagation system can be in. The value-safe and value-conflicted states are mutually exclusive, as are the assertion-consistent and assertion-conflicted states.

Value-Safe - All assertions return true, indicating that the values of all cells meet the conditions of the assertions on those cells.

Value-Conflicted - One or more assertions return false for the current values in their cells.

Assertion-Consistent - For all cells that have both a User Specified Assertion (USA) and a System Generated Assertion (SGA) the two assertions satisfy consistency conditions. In other words, the assertion consistency check function returns true when given the USA and the SGA for each cell that has both a SGA and a USA.

Assertion-Conflicted - The assertion consistency check function returns false when comparing the USA and the SGA on one or more cells.

Fully Propagated - Using the forward propagation method on all cells will result in no new assertions being added to the system. In other words, all propagation that can take place, has taken place. The system takes action after each user edit to maintain itself in a fully propagated state.

3.2 FORWARD PROPAGATION OVERVIEW

The system will attempt to propagate user specified assertions through the data-flow paths of the spreadsheet. By propagating user specified assertions through the formulas which make up the spreadsheet, the system attempts to provide information to the user that assists them in visualizing the behavior of their spreadsheet, and potentially points out conflicts between their mental model and the spreadsheet as written. The propagated assertions supplement the immediate visual feedback (of prototypical values) inherent in the spreadsheet paradigm. For example, in Figure 8, the user can immediately see the effect of their formula on both the prototypical value (thirty-four, which has doubled to sixty-eight) and the potential effect on all valid values (the input range, zero through one-hundred, has also doubled). This is a trivial example, but it serves to demonstrate the principles that are also used with more complicated formulas and multi-cell calculations.

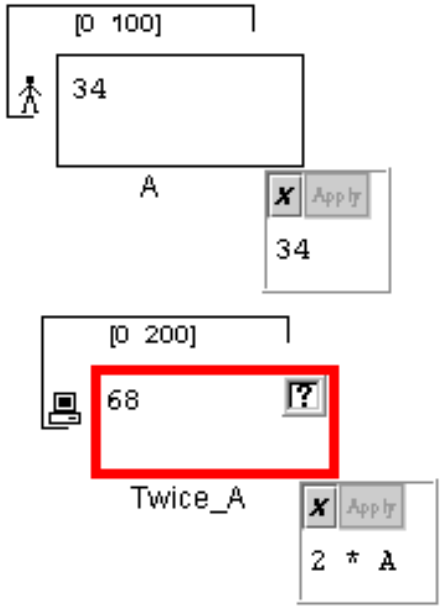


Figure 8: A simple example showing forward propagation of a range assertion. A user specified assertion is indicated by the stick-figure icon, while the system generated assertion is indicated by a computer.

When given a formula (such as "2 * A"), the system first replaces constants with an appropriate assertion[1] (in this case, a range sub-assertion of [2 2]) and all cell references with the assertion (if any) on the referenced cell. The system then replaces the standard numeric operators with assertion-specific versions. In this case, the multiplication operator is replaced with an operator that multiplies assertions. (An alternate implementation method would be overloading the standard numeric operators and replacing only the input values.) When the simple example above is evaluated, the [2 2] range sub-assertion is multiplied by the [0 100] range sub-assertion (using an assertion-specific multiplication operator), giving a range sub-assertion of [0 200].

In the case of the example above, the propagation is simple, and easily computed. To handle more complex formulas the system evaluates each operator and sub-expression of the formula individually, building up the final answer using the same evaluation rules that are used when evaluating the cells value. But instead of calculating the cell's value using input values, the system calculates the cell's assertion, using input assertions. (Input assertions are those assertions on the cells referenced by the formula.)

Assertions for more complex formulas are handled with standard evaluation rules. For a formula such as "A + B - C", an intermediate assertion would be generated for the "A + B" sub-expression (call it Z). This intermediate result would then be used in a " Z -

---

[1] The current implementation supports the replacement of numeric constants with range sub-assertions. Support for replacement of other types of constants is easy to add should they become necessary.

C" expression (processed by the assertion-specific subtraction operator) to generate the final assertion for the entire expression.

Obviously, we must specify an assertion-specific version of each operator we wish the system to propagate assertions through. Our current implementation supports the simple arithmetic (+,-,*,/), relational (=,<,>,<=,>=), logical (AND,OR,NOT) and branching (IF) operators. By implementing these operators, the system is able to propagate assertions through many formulas found in Forms/3 spreadsheets, which is sufficient at this prototype stage. We are free to add support for other operators incrementally (e.g., the less frequently used sin and cosine) as they are needed to support more advanced end-users.

The system maintains a fully propagated state. To do so, system generated assertions are automatically re-calculated whenever one of the following triggers occurs:

- The formula of the cell upon which the assertion is placed is modified by the user.
- An input assertion upon which this assertion is based (either an USA or a SGA) is modified, or removed.

A re-calculation can result in no change to the assertion, a modification of the assertion, or even removal of the assertion. Note that when an assertion is re-calculated, it will trigger the re-calculation of any other system generated assertions that depended on it. This research is focused primarily on the propagation of numeric range assertions, but we have developed and implemented assertions covering Boolean values and possible error conditions (such as a divide by zero) to support the range assertions. In the following sections, we will discuss the types of assertions that have been initially implemented in the Forms/3 system.

3.2.1 Range sub-assertions

3.2.1.1 Representation of range sub-assertions

Range sub-assertions allow limitations to be placed on the possible numeric values of cells. A single range sub-assertion consists of a lower and an upper bound. Each bound can be either inclusive or exclusive. We represent a range sub-assertion textually

by using numbers augmented with either brackets or parentheses, for inclusive and exclusive bounds. The special case of positive and negative infinity is represented with the string “INF”. For example, the range sub-assertion [0 INF] in a cell’s assertion is one possible way to specify that its value should always be positive. If the value can approach zero, but never equal it, the lower bound is made exclusive as follows:

(0 INF].

An assertion can contain multiple range sub-assertions combined with logical OR’s, so the above positive number limitation could also be expressed as [0 5] OR [5 INF]. (These two ranges would be automatically combined by the implementation into a single [0 INF] range.) This feature allows the specification of multiple exclusive ranges, such as [10 100] OR [500 1000]. Range sub-assertions are most useful for numeric values. In the implementation, ranges cover only real numbers, but potentially any range of values (e.g., a list of enumerated types) that can be mapped onto the real numbers could be specified in this manner. As shown in the example above, range sub-assertions can be combined with standard arithmetic operations. The following section discusses the behavior of assertion-specific operators on range sub-assertions.

3.2.1.2 The simple arithmetic operators (addition, subtraction, multiplication, division):

Arithmetic operators in Forms/3 operate solely on numerical values, and hence, the addition, subtraction, multiplication, and division of assertions applies solely to range sub-assertions. The assertion-specific versions of these operators work in a logical manner to produce a correct output assertion (see Chapter 4 for proofs). For example, if the system adds together two assertions having range sub-assertions of [0 10], the resulting assertion will have a range sub-assertion of [0 20].

In more complicated cases, one or both of the input assertions might contain multiple range sub-assertions. In this case, all possible ranges are calculated and the resultant ranges are merged where they overlap. For example, if assertion A has the range sub-assertions [-INF 5] and (20 100) and assertion B has the range sub-assertions [0 10] and [15 20] the result of "multiplying" these two assertions together would be a single assertion with the range sub-assertion [-INF 2000). Section 3.4 covers the correctness of each operator.

3.2.2 Boolean sub-assertions

3.2.2.1 Representation of Boolean sub-assertions

In addition to the simple arithmetical operations, we support the propagation of assertions through relational operators (including the various permutations of the equal, less-than and greater-than operators), the IF operator and the Boolean AND, OR, and NOT logical operators. To support these logical operations, we introduced a Boolean sub-assertion, which can hold one of three possible values, T, F, or TF, indicating that the value will be true, false, or either. Because these three possibilities cover all possible Boolean values, there is no need to have more than one Boolean sub-assertion in each assertion.

3.2.2.2 Relational and IF operators

Relational operators return T, F or TF Boolean sub-assertions depending upon the ranges they receive as arguments. If the result is guaranteed to be T or F for all possible input values, the appropriate Boolean sub-assertion is returned, otherwise the relational operators return a TF Boolean sub-assertion, indicating that either a True or a False result is possible. For example, using the less-than operator on two ranges where the second range is strictly larger than the first, (e.g. "[0 5] < [10 15]") will return a T Boolean sub-assertion, while the opposite case (e.g. "[10 15] < [0 5]") will produce an F Boolean sub-assertion, and using it on two ranges that overlap (e.g. "[0 10] < [5 15]") returns a TF Boolean sub-assertion.

We have defined our assertion-specific IF operator to return the assertion on the THEN clause if its input contains a T Boolean sub-assertion, the assertions on the ELSE clause if it receives an F Boolean sub-assertion, and both the assertions in the THEN and ELSE clauses ORed together if its input is a TF Boolean assertion.

3.2.3 Error and NoValue sub-assertions

While propagating range sub-assertions it is possible to generate an error instead of a valid numeric value such as by using a range which includes zero as the denominator of a division. Because of this, we have implemented an error sub-assertion. An error sub-assertion is generated whenever the assertion inputs to an assertion propagation operator allow values that could generate an error (such as a zero in the denominator of a division). In Forms/3, any operator which receives an error as an argument will produce an error as its result. Therefore, if any assertion-specific operator receives an error sub-assertion as one of its inputs, it will produce that error sub-assertion as part of its output. Note that because we are dealing abstractly with all of the possible values of a cell this does not prevent other sub-assertions from being generated as possible output values. It is possible for a cell to contain either a positive number, OR a divide-by-zero error. For example, if a division near the beginning of a large series of calculations produces an assertion that includes both range sub-assertion(s), and a divide-by-zero error sub-assertion, this error will propagate throughout the rest of the data-flow path, so the final result will include the possibility of having a divide-by-zero error. This error propagation behavior is consistent with all of the Forms/3 operators.

Forms/3 also has a NoValue data type, which has a single element that indicates a cell contains no value. As with errors, all Forms/3 operators which receive a NoValue as an argument will produce a NoValue as its result and all assertion-specific operators will propagate NoValue sub-assertions directly to their outputs to be consistent with the Forms/3 operators.

3.3 LIMITATIONS ON ASSERTION PROPAGATION

The system is able to propagate a correct assertion through an arbitrarily complex formula as long as an assertion-specific version of each required operator exists, and input assertions have no shared dependencies. However, if an operator is used in the

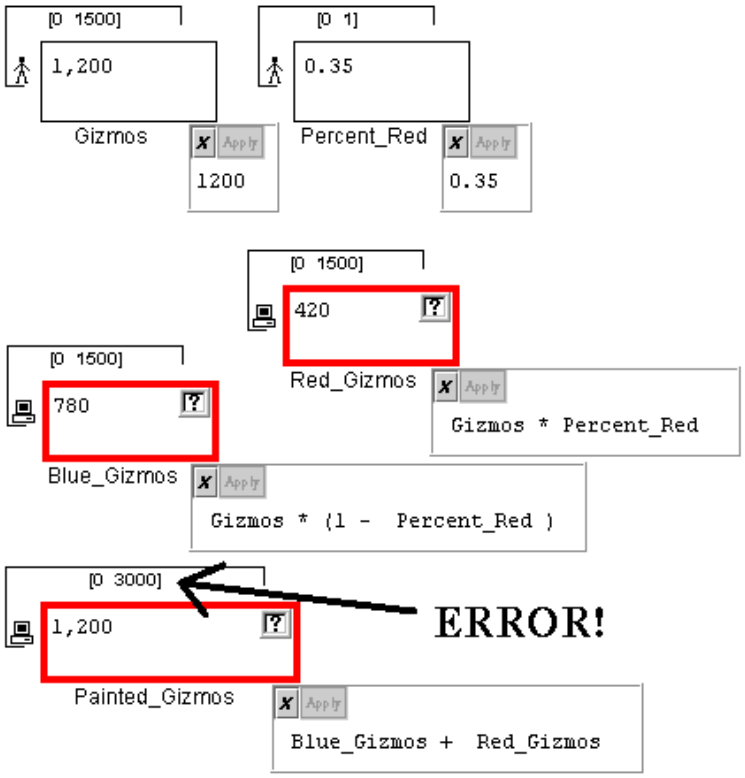


Figure 9: This figure demonstrates the difficulties of propagating assertions through formulas with shared dependencies. The actual implementation does not produce the assertion that is marked with the ERROR arrow.

formula for which an assertion-specific version does not exist, no propagation can take place.

The case of shared dependencies is more subtle, and could lead to the generation of erroneous assertions if not detected. In Figure 9 we present an example of a shared dependency. Cells Blue_Gizmos and Red_Gizmos both reference the Percent_Red cell. This does not affect their propagated assertions, but when they are both referenced by the Painted_Gizmos cell, the shared dependency (on the Percent_Red cell in this example) causes the propagated assertion to be incorrect. This is due to the fact that the Percent_Red cell cannot be set to zero and one at the same time. Another way of saying this is that the distributive law does not always hold for interval arithmetic [Moore 1966]. The current implementation automatically detects shared dependencies and fails

gracefully by not propagating an assertion under those conditions. See Section 6.1 for ideas on future work to address these limitations.

3.4 END-USER COMPREHENSION

As a portion of her Masters research, Christine Wallace performed a protocol analysis experiment of the GUI (which she developed and implemented) and the assertion system using only range assertions. After three pilot sessions, a total of ten subjects were individually presented with two spreadsheet modification tasks. Half of the subjects (5) carried out these tasks while using the assertion propagation system, while the others were used as a baseline for comparison. While primarily testing the GUI, this protocol analysis also tested the end-users' understanding of the underlying assertion propagation algorithms.

In a post session questionnaire the five subjects who used assertions (represented as guards to the users) were asked to manually propagate a guard; in other words, make a prediction on how the guard (assertion) would be propagated through a formula by the assertion propagation algorithms. Four of the five subjects predicted the correct result, with the fifth subject making an arithmetic mistake which they fixed to produce the correct answer when questioned.

Each of the subjects were then asked to make a prediction about what would happen when a guard, which had resulted in the propagation of a system generated guard to a downstream cell, was removed. None of the subjects had previously experienced the removal of a guard. Three of the five subjects correctly predicted that the system would no longer be able to propagate the guard to the downstream cell. The other two subjects couldn't predict what would happen, but once the input guard was removed were able to explain why the propagated guard had also disappeared.

In addition to the actual propagation of assertions, the system presented here generates value violation and assertion conflict messages. First, the assertions themselves identify "invalid" values for cells, and notify the GUI (which circles the bad values with a red "value violation oval"). Second, if the system generates an assertion which does not match some other assertion previously entered by the user, as judged by the assertion consistency check function, it will signal an assertion conflict, which is indicated by the

current GUI with a red “assertion conflict oval” drawn over the two conflicting assertions. All five of the users correctly answered multiple questions about these violations and conflicts, demonstrating that they could correctly interpret the feedback about value violations and assertion conflicts.

4. PROPAGATION METHOD AND CORRECTNESS

It is critical when propagating assertions through formulas and references that the correctness of the result is preserved. First, we must define what it means for a propagated assertion to be "correct". Recall that an assertion is, by definition, "a function that takes as input the value of a cell, and returns a Boolean value of true or false".

A propagated assertion for a specific cell is dependent upon the cell's formula, and the assertions on cells referenced by the formula. We call the referenced cells "input cells" and their assertions "input assertions". The cell for which an assertion is being generated is the "output cell" and the resultant System Generated Assertion is the "output assertion".

Values which are accepted by the input assertions are defined to be valid. The value of the output cell is a result of evaluating the output cell's formula with the input values. **We define an output assertion to be correct if it accepts all and only those output values that can be produced by valid input values.**

In other words, for every set of input values accepted by the input assertions, the propagated (output) assertion must accept the resulting output value. Note that we do not require the output assertion to reject a value that was calculated as the result of an invalid input(s); see Figure 10 for an example.

To determine the correctness of an output assertion, our scope of examination is limited to the output cell and the input cells which are directly referenced by the output cell's formula. The output cell's value is determined solely by the formula of the output cell and the values of the input cells. This formula, when parsed and stored in prefix notation, consists of an expression of the form: "operator (operand-1, operand-2, ..., operand-N)" where each operand is either a cell, a constant, or a nested expression.

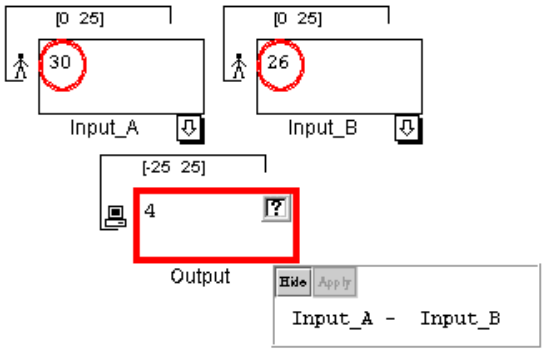


Figure 10: In this example, both input assertions are violated (as indicated to the user by red "conflict" ovals around the values), but because the value in the Output cell could be produced by values which would not violate the input assertions, (e.g. 20 - 16 = 4) the output assertion is not violated.

The propagation algorithm works by replacing the operators (on values) of the formula with their corresponding assertion-specific operators, replacing cell references with the assertions on those cells, replacing constants with constant assertions (or NIL values), and building nested sub-expressions recursively in the same way. This generates an expression of the form:

"assertion-specific-operator (assertion-operand-1, assertion-operand-2, ..., assertion-operand-N)" where each operand is either an assertion, or a nested expression which will produce an assertion (or NIL value). This expression is then evaluated and produces a system generated assertion.

Thus, the proof of correctness requires that:

- The operator and operand replacement is performed correctly.
- Each assertion-specific operator correctly combines the assertions it receives as operands.

These points will be covered in the next two sections.

4.1 CORRECTNESS OF OPERATOR AND OPERAND REPLACEMENT

Operator replacement is done via a direct one-to-one correspondence between standard language operators and their assertion-specific versions. This case is trivially correct, provided that the assertion-specific operators are themselves correct (covered in Section 4.2). If no correspondence exists (i.e. no corresponding assertion-specific operator has been defined) the operator will be replaced with a NIL value. The NIL value represents a lack of information, and an inability to generate an assertion. A non-existent (NIL) assertion can never be incorrect.

There are three types of operand replacements to consider: First, if the operand is a sub-expression it will be treated recursively as above. Second, if the operand is a reference, it will be replaced with either the assertion on the referenced cell (if it exists) or a NIL value. Replacement of a reference to a cell with that cell's assertion is trivially correct in the same sense as operator replacement. If the referenced cell has no assertion, the existence of a NIL as an operand for any assertion-specific-operator will produce a NIL (non-existent) assertion which (as stated above) cannot be incorrect.

Third, if the operand is a constant value, it will be replaced either by a constant assertion (if the system knows how to handle that type of constant) or a NIL value if the system does not know how to generate a constant assertion. Currently, the only constant replacement done is with numbers. In this case, the system replaces the number with a range sub-assertion that includes only that number. For example, the constant 7 would be replaced by an assertion containing only the range sub-assertion [7 7]. Clearly this assertion is correct, since $\forall$ N, N $\in$ <N..N> and no other number has this property.

4.2 ASSERTION-SPECIFIC OPERATOR CORRECTNESS

The following sections show the correctness of the assertion-specific operators we implemented for use with the Forms/3 system. Because the behavior of assertion-specific operators is tied directly with that of the language operators, this proof must be modified if the approach is to be used with a language where the operators exhibit different behavior. This proof serves as a model of how to prove the approach works correctly for a specific language.

4.2.1 Proof of correctness of the assertion+ operator

The assertion+ operator takes as input two assertions, A and B, and produces an output assertion, O. For the assertion O to be correct, the following property must hold:

O(q) is TRUE if and only if $\exists$ x,y such that q = x+y AND A(x) = TRUE AND B(y) = TRUE.

This implies that if the two input values are accepted by their assertion, their summation must be accepted by the output assertion O. On the other hand, if either value is not accepted by their assertion, O only rejects their result (x+y) if there are no two (acceptable) values which produce the same result, as shown in Figure 10.

4.2.1.1 Part 1: Proof of the base case

Because the assertions A and B may contain one or more range sub-assertions, the simple case will be dealt with first: Assertions A and B each contain a single range sub-assertion. Each range sub-assertion is a function which accepts numerical values that are within a specified lower and upper bound. We will use rA and rB to represent the ranges accepted by these sub-assertions. Let rA = [Al , Au] and rB = [Bl , Bu] where Al, Bl and Au,Bu are the lower and upper bounds on the ranges, which have the properties Al ≤ Au and Bl ≤ Bu. From these input ranges, the system must calculate an appropriate output range which will be used to build a range sub-assertion which is part of the final output assertion.

The system calculates four possible bounds for the output range as follows: i1=Al+Bl, i2=Al+Bu, i3=Au+Bl, and i4=Au+Bu. When summing bounds on ranges that reach positive or negative infinity the addition returns an appropriate positive or negative infinity (see Section 4.2.5). The output range sub-assertion, rO is calculated as follows: rO = [Ol,Ou], where the bounds are calculated as: Ol = MIN(i1,i2,i3,i4) and Ou = MAX(i1,i2,i3,i4). Note that in the case of the addition operator, i1 ≤ i2 ≤ i4, and i1 ≤ i3 ≤ i4, which means that Ol = i1 and Ou = i4. The results of the MIN and MAX operators return equivalent results. We use the MIN and MAX operators in our proof because our implementation also uses these operators instead of taking the shortcut of using i1 and i4 directly. This is done because much of the code for the arithmetic operators (+,-,*,/) is shared.

The theory of interval arithmetic [Moore 1966; Alefeld and Herzberger 1983] defines the correctness of this operation. A trivial proof illustrates this fact: We must show that this calculated output range includes all numbers that can be generated by adding together any two numbers in rA and rB. Let Va and Vb be two values in the ranges rA and rB respectively. By definition, Al ≤ Va ≤ Au and Bl ≤ Vb ≤ Au. Hence, i1

≤ (Va+Vb) because i1= Al + Bl, and (Va+Vb) ≤ i4 because i4=Au+Bu. Therefore, when the MIN and MAX operators pick i1 and i4 to form the output range rO, it contains all numbers that can be generated the summation of any two numbers from input ranges rA and rB. This output range is then converted into a range sub-assertion and included in the output assertion.

Hence, for any numbers x,y which are accepted by A and B, their summation will result in a number z=x+y which is accepted by O. As no other numbers are accepted by the output assertion O, the property of Section 4.2.1 is satisfied.

4.2.1.2 Part 2: Proof of the general case

When dealing with more complicated cases, where one or both input assertions (A,B) can contain multiple range sub-assertions, the above operation is performed on all combinations of input sub-assertions, generating multiple range sub-assertions as output. Because the system generates a range sub-assertion for each possible pairing of range sub-assertions from the input assertions A and B, the output range sub-assertions accept all possible numeric summations by part 1.

The two input assertions A and B contain one or more range sub-assertions, $A_1...A_n$ and $B_1...B_m$. The output assertion, O, contains m*n range sub-assertions $O_{11}...O_{mn}$ which are generated by applying the algorithm of part 1 to each possible pair of input range sub-assertions. Our proof is as follows:

First, chose two numbers (c,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. Also, d is accepted by at least one sub-assertion of B, B_y. The output assertion O contains a range sub-assertion O_{xy} which was generated via the algorithm in part 1 using A_x and B_y so O_{xy} accepts q where q = c + d (by part 1). Because O contains O_{xy}, O accepts q.

Second, pick a number q that is accepted by O. Hence, O contains a sub-assertion O_{xy} which accepts q. O_{xy} must be equal to the combination of A_x and B_y via the algorithm of part 1, and hence for any c and d which fulfill the property: q = c + d, c is accepted by A_x and d is accepted by B_y. Because A includes A_x and B includes B_y, A accepts c, and B accepts d.

Because the ranges of many of these range sub-assertions overlap, our implementation merges overlapping ranges before generating the final output assertion for efficiency reasons. To complete part 2, we must show that our merge operation results in an equivalent set of range sub-assertions.

4.2.1.3 Range Merging Lemma

Take two range sub-assertions, represented as the ranges for which they return true: rA= [Al,Au] and rB = [Bl,Bu] where Al <= Bl. rA and rB are merged as follows:

- if Bl <= Au <= Bu the resulting range, rO =[Al, Bu] (overlap)
- if Bu <= Au the resulting range, rO = [Al, Au] (enclosure)
- if Au <= Bl no merging takes place (no overlap)

See Figure 11 for graphical representations of these cases.

In case 1, the upper bound of the new range (rO) is equal to rB's upper bound, and greater than rA's upper bound. The lower bound is correspondingly equal to rA's lower bound and less than rB's lower bound. Hence, all numbers allowed by rA or rB are in the new range rO. In case 2, rO is equal to rA, because Al <= Bl, and Bu <= Au, all numbers in rA and rB are also in rO. In case 3, no merging takes place, so rA and rB are not affected.

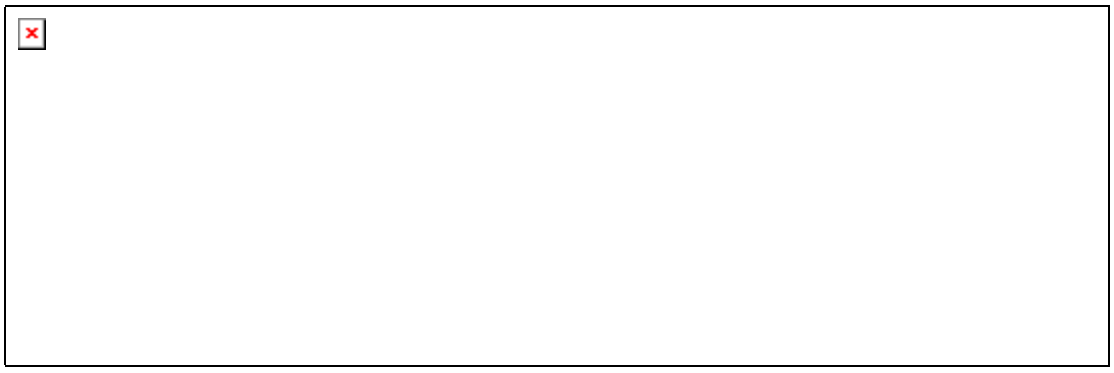


Figure 11: The three cases of range overlap, and their handling under the range merge algorithm.

These are the only three ways in which ranges will be merged and clearly, after the merge operation any number in the initial ranges will be in the output range(s). In case 1 and 2, the output range rO is converted into a range sub-assertion, otherwise (case 3) the original range sub-assertions are used. Hence, any numbers accepted by the initial range sub-assertions will still be accepted by the output range sub-assertion(s) included in the final output assertion O.

4.2.1.4 Applying the behavior of the assertion+ operator to the other arithmetical operators (-,*,/)

The behavior used by the assertion+ operator to generate all possible output range(s) for given range(s) of input values (with application of the addition operator) can be generalized to the other arithmetic functions. The only modifications that need to be made are the actual function used (e.g. replacement of addition with multiplication) and possible modifications to "special case" handling of values which may serve as range boundaries but are not handled by the system's arithmetic functions, such as positive and negative infinity. In our prototype implementation, this shared behavior was abstracted to a generic handle_continuous_function method which is passed code for the specific arithmetic operator to be used. This generic function was used directly to implement the assertion* and assertion/ operator, and indirectly (via the assertion+ operator, see Section 4.2.2.2 for details) for the assertion- operator. Other proofs in this chapter (assertion* and assertion/) will reference the proof for the assertion+ operator.

4.2.2 Proof of correctness of the assertion- operator

The assertion- operator has two forms, a unary minus and a binary minus.

4.2.2.1 Part 1: Proof of correctness of the unary assertion- operator

The unary assertion- operator takes as input an assertion A, and produces an output assertion O. For the assertion O to be correct, the following must hold: For all numbers N, if A(N) = TRUE, then O(-N) = TRUE. In other words, if the assertion A accepts a number N, the output assertion O must accept the unary negation of N. For an input assertion A, this is accomplished by negating the ranges accepted by the range sub-assertions when producing the output assertion O. For each range sub-assertion in A, which accepts a range rA= [Al,Au], the system replaces it with the negated range rO = [-

Au, -Al] before incorporating it into the output assertion O. For any value V which is within the range rA (specifically Al ≤ V ≤ Au) its negation will be within the output range (-Au ≤ -V ≤ -Al).

4.2.2.2 Part 2: Proof of correctness of the binary assertion- operator

Just as "A-B" can be written "A + (-B)", the binary assertion- operator "A assertion- B" is implemented as A assertion+ (assertion- B). We have just shown that "(assertion- B)" propagates correctly, and we previously showed that the assertion+ operator also propagates correctly. Hence, binary assertion- propagates correctly.

4.2.3 Proof of correctness of the assertion* operator

The assertion* operator is much like the assertion+ operator. In fact, the only difference in behavior (due to shared code) and the proof is that the possible bounds for the output range sub-assertions are calculated using the system's multiplication operator instead of the addition operator. So the MIN and MAX operations discussed in 4.2.1.1 choose from i1=Al*Bl, i2=Al*Bu, i3=Au*Bl, and i4=Au*Bu. We have again defined appropriate return values for multiplication operations involving positive and negative infinity, see Section 4.2.5.

To summarize the algorithm, the product of two ranges A and B is calculated as follows: A * B = [MIN{i1,i2,i3,i4}, MAX{i1,i2,i3,i4}]. A proof of correctness for this algorithm for taking the product of two ranges (or intervals) can be found in any textbook on interval arithmetic, such as [Alefeld and Herzberger 1983].

Once the base case is correct, the proof for the general case (of multiple range sub-assertions) is exactly the same as for the assertion+ operator, and we refer the reader to Section 4.2.1.2.

4.2.4 Proof of correctness of the assertion/ operator

The only difference between the assertion/ operator and the assertion* and assertion+ operators is the use of the system's division operator and handling of denominator ranges that contain zero. The assertion/ operator takes two arguments, assertions A and B. If the B assertion contains a range sub-assertion which accepts zero (a range including zero) the final output assertion O will contain a divide-by-zero error

sub-assertion, indicating the possible error. By pre-processing the range sub-assertions of assertion B which contain a zero, the system is able to make use of the same behavior (code) as the assertion* and assertion+ operator. Because the division function is not continuous when the divisor reaches zero, the theory of interval arithmetic does not hold unless the system splits ranges which contain zero into two separate ranges at the zero point.

The first range covers values that can be generated by divisors from the lower bound to zero, and the second range covers values that could be generated by divisors from zero to the upper bound. The “value” at the zero point itself is covered by the divide-by-zero error sub-assertion already mentioned.

By splitting the range sub-assertions in this manner, we are then able to use the same behavior (and proof) as the assertion+ and assertion* operators. Once again, the only modification is to use the system’s division operator when calculating i1-i4: i1=Al/Bl, i2=Al/Bu, i3=Au/Bl, and i4=Au/Bu. Refer to Section 4.2.1.2 for the description and proof for multiple range sub-assertions.

Figure 12 shows an example of the assertion/ operator at work. The range sub-assertion on cell B which ranges from [-2 5] is split into two ranges, [-2 0) and (0 5] and a divide-by-zero error is introduced. Because the two ranges on the denominator approach zero exclusively (recall that round brackets indicate exclusive bounds while square brackets indicate inclusive bounds) the resulting output range sub-assertion approaches negative and positive infinity.

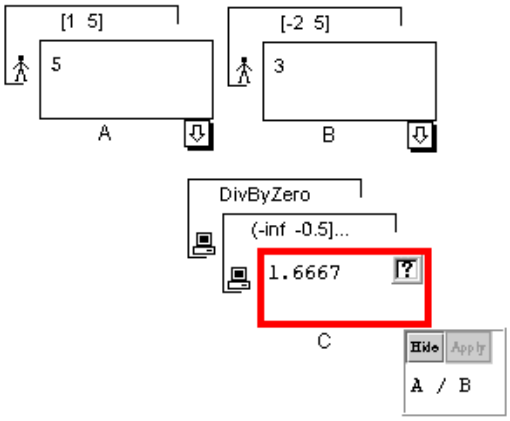


Figure 12: The results of propagating sub-assertions through a division operator. The range sub-assertion on cell C accepts numbers from negative infinity to -0.5, and then (continued off-screen) from 0.2 to positive infinity.

4.2.5 Special cases for arithmetic operators involving negative and positive infinity

Whenever possible, the Forms/3 built-in operators were used to perform calculations about ranges. However, because the upper and lower bound of ranges can represent infinite and exclusive values (approaching zero being the most notable) in addition to standard inclusive values, in some cases it was necessary to provide special case handling of certain arithmetic operations. For example, the Forms/3 addition operator would not be able to handle inputs of “INF+” and “5”, which should result in a positive infinite value. Multiplication by values that approach (but do not reach) zero, and division by infinite values are all examples of the types of special case handling needed. Table 1 specifies the results of these special operations. Note that cases which do not appear are either not applicable with the current implementation, or are automatically handled by the standard Forms/3 operators, such as the case of INF / 0, which is an error.

LEGEND:

INF	Negative or positive infinity
INF-	Negative infinity
INF+	Positive infinity
N	A number
N+	A positive number
N-	A negative number
Z-	A negative number approaching zero
Z+	A positive number approaching zero

Note: N, N- and N+ can indicate numbers approaching zero, but not infinite values.

MULTIPLICATION:

INF+ * N+	= INF
INF+ * N-	=INF-
INF- * N+	= INF-
INF- * N-	= INF
INF * 0	= 0
INF+ *INF+	= INF+
INF+ * INF-	= INF-
INF- * INF+	= INF-
INF- * INF-	= INF+
N+ * Z+	=Z+
N+ * Z-	=Z-
N- * Z+	=Z-
N- * Z-	=Z+

ADDITION:

INF- + N	= INF-
INF + N	= INF

DIVISION:

INF+ / INF+	= INF+
INF+ / INF-	= INF-
INF- / INF+	= INF-
INF- / INF-	= INF+
INF+ / N+	= INF+
INF+ / N-	= INF-
INF- / N+	= INF-
INF- / N-	= INF+
Z+ / N+	= Z+
Z+ / N-	= Z-
Z- / N+	= Z-
Z- / N-	= Z+
N+ / Z+	= INF+
N+ / Z-	= INF-
N- / Z+	= INF-
N- / Z-	= INF+
0 / INF	= 0
N+ / INF+	= Z+
N+ / INF-	= Z-
N- / INF+	= Z-
N- / INF-	= Z+

Table 1: Special cases handled by the assertion propagation algorithms.

4.2.6 Proof of correctness for the relational operators

In the current prototype, the relational operators are the only assertion-specific operators that generate Boolean sub-assertions. Recall that a Boolean sub-assertion is a function that is given a cell's value, and accepts (depending upon the specific sub-assertion) one of three Boolean values: T, F, or either (represented here as TF). In Forms/3, the relational operators (=,<,<=,>,>=) accept numerical inputs and return Boolean values. Hence, our assertion-specific operators look for range sub-assertions in their input assertions, and return Boolean sub-assertions in their output sub-assertions (along with any propagated error and/or no-value sub-assertions, see Section 3.2.3).

Each assertion-specific relational operator tests the range sub-assertion(s) on its inputs, and returns a Boolean sub-assertion as follows:

A and B represent the set of numbers accepted by the two input assertions, and C-op is the specific relational operator (=,<,<=,>,>=). The assertion-specific relational operators return the following Boolean sub-assertions:

- T iff $\forall$ a $\in$ A and $\forall$ b $\in$ B, a C-op b = TRUE.
- F iff $\forall$ a $\in$ A and $\forall$ b $\in$ B, a C-op b = FALSE.
- TF otherwise.

Because the system is using the actual relational operations on ranges directly as it would be using them on specific values, it is trivially obvious that assertions generated in this manner will accept the correct Boolean values.

For example, if the first range is [0 5] while the second is [6 10], and our relational operator is less-than (<), the system will return a Boolean sub-assertion of T, because the result of the comparison will always be true, regardless of the specific (valid) values taken on by the two input cells. Likewise, if the ranges were reversed, the system would always return a F Boolean sub-assertion, because the numbers allowed by the first range sub-assertion would always be larger than those allowed by the second. If the ranges were instead [0 5] and [5 10], the system would return a TF Boolean sub-assertion, because although the first is generally less than the second, it is possible for two values to be equal (and no longer strictly less-then) so both TRUE and FALSE

results are possible. Figure 13 in Section 4.2.8 shows an assertion specific relational operator producing a Boolean sub-assertion which is then used by the assertion specific IF operator when choosing assertions to propagate.

4.2.7 Proof of correctness for the logical Boolean operators

The logical operators (AND, OR, NOT) in Forms/3 operate on (and return) Boolean values. Hence, our assertion-specific versions look for Boolean sub-assertions in their input(s) and return Boolean sub-assertions (along with any propagated error and/or no-value sub-assertions) in their output assertion. The logical NOT is a unary operator, and as it only has three cases, we will explicitly state them: NOT(T)= F, NOT(F) = T, NOT(TF)=TF. The assertion-specific version of NOT simply negates the "always true" and "always false" case. When the input assertion will accept either true or false values, the output assertion will as well. (The system negates both the T and F, getting a F and T, which is represented as TF.)

The assertion-specific AND and OR operators work in a similar manner, but because they have two operands, we will define their behavior using the following notation: Let A and B be the set of possible Boolean values accepted by the two input assertions (T, F, or both), and L-op be the system's corresponding logical operator (AND,OR). The assertion-specific logical operators return the following Boolean sub-assertions:

- T iff $\forall$ a $\in$ A and $\forall$ b $\in$ B, a L-op b = TRUE.
- F iff $\forall$ a $\in$ A and $\forall$ b $\in$ B, a L-op b = FALSE.
- TF otherwise.

As with the relational operators, the logical assertion-specific operators make heavy use of their corresponding system defined logical operators, and it is trivially obvious that assertions generated in this manner will accept the corresponding values generated by the system logical operators.

4.2.8 Proof of correctness for the IF operator

The IF operator in Forms/3 evaluates one of two expressions based upon a Boolean expression (The second expression can be NIL, resulting in a value of “no-value” if evaluated). The Forms/3 IF returns a value as shown in Table 2.

IF(argA, argB, argC) =

argB	iff argA == True
arcC	iff argA == False
NoValue	iff argA == false AND argC == NIL (represents an else-less IF)

Table 2: Return values of the Forms/3 IF operator.

The assertion-specific IF operator looks for a Boolean sub-assertion in the input assertion A (call it binaryA) and also pulls out any error or no-value sub-assertions (call them EnvA) which will be propagated through the IF operator. It then returns an assertion as shown in Table 3.

assertionB OR EnvA	iff binaryA == T
assertionC OR EnvA	iff binaryA == F
assertionB OR assertionC OR EnvA	iff binaryA == TF
no-value OR EnvA	iff binaryA == false AND argC == NIL

Table 3: Return values of the assertion-IF operator.

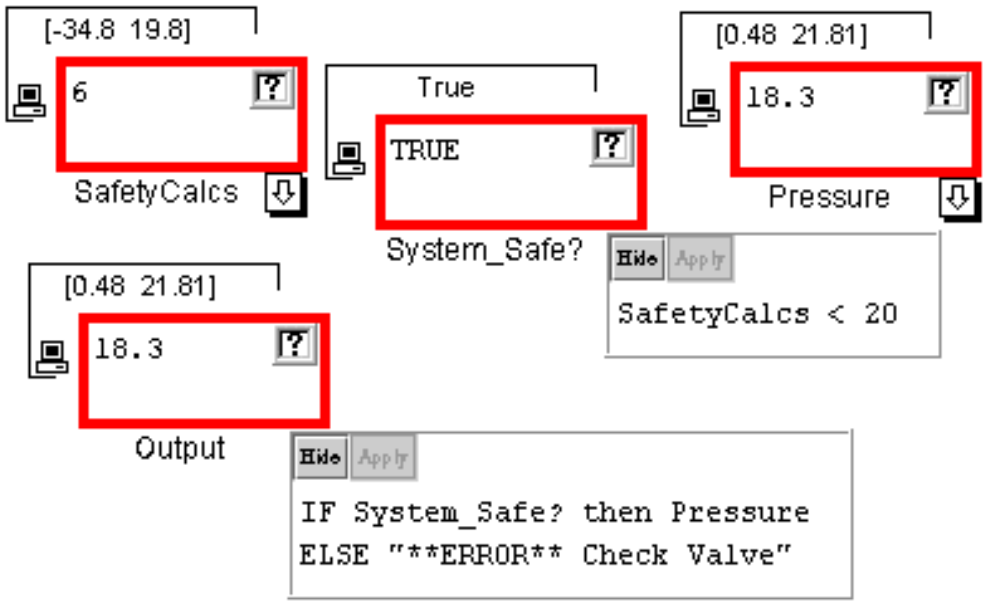


Figure 13: Because the range of possible values for the SafetyCalcs cell is below twenty, the System_Safe? cell will always have a true value (as indicated by the "True" Boolean sub-assertion displayed above it). The IF expression in the Output cell propagates only the assertion on the THEN expression (from the Pressure cell) because the predicate will always be true.

In other words, any errors or no-value sub-assertions in the predicate are always propagated to the resulting assertion. If the Binary sub-assertion of the predicate indicates a T or F value, only one of the two assertions (B or C) will be returned (See Figure 13), and if the predicate Binary sub-assertion indicates a TF value, the B and C assertions will be merged (logical OR) together and returned. When the Boolean sub-assertion takes on T and F value (indicating that the predicate can never have a different value) we can guarantee that the corresponding (B or C) expression will never be evaluated, and so the corresponding assertion can be discarded. The TF Boolean sub-assertion indicates that the predicate can take on either a true or false value, indicating that either of the (B or C) expressions can be evaluated, and that the IF expression as a whole can take on all values that can be generated by either expression. By combining the two assertions using a logical OR, we are guaranteed that the resulting assertion will accept all values calculated

by the Forms/3 IF operator. The EnvA element of the above formulas simply accomplishes the error and no-value propagation discussed in Section 3.2.3.

5. ALGORITHM COMPLEXITIES

The algorithms used to propagate assertions through formulas are polynomial. A discussion leading to worst case scenarios will be presented in Section 5.2, and Section 5.3 will present a worst case complexity analysis. In normal usage, as seen in testing and an initial protocol analysis, the algorithm propagation engine adds $O(N^2)$ to the formula editing process, where N is the number of cells in the spreadsheet. This is due to the fact that the system normally must evaluate formulas to provide values for immediate visual feedback after each formula edit. Because the assertion propagation engine follows the same evaluation path when propagating assertions, the only additional cost is due to the shared-dependency check. The relationship between evaluation in Forms/3 and the assertion propagation algorithms will be discussed in Section 5.1. Table 4 lists the major variables used in this Chapter. The following discussion of the complexity of the algorithms involved is useful for understanding the scalability of assertion propagation. In the existing prototype, response time for all the cases we have tried has been

N - Number of cells in the spreadsheet
F - Number of operations in the largest formula (formula length)
A - Number of sub-assertions in the largest assertion (assertion size)
X - A convenience upper-bound, by definition: X > N, X > F, and X > A.
C - A constant value, used to bound variables.

Table 4: List of variables used in Chapter 5.

immediate.

5.1 NORMAL USAGE COST OF ASSERTION PROPAGATION

The discussion in this Section makes the assumption that A, the maximum size of an assertion, is constant bounded, which effectively removes it from the complexity analysis. This assumption will be discussed in Section 5.2 and removed in Section 5.3.

When a formula (or constant) is changed due to a user edit, Forms/3 must propagate this change to all affected cells to provide the user with immediate visual feedback of their actions. To do this, the values of all cells affected by the change are re-calculated, by evaluating their formulas[1]. The propagation of assertions is done in a similar fashion. Because assertions are propagated using a formula evaluation model, many of the actions performed by the assertion propagation engine mirror those taken by the standard evaluation engine in Forms/3. In fact, there are only two steps taken by the assertion propagation engine that go beyond the work already done by the evaluation engine:

The first additional step is to check each formula for shared dependencies. Because this check follows cell references recursively until it finds a shared dependency or exhausts the reference tree, it adds worst case complexity of O(N), where N is the number of cells in the spreadsheet, for each checked cell. In the current implementation, the shared-dependency check is run on every cell which has an assertion propagated to it, resulting in a potential of $O(N^2)$ time complexity.

The second additional step is the formula translation, which replaces the standard operators in the formulas with assertion specific operators, and converts the arguments into assertions. The operator replacement is a simple hash-table access with order O(1), converting cell references to the assertion on that cell is also an order O(1) data structure lookup, and the conversion of numeric constants to an assertion is also order O(1). Thus,

---

[1] Forms/3 is a lazy language, meaning that values of off-screen cells do not have to be immediately updated or displayed. Because the assertion propagation engine is currently eager we assume that Forms/3 is eager for this analysis. This makes no difference when all cells are on-screen (such as when the user is working on spreadsheet consisting of a single form). See Section 6.3 for details of how assertion propagation could be modified to work in a lazy manner, in which case this discussion generalizes to multi-form spreadsheets with off-screen cells.

the formula translation step adds O(1) to the standard value propagation which Forms/3 already performs.

Hence, if the assertion-specific operators were of the same order as their general value-operator counterparts, the time used by the assertion-propagation engine could be folded into the time used by the standard evaluation engine while adding only the complexity inherent in the mutual-dependency check, specifically $O(N^2)$.

In the normal case (as we have seen during development and a protocol analysis) the arguments given to assertion-specific operators are such that they are of the same order as their general value-operator counterparts. Sections 5.2 and 5.3 will cover the analysis of worst case scenarios which could potentially make the assertion propagation algorithm add more than $O(N^2)$ to the work already performed by the system after a formula edit.

5.2 CAUSES OF WORST CASE COMPLEXITIES

In Section 5.1 we discussed the complexities of the assertion propagation with one simplifying assumption. Our assumption was that the assertion specific version of operators would have the same time complexity as their value versions. For example, if the addition operator (+) required Q sub-operations to calculate the result of an addition, the assertion specific addition operator (assertion+) would require less than C*Q sub-operations, where C is a constant.

Obviously, assertion specific operators are performing more work than their standard value counterparts. This is due to the fact that they are working abstractly, (e.g. computing interval arithmetic as opposed to standard value arithmetic) and due to the extra decoding and encoding work required by the more complicated data structures used to represent assertions. In Section 5.1 we assumed that this extra work was bounded by a constant, because as long as the size of the assertion arguments (meaning, the number of sub-assertions they contain) is bounded by a constant, there exists a constant upper bound C on the amount of work done by the assertion propagation operators.

This simplifying assumption breaks down when arguments to assertion specific operators (the assertions) contain a large number of sub-assertions that are not bounded by a constant. With the current implementation, this problem could only exhibit itself

with range sub-assertions. This is due to the fact that each assertion can have only a single Boolean, Error, or NoValue sub-assertion, while theoretically an assertion could have an infinite number of range assertions (e.g. [0 1] [2 3] [4 5] ...). As the number of hypothetical range sub-assertions grows, so does the computation required to produce output ranges. By examining the algorithm of Section 4.2.1 (which is used in some form for each of the arithmetic operators) this operation is $O(A^2)$ where A is the number of sub-assertions.

There are two ways in which A, the maximum number of sub-assertions in an assertion, can increase enough to break our simplifying assumption. First, the end user could place a large number of ranges on a cell. We do not expect this to happen in normal usage. Specifically, we expect end-users to place a manageable number of requirements on a cell, as opposed to entering a few thousand different ranges on a cell.

The other way in which A could grow to a significant size is for the system to produce more sub-assertions as part of an output assertion than it received in the input assertions. The division operator will split a range which contains zero, and all of the arithmetic operators can potentially produce more range sub-assertions in the output assertion than were present in either of the two input assertions alone. For example, two assertions, each containing two range sub-assertions, [1 2] [100 150] and [4 6] [8 10] respectively, can be multiplied to produce an assertion containing three range sub-assertions: [4 12] [16 20] [400 1500]. It could be possible to construct a pathological example in which the number of range sub-assertions would grow exponentially. Again, we do not expect this to happen in normal usage. In part this is due to our assumption that end-users will place a few large range restrictions on input cells, as opposed to multiple small ones. Additionally, as the number of range sub-assertions increase, and the number of operations which process them increases, it becomes statistically more likely that output ranges will overlap and be merged together. Section 5.3 presents an analysis of the worst case complexities that result when we remove the simplifying assumption of Section 5.1.

5.3 ANALYSIS OF WORST CASE COMPLEXITIES

In normal operation, when a formula is changed in Forms/3, the system must recalculate the cell's value. We use F to represent the number of operations in the largest formula in the program. Obviously, the number of operations that much be performed to calculate the new value is bounded by F. Additionally, if the updated cell affects other cells, their values must also be recalculated. N represents the number of cells in the program, so theoretically, it may be necessary to do N*F operations. (In most spreadsheet languages, the maximum length of a formula, and hence F, is constant bounded.)

Due to the possibility of large assertions (large in that they contain many range sub-assertions) each of our F (assertion-specific) operators may take $O(A^2)$ time, where A is the maximum number of sub-assertions in an assertion. This leads to a complexity of $O(NFA^2)$.

Finally, a shared-dependency check must be performed for each cell, which is O(N) per cell, or $O(N^2)$ overall. It is important to note that the work performed by the shared dependency check does not depend upon the actual assertions or their sizes, so the N^2 is added to and not multiplied by the work performed by the actual assertion calculation, giving: $O(NFA^2 + N^2)$.

Unfortunately A and F can not be defined in terms of N. To reduce to a single variable, X is defined as larger than N, F and A to produce $X^4 + X^2$ which is $O(X^4)$. Hence, worst case complexity is $O(X^4)$ where X is the largest of N, F, and A.

6. FUTURE WORK

This work introduces assertions and assertion propagation to an end-user programming system. As with any research, there are many areas which could be improved and open questions remaining. Improvements could be made to the assertion propagation algorithms to allow the propagation of assertions through a wider variety of formulas and to improve their efficiency. Additionally, improvements could be made to the collaborative aspects of the entire system, and there are many aspects of the Forms/3 programming language which are not yet supported with assertions, such as temporal programming and grids. The following four sections touch upon these issues, noting possible directions of future work and open questions.

6.1 IMPROVEMENTS IN PROPAGATION

As shown in Section 3.3, the current assertion propagation mechanism is unable to propagate assertions through formulas with shared dependencies. In some cases, this problem could be resolved by formula substitution and symbolic evaluation. For example, the formula for the Painted_Gizmos cell of Figure 9 could be simplified to remove the shared dependencies as shown in Figure 14. This simplified formula could

a) Blue_Gizmos + Red_Gizmos
b) (Gizmos x (1 - Percent_Red)) + (Gizmos x Percent_Red)
c) (1 x Gizmos) - (Gizmos x Percent_Red) + (Gizmos x Percent_Red)
d) Gizmos

Figure 14: This figure shows how the original formula (a) of the Painted_Gizmos cell can have references substituted (b), and through symbolic evaluation (c,d) be simplified to remove shared dependencies.

then be used to generate the appropriate assertion.

Figure 15 demonstrates another aspect of shared dependencies, using the IF operator. In this example, all three arguments (the predicate, the then clause, and the else clause) to the IF operator are dependent upon a single cell, resulting in shared

dependencies. Because of these shared dependencies, it is necessary for the system to modify the assertion on cell a before using it in the then and else clauses. (In this example the range must be split into [0 25] and [-25 0] for the then and else clauses, respectively.) Determining how to make these modifications in the general case is an open problem.

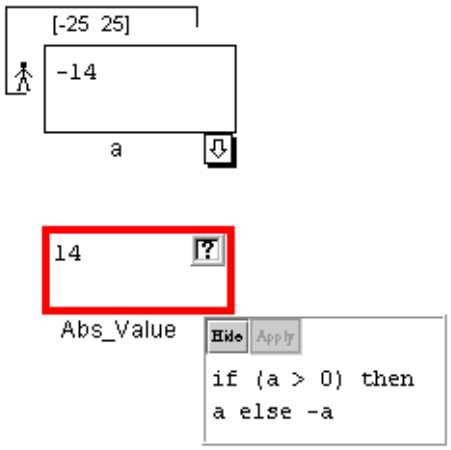


Figure 15: A simple example of the difficulties presented by the IF operator.

Currently, only positive assertions (e.g. “the value of the cell is between zero and five”) are supported. In the future we may wish to support negative assertions (e.g. “the cell can have any value except zero”). For the currently supported assertion types (numerical ranges, Boolean values, error conditions) negative assertions can easily be translated into their positive counterparts. For example, “The cell can have any value except zero” is logically the same as the range from negative to positive infinity which skips zero. However, this may not be the case if additional assertion types are added in the future, and would have to be dealt with at that point. Additional protocol analysis would have to be done to determine if end-users are able to cope with having their assertions translated by the system from negative to positive form, or if they would require support for display and editing of negative assertions, even if the system is doing translation behind the scenes.

In addition to improvements made to the forward propagation mechanism, in some cases it should be possible to propagate user specified assertions backwards, against the data-flow path. This “reverse propagation” could potentially leverage user specified assertions placed on output cells into additional system generated assertions on intermediate or input cells.

6.2 IMPROVEMENTS IN COLLABORATION

Currently, the only way for a user to add an assertion to a Forms/3 spreadsheet is to actively click on the "Guard" button and manually specify the assertion. We envision a complete software engineering system that collaborates with the user by being able to suggest possible assertions, and draw the user into dialog that results in assertions being placed on cells. This collaboration would be integrated with other software engineering methods in Forms/3.

Cells (with a non-constant formula) in Forms/3 have a testing check-box in their top right corner (see cell Abs_Value in Figure 15 for an example) that allow the user to interact with the Forms/3 testing system [Rothermel et al. 1998, 1999, 2000], and the fault localization system [Reichwein et al. 1999]. The testing system indicates the amount a cell has been tested by coloring its border somewhere in the range between red (untested) and blue (fully tested). By clicking in a testing check-box with a question mark, the user indicates that they feel the value in that cell is correct (they have validated a test case). If they feel the value is incorrect, they can right-click in the same testing check-box to place an "X" mark, indicating that the value is incorrect. If the user indicates that a value is incorrect, the fault localization system highlights cells that contribute to the problem value [Reichwein et al. 1999]. If a user wishes to fully test a cell or entire spreadsheet, but is unable to determine a specific test case that still needs to be covered, they can use the "Help-Me-Test" feature of Forms/3, which will attempt to generate input values that exercise a case that has not yet been tested [Cao 2000].

The assertion propagation system can both provide useful information to, and receive useful information from, these other software engineering features of the Forms/3 environment. The "Help-Me-Test" sub-system is more efficient when it has bounds on possible values for input cells, information which can be provided by range sub-assertions. And when the user indicates that the value in a cell is incorrect, the system can draw them into a dialog which asks for further explanation. This dialog will hopefully result in a new assertion for that cell. By keeping a history of correct values, and comparing them to the incorrect value, the system may even be able to suggest an initial assertion, which could then be approved of, or modified by, the user.

6.3 KEEPING FORMS/3 LAZY

When a formula (or constant) is changed due to a user edit, Forms/3 must propagate this change to all affected on-screen cells to provide the user with immediate visual feedback of their actions. To do this, the values of all on-screen cells affected by the changed cell are re-calculated, via formula evaluation. However, the value of an off-screen cell may not need to be re-calculated. If the off-screen cell does not affect any on-screen cells, and its value is not otherwise needed by the system, Forms/3 will postpone evaluating that cells' formula until the value is needed. This behavior is what makes the Forms/3 evaluation engine "lazy".

However, the current assertion propagation prototype is not "lazy". Currently, if it is possible to generate an assertion for a cell (even if that cell is off-screen) the assertion propagation mechanism will do so. Because of this, even though the Forms/3 evaluation model is lazy, the system as a whole is not, because it may do work (assertion propagation) before the results are needed. On spreadsheets that consist of single (on-screen) forms, this is a non-issue. However, for larger spreadsheets that are made up of multiple forms (one or more of which are off-screen), this could result in the system performing unnecessary calculations. One way to fix this problem is as follows: The system simply does not propagate assertions to cells that are off-screen and do not affect on-screen cells. Additionally, when a form is brought on-screen, assertions are propagated to the cells on it before they are displayed.

This solution would effectively return the Forms/3 system to a lazy state, but it is not without side effects. With the current (eager) assertion propagation mechanism, the system is able to guarantee that it will detect assertion conflicts at the earliest possible moment. With the lazy mechanism above, it is possible for assertion conflicts that occur on an off-screen cell (due to the user editing an on-screen cell) to go undetected until the cell where the conflict occurs is brought on-screen. We feel that it is preferable to warn the user of a conflict immediately after the program edit that caused the conflict. It is an open question if a lazy assertion propagation mechanism can be developed that will retain the ability to detect an assertion conflict immediately. It may even be possible for eager assertion propagation to co-exist with lazy evaluation, although we are unsure as to the modifications that would be needed make such an approach scalable.

6.4 SUPPORTING TEMPORAL AND REGION-BASED PROGRAMMING

Forms/3 supports temporal programming, in which formulas can reference "earlier" and "later" values of cells [Burnett et al. 2000]. For example, a formula such as "aCell<T-1>" references the value of aCell one "tick" earlier in time. Additionally, a cell can have different formulas at different points in time. The user can move the system clock forward and backward in time, and cell values will update themselves based upon the current system time. Currently, formulas which index into time are not supported by assertion propagation, and assertions do not update themselves based upon the system clock (they are eternal).

It is an open question whether the user should be able to specify assertions that change over time, or whether assertions should be unchanging through time. If we assume that there should not be support for having user specified assertions change over time (leave assertions eternal, as they now are) the only change that would need to be made to support assertion propagation is to develop support for the temporal reference operator. This would be very similar to the current reference operator (except that it would have to reference a cell at a different point in time) and should not pose much difficulty.

In addition to temporal programming, Forms/3 supports the display and calculation of structured data in grids which bear a resemblance to the layout of traditional spreadsheets cells. Grids in Forms/3 are divided into regions, which can encompass an entire grid or be as small as a single cell. The cells in a region use a shared formula to calculate their value. See Figure 16 for an example, where the data input cells are individual regions, and the cells that calculate the result make up a region and share a single formula. It would be simple to add support for assertion propagation to grids by simply treating each grid cell as a single cell. There would be a GUI issue to contend with, that of displaying the assertions in the limited space around grid cells, but no modifications to the actual propagation algorithms would be required.

However, in Forms/3 grids (and the regions of which they are made) were developed to assist the user and give the system a scalability advantage. By reasoning about regions instead of individual cells, the Forms/3 testing system can sometimes leverage a small amount of user work validating individual cells into coverage for all the

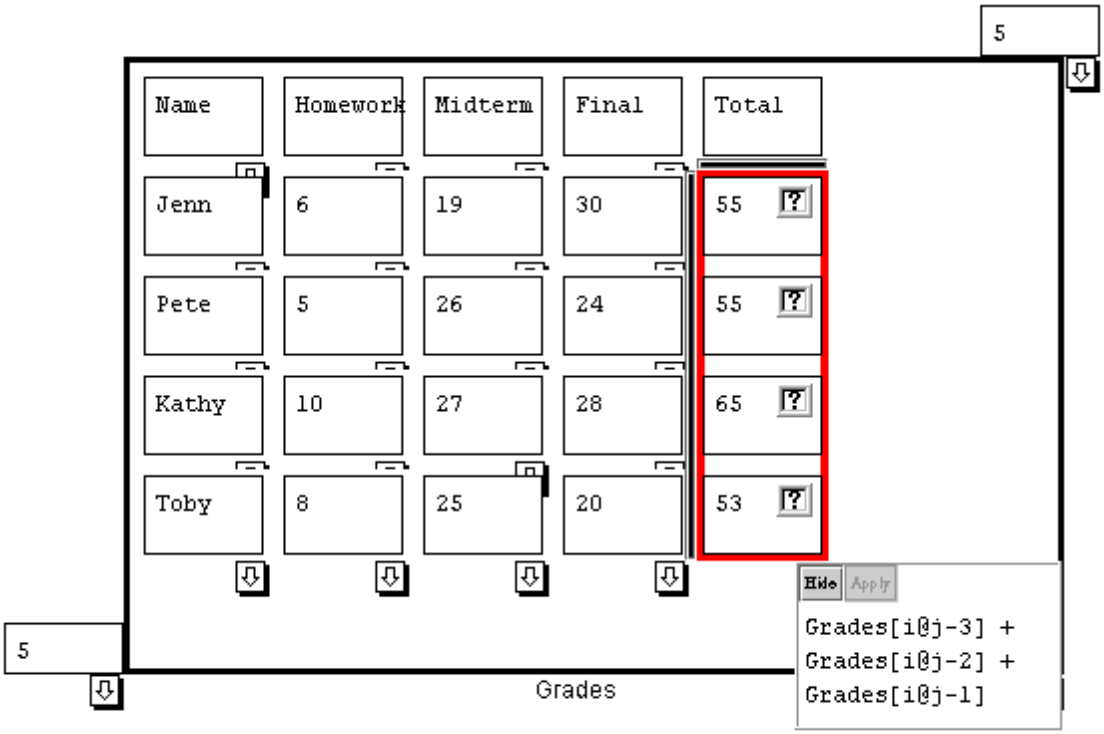


Figure 16: A grid cell in Forms/3. The four cells on the lower right share a formula (displayed).

cells in a region [Burnett et al. 2001b]. Additionally, the testing system gains a large degree of scalability by reasoning at the level of regions, which may include hundreds or thousands of individual cells. When extending assertions to work with grids, we hope to be able to duplicate some of these past successes of the testing methodology. Ideally, we will be able to reason about assertions in grids on the level of regions instead of individual cells.

7. CONCLUSION

Our overall goal is to develop methodologies that assist end-users in building spreadsheets. Research shows that a significant number of spreadsheets (20%-40%) created by end-users contain errors [Teo and Tan 1997; Panko 1995; Panko 1998]. In an

attempt to reduce this error rate, we wish to give end-users some of the benefits of professional software engineering practices, without requiring them to undergo formal training in software engineering.

This work focuses on developing a system that collaborates with users, using assertions, to improve spreadsheet reliability in an end-user programming environment. In addition to the traditional benefits of assertions gained by professional programmers, namely dynamic error checking and the documentation of programmer assumptions, this system deductively propagates the implications of user assertions, giving end-users two benefits that are the main contribution of this work.

The first benefit is that this propagation allows the system to cross-check assertions entered by the user against cell formulas. This cross-checking of program logic allows the end-user to specify their program in two separate ways. In addition to the more traditional formulas which tell the system how to calculate a cell's value, the user can specify valid ranges (limits) for values. This allows the system to cross-check the user's logic, and point out problems which may indicate errors.

The second benefit that assertion propagation provides the end-user is additional immediate visual feedback about the range of behavior of their code. The traditional display of prototypical values in spreadsheets displays one (current) value to the user. Assertion propagation evaluates and displays all the possible values the formula could generate and gives the user another method for understanding the behavior of their spreadsheet.

In addition to the development of algorithms for the propagation of assertions, their correctness proofs, and complexity analysis (Chapters 3, 4 and 5), this work includes a prototype implemented for the Forms/3 research language. The prototype demonstrated that the algorithms can support immediate visual feedback when running on a standard desktop computer, and allowed an initial protocol analysis performed by Christine Wallace. This protocol analysis indicated that end-users were able to understand the behavior of the assertion propagation algorithms, and in some cases gain extra understanding of their programs due to the propagated assertions. Future work is planned for a full scale end-user study, which we expect will show that users of this methodology produce spreadsheets with significantly fewer errors.

BIBLIOGRAPHY

[Abramsky and Hankin 1987] Abramsky, S. and Hankin C. Abstract Interpretation of Declarative Languages. Ellis Horwood, Chichester, 1987

[Ambler 1999] Ambler, A. The Formulate Visual Programming Language. Dr. Dobb's Journal, August 1999. 21-28

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

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

[Bjørner et al. 1997] Bjørner, N., Browne, A., and Manna, Z. Automatic Generation of Invariants and Intermediate Assertions. *Theoretical Computer Science*, 173(1):49-87, February 1997.

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

[Brown and Gould 1987] Brown, P. and Gould, J. Experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems* 5, 1987, 258-272.

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

[Burnett et al. 2000] Burnett, M., Cao, N., and Atwood, J. Visual programming in time vs. visual programming in space, TR #00-60-02, Oregon State University, February 2000.

[Burnett et al. 2001a] Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J. and Yang, S. Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm, *Journal of Functional Programming 11(2)*, March 2001,

155-206.

[Burnett et al. 2001b] Burnett, M., Sheretov, A., Ren, B., Rothermel, G. Testing Homogeneous Spreadsheet Grids with the 'What you See Is What You Test' Methodology, *IEEE Transactions on Software Engineering,* (to appear)

[Cao 2000] Mingming Cao. Automatic Test Case Generation for Spreadsheets. Masters Thesis, Oregon State University June 27th, 2000.

[Cook et al. 1999] Cook, C., Rothermel, K., Burnett, M., Adams, T., Rothermel, G., Sheretov, A., Cort, F., Reichwein, J. Does immediate visual feedback about testing aid debugging in spreadsheet languages? TR #99-60-07, Oregon State University, March 1999.

[Cousot P. and Cousot R. 1977] Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. 4th annual ACM Symposium on Principles of Programming Languages, 1977, 238-252.

[Cragg and King 1993] Cragg, P. and King, M. Spreadsheet modeling abuse: An opportunity for OR? *Journal of the Operational Research Society* 44(8), 1993, 743-752.

[Curcio 1998] Curcio, I.D.D. ASAP: A simple assertion pre-processor. *SIGPLAN Notices 33(12)*. December 1998, 44-51.

[Cypher and Smith 1995] Cyper, A. and Smith, D.C. "KidSim: End User Programming of Simulations", in Proc. ACM Conference on Human Factors in Computing Systems CHI'95, Denver, Colorado, 27-36.

[Djang et al. 2000] Djang, R., Burnett, M. and Chen, R. Static type inference for a first-order declarative visual programming language with inheritance, *Journal of Visual Languages and Computing*, April 2000 191-235.

[Ernst et al. 1999] Ernst, M., Cockrell, J., Griswold, W. and Notkin, D. Dynamically discovering likely program invariants to support program evolution. *International*

Conference on Software Engineering, Los Angeles, California, May 1999, 213-224.

[Heger et al. 1998] Heger, N., Cypher, A., and Smith, D.C., "Cocoa at the Visual Programming Challenge," Journal of Visual Languages and Computing, vol. 9, no. 2, 151-69, April1998.

[Hendry and Green 1994] Hendry, D. and Green, T. Creating, comprehending, and explaining spreadsheets: A cognitive interpretation of what discretionary users think of the spreadsheet model. *Int. J. Human-Computer Studies,* 40(6), 1994, 1033-1065.

[Jeffords and Heitmeyer 1998] Ralph Jeffords and Constance Heitmeyer. Automatic generation of state invariants from requirements specifications. *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, Orlando, Florida, November 3-5, 1998, 56-69.

[Kahn 1996] K. Kahn, "ToonTalk - An Animated Programming Environment for Children," *Journal of Visual Languages and Computing*, 197-217, June 1996.

[Luckham 1985] Luckham, D. C. and von Henke, F. W. An overview of Anna, a specification language for Ada. *IEEE Software* 2. March 1985, 9-23.

[Luckham 1990] Luckham, D. C. *Programming with Specifications: An Introduction to Anna, a Language for Specifying Ada Programs.* New York: Springer-Verlag, 1990.

[McDaniel and Myers 1999] McDaniel R. and Myers, B. Getting more out of programming-by-demonstration, *ACM Conference on Human Factors in Computing Systems (CHI'99)*, Pittsburgh, PA, May 15-20, 1999, 442-449.

[Moore 1966] Moore, R. Interval Analysis. Prentice-Hall, Inc. Englewood Cliffs, N.J., 1966.

[Ostroff 1989] Ostroff, J. *Temporal Logic For Real-Time Systems.* Research Studies Press LTD., Taunton, Somerset, England, 1989.

[Panko 1995] R. Panko, "Finding spreadsheet errors: Most spreadsheet models have

design flaws that may lead to long-term miscalculation," *Information Week,* May 29, 1995, 100.

[Panko 1998] R. Panko, "What we know about spreadsheet errors", *Journal of End User Computing,* Spring 1998.

[Perrone and Repenning 1998] Perrone, C., and Repenning, A., Graphical Rewrite Rule Analogies: Avoiding the Inherit or Copy & Paste Reuse Delemma. In Proceedings of the 1998 IEEE Symposium of Visual Languages, Nova Scotia, Canada, 1998, 40-46.

[Reichwein et al. 1999] Reichwein, J., Rothermel, G. and Burnett, M. Slicing spreadsheets: an integrated methodology for spreadsheet testing and debugging, *Conference on Domain Specific Languages*, Austin, Texas, October 3-5, 1999.

[Repenning and Summer 1995] Repenning, A. and Summer, T. Agentsheets: A medium for creating domain-oriented visual languages. IEEE Computer, 28(3), March 1995.

[Rosenblum 1995] Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering* 21(1), Jan. 1995, 19-31.

[Rosenblum et al. 1986] Rosenblum, D. S., Sankar, S. and Luckham, D. C. Concurrent runtime checking of Annotated Ada programs. *Proc. 6th Conf. on Foundations of Software Technology and Theoretical Computer Science.* New York, Springer-Verlag (Lecture Notes in Computer Science No. 241), Dec. 1986, 10-35.

[Rothermel et al. 1998] Rothermel, G., Li, L., DuPuis, C. and Burnett, M. What you see is what you test: A methodology for testing form-based visual programs. *International Conference on Software Engineering*, Apr. 1998.

[Rothermel et al. 1999] Rothermel, G., Burnett, M., Li, L., DuPuis, C., and Sheretov, A. A methodology for testing spreadsheets, Oregon State University TR 99-60-02, January 1999.

[Rothermel et al. 2000] Rothermel, K., Cook, C., Burnett, M., Schonfeld, J., Green, T.R.G., Rothermel, G., WYSIWYT Testing in the Spreadsheet Paradigm: An Empirical Evaluation. *International Conference on Software Engineering*, Limerick, Ireland, June 2000, 230-239.

[Sankar 1991] Sankar, S. Run-time consistency checking of algebraic specifications. *4th Software Testing, Analysis and Verification Symposium* (ACM SIGSOFT), Oct. 1991, 123-129.

[Sankar 1993] Sankar, S. and Mandal, M. Concurrent runtime monitoring of formally specified programs. *Computer* 26. March, 1993, 32-41.

[Teo and Tan 1997] Teo, T. and Tan, M., Quantitative and qualitative errors in spreadsheet development, *30th Hawaii International Conference on System Sciences*, Wailea, Hawaii, January 1997, Part 3, Vol. 3, 149-155.

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

[Wilcox et al. 1997] Wilcox, E., Atwood, J., Burnett, M., Cadiz, J., and Cook, C. Does continuous visual feedback aid debugging in direct-manipulation programming systems? *Proceedings CHI '97: Human Factors in Computing Systems*, Atlanta, GA, March 1997.