A Type System Based on End-User Vocabulary*

Robin Abraham and Martin Erwig and Scott Andrew
School of EECS, Oregon State University
[abraharo|erwig|andrewsc]@eecs.oregonstate.edu

Abstract

In previous work we have developed a system that automatically checks for unit errors in spreadsheets. In this paper we describe our experiences using the system in a workshop on spreadsheet safety aimed at high school teachers and students. We present the results from a think-aloud study we conducted with five high school teachers and one high school student as the subjects. The study is the first ever to investigate the usability of a type system in spreadsheets. We discovered that end users can effectively use the system to debug a variety of errors in their spreadsheets. This result is encouraging given that type systems are difficult even for programmers. The subjects had difficulty debugging "non-local" unit errors. Guided by the results of the study we devised new methods to improve the error-location inference. We also extended the system to generate change suggestions for cells with unit errors, which when applied, would correct unit errors. These extensions solved the problem that the study revealed in the original system.

1. Introduction

The benefits of types systems are well known. The fact that type checkers can automatically detect erroneous parts of a program makes them a helpful tool to develop correct programs. Although the effectiveness of type checkers in preventing errors has been previously shown for general-purpose languages [15], the usability of type checkers for end-user languages has not been investigated so far. In any case, many programmers (especially beginners) find type systems more of a hinderance than help. The main reason behind this perception of type systems is the fact that type error messages generated by compilers are hard to understand—the cause of the type error might not be obvious from the error message, and the corrective action even less so. This problem poses a particular challenge for type systems for end-user languages. In our work on helping end users develop more reliable software, we are looking at ways to bring the power of type systems to the domain of spreadsheets, and make the benefits accessible to end users.

We have approached the task of creating an end-user type system for spreadsheets by employing the idea of spreadsheet-specific type definitions, that is, the labels used in the spreadsheet are employed as headers for the data cells. A rule system that defines how headers can be combined for formula cells allows us to then identify formulas that are inconsistent with the header information [8]. The perceived advantage of this approach was that it would be easier for the users to understand the errors in the spreadsheet if the headers they themselves had entered were used while reporting errors. We realized early on that for the system to be successful and usable, it should only require minimal effort from the user. With this goal in mind we have developed the algorithms for header inference through spatial analyses described in [1]. The resulting system, UCheck, automatically detects and reports errors in spreadsheets [4].

We describe related work in the next section. In Section 3, we describe how UCheck works using examples from unit errors seeded in the spreadsheets used in the preliminary evaluation described in Section 4. In Section 5, we describe the enhancements done to the system so that it can infer change suggestions for correcting unit errors. We present conclusions and directions for future research in Section 6.

2. Related Work

Empirical studies have demonstrated the defect-detection capabilities of static type checking [10, 15]. Even though the benefits of type systems are widely accepted, not many studies have been carried out to test or compare the usability of type systems in general-purpose programming languages. Many researchers have looked at errors in programs developed by novice programmers [19, 18, 7]. But the frequency and impact of type errors in novice programs is not known.

Physical units (meters, grams, seconds, etc.) have been used to detect unit errors in general-purpose programming languages [11, 12], and in spreadsheets [6]. The approach originally proposed in [8] uses headers used by end users to label the data in their spreadsheets as implicit unit declarations, and carries out consistency checking based on the allowed combinations of units. The system described in [5] extends the approach but requires the users to annotate the spreadsheets cells with the unit information. To minimize the effort required of the user, we have developed the UCheck system [4] that infers the headers automatically [1] and uses the rule system from [8] to carry out consistency checking of spreadsheets.

The "What You See Is What You Test" (WYSIWYT) approach presented in [17] helps users test spreadsheets. The system uses data-flow adequacy and coverage criteria

*This work is partially supported by the National Science Foundation under the grant ITR-0325273 and by the EUSES Consortium (`http://EUSESconsortium.org`).

to give the user feedback on how well tested the spreadsheet is. The fault-localization mechanism in WYSIWYT [14] uses cell shading to guide the users while they are debugging faults uncovered by their test cases.

The approach presented in [5] also uses header information for consistency checking of spreadsheets. Checking of spreadsheet formulas based on the actual physical or monetary units has been presented in [6]. Both these approaches require the user to annotate the spreadsheet cells with extra information to aid the consistency checker. This activity could be very time consuming and error prone in the case of large spreadsheets. UCheck, on the other hand, infers the header information automatically and carries out unit checking without the need for any extra effort from the user.

In other previous work, we have developed a type checker for spreadsheets [3]. In addition to type checking the spreadsheet formulas and the arguments along the lines of more traditional approaches to type checking, our system also exploits information about the spatial arrangement of cells for consistency checking.

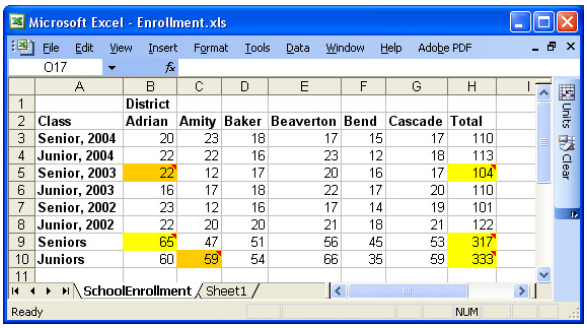
3. Unit Errors in Spreadsheets

From our observations of users in real life we have found that users enter headers within their spreadsheets to label the data. For example, in the spreadsheet shown in Figure 1, the header Adrian in B2 indicates that the data in column B is somehow related to "Adrian", which is in turn a "District" (as indicated by the header District in B1). Headers serve as documentation to help the user remember what the data means.

In the context of the spreadsheet[1] shown in Figure 1, the number 20 in cell B3 is not simply an integer. The row (Senior, 2004) and column (Adrian) headers tell us that the cell contains the number of students registered in the senior class of 2004 in the Adrian school district. We call headers in their function as labels *units*. We also see that Senior, 2004 in A3 has Class as its header, and Adrian in B2 has District as its header. These header dependencies give rise to what we call *dependent units*, which in this case are Class[Senior, 2004] and District[Adrian]. Since both the row and column headers apply at the same time, we combine the inferred dependent units using the *and operator* (&) to give the *and unit* District[Adrian]&Class[Senior, 2004] for the number 20 in B3. This inferred unit is treated as an implicit type declaration for the cell B3. The units for the other cells that contain data values can be inferred along similar lines.

The units obtained for the data cells are then used to infer the units for the formula cells depending on the operations in the formulas. For example, cell B9 contains the formula SUM(B3,B5,B7). Its unit is inferred as a combination of the units of the cells participating in the operation. All three cells have District[Adrian] as a common factor from the column-level header. The components from the row-level

[1]This spreadsheet was used in the second task of the study described later in this paper.



	A	B	C	D	E	F	G	H
1		District						
2	Class	Adrian	Amity	Baker	Beaverton	Bend	Cascade	Total
3	Senior, 2004	20	23	18	17	15	17	110
4	Junior, 2004	22	22	16	23	12	18	113
5	Senior, 2003	22	12	17	20	16	17	104
6	Junior, 2003	16	17	18	22	17	20	110
7	Senior, 2002	23	12	16	17	14	19	101
8	Junior, 2002	22	20	20	21	18	21	122
9	Seniors	65	47	51	56	45	53	317
10	Juniors	60	59	54	66	35	59	333
11								

Figure 1. Unit errors in the enrollment spreadsheet as reported by UCheck.

headers are Class[Senior, 2004] for B3, Class[Senior, 2003] for B5, and Class[Senior, 2002] for B7. Since the values in the three cells are added together, we combine the units using the *or operator* (|) to give the *or unit* Class[Senior, 2004]|Class[Senior, 2003]|Class[Senior, 2002]. The common component can be factored out to yield the unit

Class[Senior, 2004|Senior, 2003|Senior, 2002].

For the sake of brevity, we shorten Class to C, Distrcit to D, Senior to S, and Junior to J in the discussions about units from here on. The inferred row-level component of the unit can be combined with the column-level component using the & operator to give

D[Adrian]&C[S, 2004|S, 2003|S, 2002]

as the the unit for B9.

Unit expressions can be transformed according to the laws detailed in [8]. We can then identify a class of unit expressions that are considered to be well formed. Cell formulas whose derived unit expressions cannot be transformed into well-formed units are considered erroneous, and the system reports unit errors for such cells. In the current version of the system, cells that have unit errors are shaded orange. Such errors are called primary unit errors. The cells that have formulas that reference cells with unit errors are shaded yellow. Such errors are called propagation unit errors. This *fault-localization feedback* is aimed at directing the user's attention to the cells that have primary unit errors since correcting these also removes the propagation unit errors (at least in cases they do not contain their own unit errors).

We now go over the errors we seeded in the spreadsheet for the think-aloud study and discuss why they are unit errors and the feedback from the system (as shown in Figure 1). The spreadsheet has the number of students enrolled in the senior and junior classes in 6 school districts of Oregon collected over a 3-year period (the numbers are fictitious). Row 9 has the total number of seniors over the 3-year period and row 10 has the total number of juniors over the 3-year period. Column H has the total number of students over all the school districts.

1. The formula in cell B5 is a reference to cell C4. This situation might arise if the user accidentally clicked a cell or if the user wanted the values in the cells to be the same. The reference in B5 leads to a unit error because the cell has the unit D[Adrian]&C[S,2003] by virtue of its position and the unit D[Amity]&C[J,2004] by virtue of the reference. After running UCheck, the cell B5 is shaded orange because the cell has a unit error, and the cells H5 and B9 are shaded yellow since their formulas have references to B5. H9 is shaded yellow because its formula has a reference to B9.
2. The formula SUM(C2,C4,C6,C8) in C10 includes a reference to C2. Cells C4, C6, and C8 have D[Amity] as the unit from the column headers whereas C2 has D as its unit. Therefore the unit for the formula in C10 cannot be simplified to a well-formed unit, and the cell is shaded orange as shown in Figure 1. H10 is shaded yellow since its formula contains a reference to C10.
3. The formulas in cells B9, C9, D9, F9, and G9 have references to cells B5, C5, D5, F5, G5, respectively, and have C[S,2003] as part of their units.[2] E9, on the other hand, has C[J,2003] as part of its unit since it has a reference to E6 (instead of E5). The formulas in cells B9 through G9 are all unit correct individually, but when their values are combined by the formula SUM(B9:G9) in H9, the resulting unit cannot be simplified to a well-formed unit since the unit of E9 has a component that does not match those from the other cells. Similarly, B10, C10 (once the reference to C2 has been removed), D10, E10, and G10 refer to cells in the same column and rows 4, 6, and 8. The formula in F10 does not have a reference to F4. The units of the cells B10 through G10 are all valid individually, but result in a unit error when combined by the formula in H10 since the unit of F10 does not agree with those from the other cells. In Figure 1, both H9 and H10 are shaded yellow since at this point they also have the effect of the unit errors propagated from B5 and C10, respectively. Once those errors have been corrected, the system will shade H9 and H10 orange since they now have primary unit errors only.

UCheck's label-based type system works on a finer level of granularity than types like Integer and String in traditional programming languages. UCheck uses the metaphors from the user's domain by employing the header information from the spreadsheet the user is working on.

4. Preliminary Evaluation

As part of the outreach component of the EUSES collaboration, courses in spreadsheet safety targeted at high school teachers and students have been conducted [9]. A pilot session was conducted with two high school students. They were given a 20 minute introduction to the UCheck system. After the introductory presentation, the participants were asked to identify and remove unit errors from a spreadsheet seeded with 8 unit errors. It was observed that the two participants successfully removed all 8 unit errors in under five minutes. During the task, it was also observed that the participants debugged the spreadsheet left-to-right and top-to-bottom. This strategy might have served them well since the spreadsheet was *well designed* according to Teo and Tan [20], because input/data cells are to the left and upper part of the spreadsheet and formula cells are to the right and bottom part of the spreadsheet. We also observed that the students were not really using the written spreadsheet specification to help them correct the unit errors. They were simply looking at the cells shaded orange by UCheck, comparing their formulas with the formulas of the neighboring cells, and then changing the formulas of the orange cells. This observation prompted us to design one of the tasks for the later think-aloud study such that this strategy would not work. We were hoping that would give us better insight into other debugging strategies end users might come up with, especially in those cases where neighboring cells are not available for help/reference.

We also conducted a course spread out over 4 weeks, with 3-hour sessions on 3 days, each week, that dealt with spreadsheet safety and the use of spreadsheets in Mathematics education. In this context, the participants were given a brief introduction to the WYSIWYT testing methodology and UCheck.

During the sessions with UCheck, the subjects expressed their appreciation for its automatic error-checking capabilities. The subjects also found the fault-localization feature of UCheck a big time saver. When asked about their understanding of units, one of the teachers pointed out that UCheck is more related to the idea of sets (for example, the set of fruits with the set of apples and the set of oranges as subsets) rather than to the physical concept of units (for example, m/s^2 as the unit for speed). This comment was interesting since it illustrates how the teacher was trying to understand UCheck within the framework of Mathematical concepts he was familiar with. During our discussions with the teachers, it became clear that they had a pretty good understanding of how UCheck uses headers for carrying out consistency checking in spite of the fact that the underlying formal system was never discussed. In some situations, the teachers were also able to identify false positives reported by UCheck as due to insufficient header information.

The workshops showed that high school teachers found UCheck easy enough to understand so that they could use it effectively. Moreover, the teachers were able to develop lesson plans and use those to guide high school students on how to create safe spreadsheets.

To obtain a better picture of the usability of UCheck, we carried out a think-aloud study, which we will describe in the remainder of this section.

[2] For example, the formula in B9 is Sum(B3,B5,B7).

	A	B	C	D	E	F	G	H
1			Courses					
2		Students	Math	Social Studies	Natural Science	Literature	Student Average	Student GPA
3		M. Abrams	95	82	89	91	89.25	3.57
4		J. Besseck	84	64	69	67	71	2.84
5		T. Cooper	36	87	73	99	73.75	2.95
6		A. Dunn	78	76	85	87	81.5	3.26
7		L. Fannon	82	69	99	76	81.5	3.26
8		J. Greene	69	99	94	72	83.5	3.34
9		I. Harmon	75	87	76	85	80.75	3.23
10		L. Irving	86	63	81	83	78.25	3.13
11		T. James	98	72	75	94	84.75	3.39
12		K. Kady	92	85	88	72	84.25	3.37
13		R. Lennon	76	81	96	67	80	3.20
14		E. McGovern	55	99	92	72	79.5	3.18
15		W. Nash	97	91	100	94	95.5	3.82
16		O. Peterson	88	95	84	94	90.25	3.61
17		Y. Richards	92	72	65	83	78	3.12
18		P. Stone	90	86	42	89	76.75	3.07
19		C. Torok	73	71	67	72	73.75	2.95
20		S. VanBuren	79	68	98	79	81	3.24
21		I. Walters	88	78	72	94	83	3.32
22		P. Young	82	94	88	91	88.75	3.55
23		Class Average	80.75	81.3	81.65	83.05		
24		Class GPA	3.23	3.252	3.266	3.322		

Figure 2. Grade sheet with unit errors.

4.1. Participants

Five teachers (we refer to them as Subject A through Subject E), from among the ten who were in our course on spreadsheet safety, participated in the think-aloud study. We also used a high school student working with us under the Saturday Academy Program as another subject (Subject F). All the subjects had been introduced to UCheck prior to the study. The study was designed to gauge their understanding of UCheck, and how effective they would be in debugging unit errors.

Subjects A through E had little or no prior programming experience. Subject F was familiar with two programming languages he has used for class projects at high school level. Subjects B and C had never used spreadsheets before our course on spreadsheet safety. All the others had some exposure to spreadsheets (mostly ones they had developed for their own use) prior to the study.

4.2. Procedure

At the start of the think-aloud session, the subjects were given a tutorial on UCheck followed by an exercise on how to think aloud. The subjects were then asked to debug unit errors in two spreadsheets.

In the first task, the subjects were asked to debug unit errors seeded in the grade sheet shown in Figure 2. Even though this spreadsheet has more cells than the one used in the second task, the errors were more straightforward to debug. Therefore, this task can be considered the easier of the two tasks. The grade sheet was seeded with 4 different kinds of unit errors. These errors led to invalid units being inferred in 8 other cells. On running UCheck on the spreadsheet, the subjects would initially see the fault localization feedback shown in Figure 2.

In the second task, the subjects were asked to debug unit errors in the spreadsheet shown in Figure 1. The spreadsheet was seeded with 4 errors. The fault localization feedback presented to the subjects after the first run of UCheck is shown in Figure 1. These errors led to invalid units being inferred in 4 other cells. Only six errors are displayed because cells H9 and H10 each contain two errors. They are shaded yellow in the figure since they have propagation errors from B5 and C10, respectively, in addition to the primary errors we seeded in the cells themselves.

In addition to the audio recording of the think-aloud sessions, we also captured video data of the participants' on-screen interactions. Both data sources were used for further analysis.

At the end of the two tasks, the subjects were asked to fill out a post-test questionnaire aimed at finding out more about their experiences debugging the spreadsheets using UCheck. The questionnaire also sought feedback on aspects of the system that the subjects would like changed. To gauge the subjects' understanding of the different kinds of unit errors, we asked the subjects to describe in their own words the different types of errors that UCheck would help them detect.

4.3. Research Questions

The think-aloud study was aimed at answering the following research questions.

RQ1: *Do end users understand the concept of units well enough to be able to correct the unit errors reported by UCheck? Do they introduce new errors while debugging unit errors?*
UCheck automatically detects unit errors within the spreadsheet and shades the cells that have unit errors. The changes to the cell formulas to correct the unit errors have to be figured out by the user. This effort requires at least a basic understanding of units and their causes. Furthermore, RQ1 is important because the system might report unit errors incorrectly in the following cases. For example, UCheck relies on automatic header inference to assign units to the core cells [4, 1]. The spatial analysis algorithms of the header inference system work well in practice. Even so, they are not infallible, that is, the system might report unit errors incorrectly if the headers have been inferred incorrectly. It is therefore important for the user to have a reasonably clear understanding of the requirements of the spreadsheet and the limitations of unit checking so that they can override the system when it is working incorrectly rather than blindly making changes to the spreadsheet formulas.

RQ2: *Does the fault-localization mechanism of UCheck help the end users employ debugging efforts effectively?*
When the user runs UCheck on a spreadsheet, cells that have unit errors are shaded orange, and cells that refer to other cells that have unit errors are shaded yellow. The feedback is aimed at directing the user's attention to the cells that are shaded orange. We were curious if the users would find this feature useful and limit their debugging efforts to the cells that are the sites of unit errors.

RQ3: *What debugging strategies do end users adopt?* We were also interested in any common strategies or techniques adopted by users while debugging unit errors in spreadsheets since such patterns of behavior would help us refine the system. Improvements or modifications to the system could be targeted at supporting preferred strategies of the users.

4.4. Observations

While debugging the unit errors pointed to by UCheck, all the subjects traversed the spreadsheets top-to-bottom. Two of the subjects verbalized their traversal strategies while getting started with the first task at F8.

> "I guess I'll start with F8 since it is the highest cell and work my way down." (Subject F)

> "... an orange one here. I'm just going to start here because that is the first one." (Subject D)

The user would have to correct the cells that are shaded orange to make the shading go away—cells shaded yellow might lose their shading once the corresponding source of error has been corrected. Cells H9 and H10 in the enrollment spreadsheet (shown in Figure 1) are initially shaded yellow because of the propagated errors. Once the sources of those errors have been corrected, the system identifies the primary unit errors in these cells and shades them orange. Therefore, only in such cases a cell's shading might go from yellow to orange. We looked at ways to analyze the unit error in a cell to determine if it is the result of propagation errors only or the combination of primary and propagation errors. Such analyses would allow us to shade all cells with primary unit error orange, even in those cases in which the cells have propagation errors as well. We found out that it is not trivial to determine this difference by the available unit information. We were concerned that this non-monotonic behavior of the fault-localization mechanism might seem counter intuitive to the subjects when they correct the errors in a cell that was shaded orange and the total number of cells shaded orange remains the same or goes up. However, none of the subjects remarked about this effect.

We noticed the following strategies in how the subjects sought feedback from the system about effectiveness of the changes they had made to the cells marked orange by UCheck.

1. *Stepwise debugging* approach: The subject inspects (just) one cell shaded orange, makes a change to the formula and then clicks the "Units" button to seek confirmation the error has been corrected before moving on to the next cell with primary unit error.
2. *Batch debugging* approach: The subject goes through two or more (or potentially all) of the cells that have been shaded orange, makes changes to the formulas, and then clicks the "Units" button to check if the changes done have corrected the unit errors in the spreadsheet.

Subjects B, C, D, and F followed the stepwise debugging approach for both the tasks, whereas Subject A used the batch debugging approach for both the tasks. Subject E switched between the two approaches. During the first task he used stepwise debugging and then used batch debugging for the second task. When asked about it, none of the subjects had any explanation for why they adopted a particular approach over the other in the given situation. However, during the first task Subject C said:

> "Now I think I understand why we'd want to click Units each time. Some of these other ones might go away." (Subject C)

We do not see any advantage of one approach over the other in the case of the tasks used in this study since the formulas are not particularly complex. In the case of spreadsheets with more complex formulas that might cause the user to introduce more errors while editing formulas, the stepwise debugging approach might work better since it yields immediate rewards and helps the user make steady progress.

We can see from the videos of the sessions and the transcripts that the fault-localization mechanism of UCheck guided the subjects' debugging efforts. Three out of the six subjects reviewed the formulas in the cells shaded yellow during the tutorial. None of the subjects spent any time reviewing the error messages or formulas in the yellow cells while working on the tasks. The video data tell us that the fault-localization mechanism helped the subjects focus their attention on cells with primary unit errors and not waste their time and effort inspecting other cells. This observation would take on greater significance in the case of larger spreadsheets. Moreover, studies using subjects working with other programming environments have shown that programmers spent an average of 35% of their time navigating between dependencies, and an average of 46% of their time inspecting task-irrelevant code [13]. The fault localization in UCheck helps to avoid these problems to a large degree. The feedback we gathered using the post-session questionnaire also indicates that the subjects found the fault-localization mechanism of UCheck very helpful. Five out of the six subjects rated the fault localization of UCheck as the most useful feature.

> "Coding by orange and yellow helps me find mistakes quickly and lets me check that my edits are correct." (Subject D)

We observed the following strategies adopted by the subjects in correcting the formulas within the cells with unit errors.

Subjects A, B, C, and F used the cells in the neighborhood of the orange cell as examples to guide the changes they made to the formula in the orange cells.

> "... which is an assumption I should check by scrolling down to correct cells ... " (Subject A)

> "... before I do that, I'm sure that's probably right, I'm just going to look up here, at another cell up above ... and I looked above it to double check ..." (Subject C)

Subject D inspected the cells in the neighborhood of orange cells and also looked at the UCheck error message that pops up when the user places the mouse cursor over the shaded cells. The subject would then come up with a prediction of what the correct formula in the cell should be and write it down before clicking on the cell to inspect the formula within the cell and compare with the predicted formula. The subject basically changed the formula in the cell to the predicted formula to correct the errors.

> "I think I am going to look at different student average just to see ... so I am thinking whatever this one is, this orange one, should be an average something[3]-16 ... so this should say AVERAGE(C19:F19) and if it doesn't say that, I'm going to say this is bad. It says AVERAGE(C18:F19), and that's the error. So I am going to change that, C18 to C19, which should be right."

Subject E inspected the cell values to determine errors within the cells that were shaded orange. This strategy was effective especially in cases in which the cause for the unit errors was not readily obvious as in H9 and H10 from the enrollment sheet.

> "Adding down we get 110, 114, and 315, ... ok, so there's an error somewhere in the adding down or adding across range."

After cross checking the totals in H3, H5, and H7 the subject realized that the error could be in row 9.

> "Well, actually these totals are right. So it should be a horizontal error."

4.5. Discussion

We seeded a total of 8 unit errors in the two spreadsheets. Subjects D and E corrected all the errors in the spreadsheet, and Subjects A, B, and C corrected all the errors except those in cells H9 and H10 in the enrollment spreadsheet. On being told "The error might not necessarily be in the cell that it is being reported in", these three subjects were able to locate and correct the remaining errors as well. Subject F changed the formulas in H9 and H10 such that the unit errors in those cells were corrected but the "errors" originally seeded in E9 and F10 remained uncorrected.[4]

[3]The subject was trying to figure out the column the cell belongs to.

[4]As discussed earlier in the paper, the formulas in cells E9 and F10 by themselves are unit correct.

When debugging cells marked as sites of unit errors, five out of six subjects used the neighboring cells that were free from unit errors as examples to guide their debugging efforts. This kind of behavior ("reuse of uses") has been previously documented in other environments [16]. This approach is more common when the users do not really understand the requirements of the spreadsheets they are debugging. Note that the subjects who relied on this technique exclusively (A, B, C, and F) failed to correct the unit errors reported in H9 and H10 without help since the formulas in these cells agree with those in the neighboring cells.

In the case of cells H9 and H10, the cells in their *spatial proximity* (the immediate neighborhood) are not in their *conceptual proximity* (similarity with respect to the underlying model of the spreadsheet application). The subjects try to derive conceptual proximity from spatial proximity, and this approach fails in the presence of non-trivial spatial patterns the user is unaware of.

4.6. Results

RQ1: All participants in the think-aloud study were successful at debugging the unit errors in the spreadsheets except in the cases in which the system reported the errors away from the site of the errors. The subjects did not introduce any new errors while debugging their spreadsheets. The teachers also created their own spreadsheets, seeded with unit errors, and prepared lesson plans to teach their class about unit checking with UCheck. These two aspects indicate that the users developed a working knowledge of units from the tutorials and introductory sessions. Another important observation was that the users could use the system without any knowledge of the underlying formal rule system for unit inference.

RQ2: On analyzing the video data collected during the think-aloud study, we observed that the debugging efforts of the participants was guided by the fault localization mechanism of the system. During the pre-session tutorial three out of the six subjects inspected the cells shaded yellow, but all six subjects only inspected the cells shaded orange while performing the study tasks. Since we told the subjects that the cells were seeded only with unit errors, the subjects indicated they were done debugging the spreadsheets when no cells were shaded orange by the system.

RQ3: The think-aloud sessions showed us that the subjects navigate the spreadsheets left-to-right and top-to-bottom. The subjects also used cells in the spatial vicinity of the erroneous cell as examples before making changes to their formula. Four subjects consistently followed the stepwise debugging approach, which gave them immediate feedback about their progress. One subject used the batch debugging approach and made corrections to formulas of all the cells marked orange before invoking the unit checker to seek confirmation that the changes made were indeed correct. One subject switched between the two strategies.

From the participants' comments in the post-session

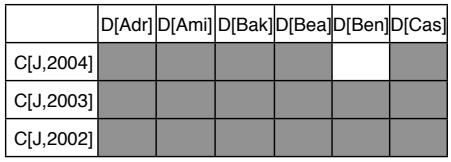


Figure 3. Unit space for missing reference.

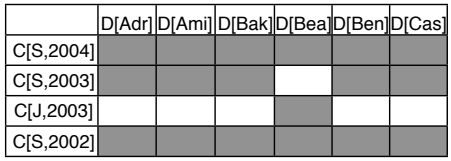


Figure 4. Unit space for incorrect reference.

questionnaire, we see that they found UCheck useful. The participants' success at debugging the unit errors show that the system was easy to use.

> "Without UCheck, I probably wouldn't have noticed the more unobtrusive errors in the spreadsheets." (Subject F)

> "Useful for catching errors without having to *calculate*[5] the entire spreadsheet." (Subject C)

5. Generating Change Suggestions

The preliminary evaluation showed that in some cases the fault-localization mechanism and the rudimentary error messages do not provide sufficient help to detect and debug unit errors. We therefore tried to improve change suggestions and fault-localization feedback to meet the following requirements.

1. Shade the cells that caused the error, not the one in which the unit error is detected.
2. The generated change suggestions should correct the unit error where possible.

We have improved the reasoning behind the fault-localization mechanism by exploiting two additional sources of information.

Unit Space: From the inferred unit for a cell, UCheck can compute the cartesian product of the components of the unit to give what we call the *unit space*. For example, the unit space for the unit in H10 in Figure 1 (after the error in C10 has been corrected) is shown in Figure 3. (The names of the districts have been shortened to save space.) The components that are present in the actual unit for H10 are shaded and the component that is missing is unshaded. We see that there is a concavity in the unit space. With this information, UCheck now looks up the spreadsheet for a cell with the unit D[Bend]&C[J,2004] (F4 in this case) that would fill the concavity, and generates the error message "Including a reference to F4 in the formula in F10 (or H10) would correct the unit error".

The unit space for the unit in H9 in Figure 1 is shown in Figure 4. In this case, UCheck detects convexity and concavity in the unit space. Removing a reference to a cell with the unit D[Beaverton]&C[J,2003] (E6 in this case) from the formula in cell E9 (the only option in this case) would remove the convexity. Adding a reference to a cell with the unit D[Beaverton]&C[S,2003] (E5) to the formula in E9 (or H9) would remove the concavity. Both these changes are required to correct the unit error in H9 and are reported now by UCheck.

Usage Profile: Spreadsheet cells can be classified on the basis of the roles they play as shown in Table 1. The role played by a cell within the spreadsheet is one kind of usage profile. Another kind of usage profile that is considered for an intermediate cell or an input cell is the number of times it is referenced by other cells. Outliers detected by analysis of usage profiles can be indicative of errors.

	formula	**data**
referenced	intermediate	input
not referenced	output	label

Table 1. Cell roles

Spatial Neighbors: Unit space information and usage profile help the fault-localization mechanism identify the cells that are the cause of unit errors and also in generating change suggestions to correct the error. The formulas in the cells that are spatial neighbors of the cell with the unit error are another source of information that can be exploited to generate change suggestions.

After implementing the changes described above, we carried out an evaluation of the modified system to study the effect of the changes. In previous work we have developed a suite of mutation operators for spreadsheets [2]. From the suite of operators, we picked the ones that cause unit errors to seed errors in the spreadsheets used in the study. UCheck was then run on the mutant sheets and the change suggestions and the fault-localization feedback were recorded. The suggested changes were then applied to the mutant sheets and the resulting sheets were then compared with the original sheets to check if the mutations were reversed by the suggested changes.

The empirical evaluation was designed to answer the following research questions.

RQ4: *Does the system shade the cells that cause the unit error?* More specifically, the system should shade the cells that cause the unit error orange.

RQ5: *How effective are the suggested changes in correcting the unit errors?*

[5] On being asked what he meant by "calculate" the subject clarified that he was happy he did not have to step through and check the formulas manually.

The operators used to seed unit errors are shown in Table 2. CRE expands a contiguous range. For example, the contiguous range A3:A7 could get changed to A2:A7 or A3:A8. CSR, on the other hand, shrinks a contiguous range. NRE and NRS operators expand and shrink non-contiguous ranges respectively. CRR operator replaces a reference with a constant. For example, it could mutate the formula A2+A3 to A2+3. The RRR operator replaces a reference within a range with another one not already in the range. For example, RRR could mutate the formula SUM(A1,A3,A5) to SUM(A1,B3,A5).

As can be seen from the results of the evaluation shown in Table 2, the generated change suggestions were able to reverse all the seeded errors (100% for both the grade and enrollment sheets). The fault-localization mechanism was modified so that the cell with the change suggestion would be shaded orange in order to direct the user's attention to the change suggestion. As a result, in all the cases studied, the system shaded the cells with the unit errors orange.

	Grade sheet	Enrollment sheet
Total mutants	1928	2639
Unit errors	1728	2330
Unit errors corrected	1728	2330

Table 2. Change suggestion effectiveness scores

6. Conclusions and Future Work

Our experiences from using UCheck in courses on spreadsheet safety have shown that teachers and students are able to follow how the system exploits header information to carry out consistency checking. The participants of the courses were able to come up with their own examples of unit errors and explain why they were unit errors. They were also able to identify false positives in their own spreadsheets when they occur as the result of insufficient header information.

The think-aloud study we carried out showed that the participants were able to use the fault-localization feedback provided by UCheck and debug unit errors. The study helped us to identify problems participants had while debugging non-local unit errors, and led to a redesign of the fault-localization mechanism. We also extended the unit inference so that the system now generates suggested changes to correct unit errors. The empirical evaluation of the new system shows that it works well in practice, especially in the case of non-local unit errors.

References

[1] R. Abraham and M. Erwig. Header and Unit Inference for Spreadsheets Through Spatial Analyses. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 165–172, 2004.

[2] R. Abraham and M. Erwig. Mutation Testing of Spreadsheets. 2006. Submitted for publication.

[3] R. Abraham and M. Erwig. Type Inference for Spreadsheets. In *ACM Int. Symp. on Principles and Practice of Declarative Programming*, pages 73–84, 2006.

[4] R. Abraham and M. Erwig. UCheck: A Spreadsheet Unit Checker for End Users. *Journal of Visual Languages and Computing*, 18(1):71–95, 2007.

[5] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A Type System for Statically Detecting Spreadsheet Errors. In *18th IEEE Int. Conf. on Automated Software Engineering*, pages 174–183, 2003.

[6] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the Unit Correctness of Spreadsheet Programs. In *26th IEEE Int. Conf. on Software Engineering*, pages 439–448, 2004.

[7] A. Ebrahimi. Novice Programmer Errors: Language Constructs and Plan Composition. *Int. Journal of Human-Computer Studies*, 41(4):457–480, 1994.

[8] M. Erwig and M. M. Burnett. Adding Apples and Oranges. In *4th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 2257, pages 173–191, 2002.

[9] EUSES. Education Outreach Program. `http://eusesconsortium.org/edu/education_activities.php`.

[10] J. D. Gannon. An Experimental Evaluation of Data Type Conventions. *Comm. of the ACM*, 20(8):584–595, 1977.

[11] L. Jiang and Z. Su. Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs. In *Int. Conf. on Software Engineering*, pages 262–271, 2006.

[12] A. Kennedy. Dimension Types. In *5th European Symp. on Programming*, LNCS 788, pages 348–362, 1994.

[13] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In *Int. Conf. on Software Engineering*, pages 126–135, 2005.

[14] S. Prabhakarao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett. Strategies and Behaviors of End-User Programmers with Interactive Fault Localization. In *IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, pages 203–210, 2003.

[15] L. Prechelt and W. F. Tichy. A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking. *IEEE Transactions on Software Engineering*, 24(4):302–312, 1998.

[16] M. Rosson and J. M. Carroll. The Reuse of Uses in Smalltalk Programming. *ACM Trans. on Computer-Human Interaction*, 3(3):219–253, 1996.

[17] G. Rothermel, M. M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A Methodology for Testing Spreadsheets. *ACM Transactions on Software Engineering and Methodology*, pages 110–147, 2001.

[18] J. C. Spohrer and E. Soloway. Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM*, 29(7):624–632, 1986.

[19] J. G. Spohrer and E. Soloway. Analyzing the High Frequency Bugs in Novice Programs. In *First Workshop on Empirical Studies of Programmers*, pages 230–251, 1986.

[20] T. Teo and M. Tan. Quantitative and Qualitative Errors in Spreadsheet Development. In *30th Hawaii Int. Conf. on System Sciences*, pages 25–38, 1997.