Is Spreadsheet Ambiguity Harmful? Detecting and Repairing Spreadsheet Smells due to Ambiguous Computation

Wensheng Dou
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing, China
wsdou@otcaix.iscas.ac.cn

Shing-Chi Cheung
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

Jun Wei
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing, China
wj@otcaix.iscas.ac.cn

ABSTRACT

Spreadsheets are widely used by end users for numerical computation in their business. Spreadsheet cells whose computation is subject to the same semantics are often clustered in a row or column. When a spreadsheet evolves, these cell clusters can degenerate due to ad hoc modifications or undisciplined copy-and-pastes. Such degenerated clusters no longer keep cells prescribing the same computational semantics, and are said to exhibit *ambiguous computation smells*. Our empirical study finds that such smells are common and likely harmful. We propose *AmCheck*, a novel technique that automatically detects and repairs ambiguous computation smells by recovering their intended computational semantics. A case study using AmCheck suggests that it is useful for discovering and repairing real spreadsheet problems.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – *restructuring, reverse engineering, and reengineering*

General Terms

Reliability, Experimentation, Human Factors

Keywords

Spreadsheet, ambiguous computation, smell, repair

1. INTRODUCTION

Spreadsheets are generally developed and maintained by end users who are not familiar with appropriate software development practice. As a result, spreadsheets have been found to be error-prone [28]. Spreadsheet errors can induce great financial losses [26]. Various techniques have been proposed to improve the quality of spreadsheets. Some examples include testing [12][23][2], error or smell detection [6][19][20], and debugging [3][30].

A spreadsheet comprises tables of cells arranged in rows and columns. We refer to a cell cluster as a *cell array* when it is subject to the same computational semantics. For example, the cells [D2:D7] in Figure 1(b) refer to the semantics of "Total" and uniformly follow a formula pattern of $D_i = B_i + C_i$, where $2 \leq i \leq 7$. In general, cell arrays can be specified manually or with tool assistance. In this paper, we focus only on those cell arrays subject to computational semantics expressed in formula patterns without using "if" conditions. Our empirical study reports that there are altogether 16,385 cell arrays among 993 (out of 4,037) spreadsheets in the EUSES corpus [11]. This indicates that cell arrays are common in real-life spreadsheets.

Spreadsheet smells can occur due to a distortion of, or an ambiguity in, the meaning of data or formulas [29]. Spreadsheet software like Excel provides two useful features, *copy-and-paste* and *auto-fill*, to reduce the chances of introducing smells during the creation of new cells in a cell array. Both features can help automatically deduce a formula pattern from selected sample cells [35], and apply it to the new cells in a cell array.

Although the two features provide convenience in editing spreadsheets, their application is restrictive in the sense that end users have little control on the formula pattern deduction process. They may not even be aware of deduced formula patterns. After editing, there is no record in the new cells that they have been created using these two features, and therefore have to be uniformly modified in future. Little provision is offered to warn end users from modifying these cells arbitrarily. In principle, all cells in a cell array should prescribe the same computational semantics. A cell array is said to suffer from an *ambiguous computation smell* when there is more than one computational semantic among the cells it contains. Ad hoc modifications to these cells are one major cause of ambiguous computation smells. For example, the cell array [D2:D7] in Figure 1(a) could be a consequence of ad hoc cell modifications that result in four different formula patterns, leading to an ambiguous computation smell. Note that no warning can be raised by spreadsheet software to alert end users of such a smell. This smell can exist for a long time and even be replicated to other spreadsheets without being discovered. Even though this cell array currently offers a correct value in each cell, it is error-prone and susceptible to developing errors upon future data updates. For example, the value in D2 would be incorrect if the value of C2 is later updated to 1. As ambiguous computation smells are vulnerable to errors, their early detection is important. It is particularly the case for those spreadsheets subject to liability consequences such as company reports for release to authorized third parties.

Spreadsheet software like Excel provides a mechanism to detect cells with inappropriate formulas. However, the detection is appli-

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568316

cable only to the situation where: (a) a cell's formula is syntactically inconsistent with those of its two adjacent cells, and (b) the formulas of the two adjacent cells are syntactically consistent. As such, Excel is not able to raise any warning for the cell array [D2:D7] in Figure 1(a). Since each cell in the cell array [D2:D7] in Figure 1(a) does not have any unit or dimension error, the smell cannot be detected by UCheck [4] and dimension inference [7], either.

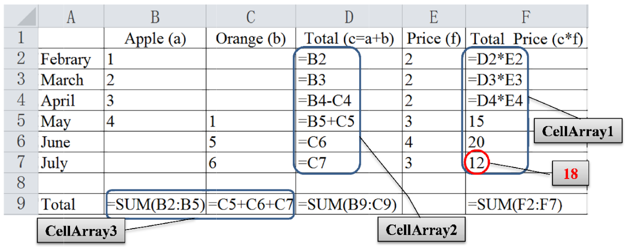
Like semantic bugs in programming languages [24][33], it is hard to identify which cells contain inappropriate formulas, because this involves knowledge of intended semantics, which often requires human judgments or specifications. Automatic repairing of inappropriate cell formulas is another non-trivial challenge.

In this paper, we focus on automated identification of cell arrays as well as detection and repairing of ambiguous computation smells. In order to detect and repair ambiguous computation smells, we need to know formula patterns that capture computational semantics in cell arrays. The key challenge is how to infer appropriate formula patterns for repairing cell arrays that suffer from ambiguous computation smells. Our technique is based on two observations: (1) Consecutive cells in a row or column often have the same computational semantics. They likely share similar formula patterns. (2) Even if some cells in a cell array have been modified arbitrarily, a majority of cells in this array could still share an equivalent formula pattern, and other cells could prescribe a sub- or super-form of this formula pattern. These two observations allow us to identify candidate cell arrays effectively using heuristics. We then map ambiguous computation smell detection to a constraint satisfaction problem. To repair an ambiguous computation smell, we propose an algorithm adapted from existing work on program synthesis by Jha et al. [21] to infer an appropriate formula pattern that generalizes the formulas of cells in an array that suffers from this smell as many as possible.

We evaluated our technique from two perspectives. First, we analyzed the EUSES corpus [11] to learn how often ambiguous computation smells can occur, and measured the precision and performance of our technique for detecting ambiguous computation smells. Second, we conducted a case study using our technique with real-life spreadsheets prepared by budget finance officers in the Institute of Software, Chinese Academy of Sciences. Our evaluation reports that: (1) 27.3% of spreadsheets with formulas in the EUSES corpus suffer from ambiguous computation smells, which cover 21.6% of identified cell arrays. (2) Ambiguous computation smells reveal weakness and have caused errors in spreadsheets. From randomly sampled 319 true ambiguous computation smells, 434 errors are found with these smells. (3) Ambiguous computation smells are often caused by careless updates to cells previously created by auto-fill and copy-and-paste. Our technique can help end users detect and repair such smells, thus improving the quality of their spreadsheets.

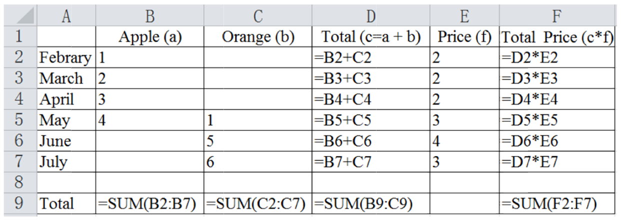
We summarize main contributions of this paper as follows:

- We propose a new and common type of spreadsheet smell, ambiguous computation smell, which is error-prone.
- We propose a novel technique, AmCheck, to detect and repair ambiguous computation smells by identifying arrays of cells that are subject to the same computational semantics, inferring these cells' formula patterns, spotting incompatible patterns, and synthesizing new patterns to repair the smells.
- We evaluate our tool implementation experimentally on the EUSES corpus and real-life spreadsheets used in practice. The experimental results show that ambiguous computation smells are common and indeed vulnerable to errors.



	A	B	C	D	E	F
1		Apple (a)	Orange (b)	Total (c=a+b)	Price (f)	Total Price (c*f)
2	Febrary	1		=B2	2	=D2*E2
3	March	2		=B3	2	=D3*E3
4	April	3		=B4-C4	2	=D4*E4
5	May	4	1	=B5+C5	3	15
6	June		5	=C6	4	20
7	July		6	=C7	3	12
8						
9	Total	=SUM(B2:B5)	=C5+C6+C7	=SUM(B9:C9)		=SUM(F2:F7)

(a) A spreadsheet with ambiguous computation smells.



	A	B	C	D	E	F
1		Apple (a)	Orange (b)	Total (c=a + b)	Price (f)	Total Price (c*f)
2	Febrary	1		=B2+C2	2	=D2*E2
3	March	2		=B3+C3	2	=D3*E3
4	April	3		=B4+C4	2	=D4*E4
5	May	4	1	=B5+C5	3	=D5*E5
6	June		5	=B6+C6	4	=D6*E6
7	July		6	=B7+C7	3	=D7*E7
8						
9	Total	=SUM(B2:B7)	=SUM(C2:C7)	=SUM(B9:C9)		=SUM(F2:F7)

(b) A spreadsheet without ambiguous computation smells.
Figure 1. A motivating example.

The remainder of this paper is organized as follows. §2 gives a motivating example and explains the use of our technique. §3 defines ambiguous computation smell. §4 proposes our smell detection and repairing technique. §5 presents our tool implementation. §6 evaluates AmCheck with real-life spreadsheets. §7 discusses related work, and finally §8 concludes this paper.

2. MOTIVATION

In this section, we illustrate ambiguous computation smells using an example spreadsheet extracted from the EUSES corpus [11]. We then explain how to detect and repair these smells.

2.1 Example

Figure 1(a) is a spreadsheet computing monthly harvest of fruits. It exhibits two kinds of ambiguous computation smells:

Missing formula smell. This ambiguous computation smell occurs when some cells in a cell array do not prescribe any formula. Such smell can be introduced to a cell array when end users override the formula in a cell with a plain value. For example, CellArray1 [F2:F7] is subject to the computation of "Total Price" with an intended formula pattern of $F_i = D_i * E_i$, where $2 \le i \le 7$. Unlike cells F2, F3 and F4, the values of cells F5, F6 and F7 are not computed by formulas.

Inconsistent formula smell. This ambiguous computation smell occurs when the cells in a cell array prescribe different formula patterns. Such smell can be introduced to a cell array when end users specify the formula of a cell in the cell array inappropriately without realizing its computational semantics. For example, CellArray2 [D2:D7] is subject to the computation of "Total" with an intended formula pattern of $D_i = B_i + C_i$, where $2 \le i \le 7$. End users may understand that there is no orange output in February, and thus leave C2 empty. They specify a formula that ignores C2 at D2, and as a result CellArray2 prescribes more than one formula pattern. A similar smell also occurs to CellArray3 [B9:C9].

Although CellArray2 and CellArray3 in Figure 1(a) suffer from ambiguous computation smells, the values given by their member cells are appropriate. However, the smells can lead to errors in D2 and C9 if C2 is updated later with any non-zero value. Besides, problems can arise when end users apply copy-and-paste and auto-fill to these cell arrays later. An ambiguous computation smell would also likely contain an error (e.g., F7) if one is unable to find any formula pattern that can lead to values in such a cell array.

2.2 AmCheck Overview

Several technical challenges need to be addressed in the detection and repairing of ambiguous computation smells in spreadsheets. We explain them using the example in Figure 1(a). First, does a cell (e.g., F5) belong to a cell array? If yes, what is the boundary of such a cell array? Second, do cells in a cell array inappropriately prescribe semantically different formula patterns? Note that we consider two formula patterns to be the same if the formulas derived from these patterns offer the same computation, e.g., $x + x$ and $2*x$. Third, how may one construct an appropriate formula pattern for a cell array that prescribes more than one formula pattern? This is a challenging question because there are chances that none of cells in such a cell array is using an appropriate formula, e.g., cells B9 and C9 in CellArray3. Even worse, cells in such a cell array may prescribe conflicting formulas patterns, e.g., cells D4 and D5 in CellArray2. Fourth, some cells (e.g., F5) in a cell array may prescribe no formula. The values of these cells (e.g., F7) may even conflict with their appropriate formula patterns.

Our AmCheck infers formula patterns by means of constraints in two steps. First, it uses values and formulas in a cell array to infer underlying constraints of formula patterns prescribed by this cell array. Second, it uses the inferred constraints to derive target formula patterns. AmCheck uses component-based program synthesis [21] to construct candidate formula patterns for repairing ambiguous computation smells. To achieve this, AmCheck needs to cope with noises induced by conflicting formulas (e.g., D4 and D5) and potential errors (e.g., F7). For example, AmCheck can construct a candidate formula pattern ($B_i + C_i$, where $2 \le i \le 7$) for repairing the ambiguous computation smell in CellArray2. It can also use its inferred formula patterns to detect errors (e.g., F7) in cell arrays.

3. AMBIGUOUS COMPUTATION SMELLS

In this section, we introduce spreadsheet programming model, and explain key concepts like cell array and ambiguous computation smell to be used in subsequent discussions.

3.1 Spreadsheet Programming Model

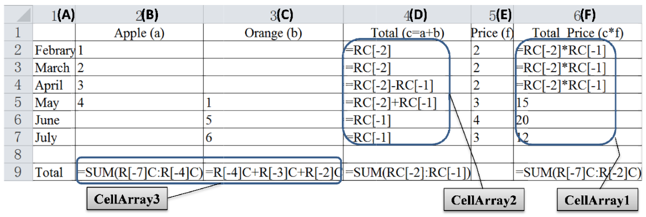
A spreadsheet can be modeled as a set of expressions referenced by two-dimensional cell addresses [5]. Cells in a spreadsheet can be partitioned into *data cells* and *formula cells*, depending on whether they contain formulas. If a cell contains a formula, the expression of that cell is its formula. Let A be the set of possible address references, EXP be the set of possible expressions, and V be the set of plain values. An expression *exp* is either a plain value $v \in V$, an address reference ($a \in A$) to another cell, or a function (φ) applied to one or more expressions. Functions in spreadsheets include basic operators (e.g., +, –, *, /) and built-in functions from spreadsheet software (e.g., SUM, AVERAGE and MAX). Formally, an expression *exp* is:

$$exp = v \mid a \mid \varphi(exp_1, \dots, exp_n).$$

We define a function $\sigma : EXP \rightarrow 2^A$, which gives the set of address references used by an expression *exp*. The cells indexed by references in $\sigma(exp)$ are referred to as the *input cells* of *exp*:

$$\sigma(exp) = \begin{cases} \emptyset & exp \in V; \\ \{exp\} & exp \in A; \\ \sigma(exp_1) \cup \dots \cup \sigma(exp_n) & exp = \varphi(exp_1, \dots, exp_n). \end{cases}$$

Relative R1C1 is a widely used notation for referencing cells in spreadsheet programming [34]. A cell at n rows below and m columns on the right of the current cell is notated as R[n]C[m]. The bracket for the row (or column) can be omitted when a refer-



	1(A)	2(B)	3(C)	4(D)	5(E)	6(F)
1		Apple (a)	Orange (b)	Total (c=a+b)	Price (f)	Total Price (c*f)
2	Febrary	1		=RC[-2]	2	=RC[-2]*RC[-1]
3	March	2		=RC[-2]	2	=RC[-2]*RC[-1]
4	April	3		=RC[-2]-RC[-1]	2	=RC[-2]*RC[-1]
5	May	4	1	=RC[-2]+RC[-1]	3	15
6	June		5	=RC[-1]	4	20
7	July		6	=RC[-1]	3	12
8						
9	Total	=SUM(R[-7]C:R[-4]C)	=R[-4]C+R[-3]C+R[-2]C	=SUM(RC[-2]:RC[-1])		=SUM(R[-7]C:R[-2]C)

Figure 2. Motivating example with relative R1C1 notation.

enced cell is on the same row (or column). This notation allows cell references in a formula to be expressed as indexes relative to the cell containing this formula. An interesting observation is that formulas prescribing the same pattern have equivalent relative R1C1 expressions. For example, the formula B5 + C5 in cell D5 in Figure 1(a) can be rewritten as RC[-2] + RC[-1] in Figure 2 using this relative R1C1 notation. It means a summation of two values. The first value is given by a cell at the same row but two columns left. The second value is given by a cell at the same row but one column left. Figure 2 gives corresponding relative R1C1 expressions for all formulas contained in the spreadsheet in Figure 1(a). A referenced cell with *absolute* R1C1 can be treated as a constant. In subsequent discussions, "R1C1 notation" always means relative R1C1 notation for ease of presentation.

3.2 Cell Array and Smell

In a spreadsheet, cells with the same computational semantics are usually grouped together in a row or column.

Definition 1: A *cell array* is a consecutive range of cells (e.g., CellArray1, CellArray2 and CellArray3 in Figure 1(a)) in a row or column prescribing one business concept and subject to certain computational semantics.

Since cells in a cell array often use formulas to express such computational semantics, we name a cell array's computational semantics its *formula pattern* ($f_{pattern}$). Let *CellArray* be the set of cells in a cell array. We say that a cell array is *well-formed* if the following condition holds (in R1C1 notation):

$$\forall c_1, c_2 \in CellArray, \sigma(c_1.exp) = \sigma(c_2.exp) \\ \wedge c_1.exp = c_2.exp.$$

Intuitively, a cell array is well-formed when all its member cells contain equivalent R1C1 expressions. The first condition states that any two cell expressions in this cell array have the same set of references (i.e., cell addresses in R1C1 notation). The second condition states that any two cell expressions should be evaluated to the same value given the same bindings of their input cells. For example, two expressions "2 * (R[-2]C + R[-1]C)" and "2 * R[-2]C + 2 * R[-1]C" are equivalent although they are syntactically different. Our AmCheck checks well-formedness using constraint solver Choco [22]. Since a well-formed cell array has all its cell expressions equivalent, we can take any of them as the cell array's $f_{pattern}$. If a cell array is not well-formed, we say that it suffers from an *ambiguous computation smell* or it is *smelly*.

There are two common types of cell arrays in spreadsheets:

Row-based cell array. It comprises consecutive cells in a row. Each of these cells often has a formula referencing other cells in the same column as inputs. For example, [B9:C9] in Figure 1(b) is a row-based cell array.

Column-based cell array. It comprises consecutive cells in a column. Each of these cells often has a formula referencing other cells in the same row as inputs. For example, [D2:D7] in Figure 1(b) is a column-based cell array.

Smells can occur in a cell array when end users make ad hoc modifications to its cells. Such modifications can be made by inexperienced end users to accommodate last-minute modifications under tight deadlines. We find two common types of ambiguous computation smell: *missing formula smell* and *inconsistent formula smell*, as explained earlier. A *missing formula smell* occurs in a not well-formed cell array when it contains a data cell. An *inconsistent formula smell* occurs in a not well-formed cell array when it has two formula cells with semantically different R1C1 expressions. A cell array of more than two cells can suffer from missing formula and inconsistent formula smells at the same time.

Definition 2: A *conformance error* occurs when the value of a cell in a cell array does not conform to that computed by this cell array's $f_{pattern}$:

$$\exists c \in CellArray, c.value \neq f_{pattern}(c.inputs).$$

A conformance error may be caused by improper modifications to a cell array such that it suffers from ambiguous computation smells. Conformance errors reflect true data discrepancies in spreadsheets, such as F7 in Figure 1(a).

4. DETECTING AND REPAIRING AMBIGUOUS COMPUTATION SMELLS

Given a spreadsheet, our AmCheck analyzes it and reports all detected ambiguous computation smells with repair suggestions. AmCheck heuristically identifies cell arrays (§4.1), and detects ambiguous computation smells via constraint solving (§4.2). To repair ambiguous computation smells, AmCheck infers an $f_{pattern}$ in two steps. First, AmCheck uses values and formulas in a cell array to derive constraints associated with its underlying formula pattern (§4.3). Second, AmCheck infers an $f_{pattern}$ based on these constraints. In order to expedite the inference process, AmCheck combines heuristics (§4.4) and program synthesis techniques (§4.5). After the inference, AmCheck reports a conformance error if any cell in a cell array has a value not conforming to the one computed by the inferred $f_{pattern}$.

4.1 Extracting Cell Arrays

The first challenge of ambiguous computation smell detection is to identify cell arrays from a given spreadsheet, which has no record about cells previously prepared by copy-and-paste and auto-fill. We observe that a spreadsheet snippet usually provides useful hints about boundaries of cell arrays. Besides, the formula of a cell in a cell array often references other cells in the same row or column as this cell. These two observations facilitate our cell array identification and extraction.

The first step is to identify potential snippets. Related data and formulas in a spreadsheet are often clustered together in a rectangle circumscribed by empty cells or labels [16]. We refer to such rectangles of cells as *snippets*. Examples of spreadsheet snippets in Figure 1(a) include two rectangles comprising cells [B2:F7] and [B9:D9], respectively.

To identify snippets, we adopt a cell classification strategy, similar to what Abraham and Erwig [4] suggested. We define a *fence* as a row or column of cells that comprises only empty cells or labels in a spreadsheet. We use fences to identify boundaries for each spreadsheet snippet. Other cells inside the identified boundaries are considered as cells of this spreadsheet snippet.

We describe our spreadsheet snippet identification algorithm briefly as follows. Initially, each spreadsheet is considered as one snippet. We then identify fences in this snippet, and divide this snippet into more ones by the identified fences. For each newly identified snippet, we repeat this refinement until no further snippet can be identified.

The second step is to extract cell arrays from identified spreadsheet snippets. As mentioned, the formula of a cell in a cell array often references other cells in the same row or column as this cell. Our cell array extraction algorithm works as follows. For each snippet, it examines consecutive cells clustered in a row or column, and considers a cluster as a cell array if: (1) the cluster is not a subset of another cell array, and (2) the formula of each cell in the cluster consistently references input cells from the same column or row as this cell. The algorithm may encounter the following four scenarios in its extraction process:

- **Row-based cell array:** For a cluster of consecutive cells in a row, some cells contain formulas, and for each cell with a formula, its formula only references input cells from the same column as this cell. We then consider this cluster as a row-based cell array. One example is [B9:C9] in Figure 1(a).
- **Column-based cell array:** For a cluster of consecutive cells in a column, some cells contain formulas, and for each cell with a formula, its formula only references input cells from the same row as this cell. We then consider this cluster as a column-based cell array. One example is [D2:D7] in Figure 1(a).
- **Pure value:** It is hard to judge whether a row or column containing only data cells prescribes one business concept and is subject to certain computational semantics. We do not consider such rows or columns as cell arrays.
- **Other cases:** If a row or column does not belong to any of the above cases, we also do not consider it as a cell array.

4.2 Detecting Ambiguous Computation Smells

By Definition 1, a cell array is well-formed if it satisfies: (1) it contains only formula cells, (2) all its formulas use the same inputs (in R1C1 notation), and (3) all its formulas give the same outputs given the same input values. Thus, we can map ambiguous computation smell detection to a constraint satisfaction problem, and rewrite the above three conditions as follows:

$$\forall c_1, c_2 \in CellArray, \sigma(c_1.exp) = \sigma(c_2.exp)$$
$$\wedge \nexists\, input, c_1.exp(input) \neq c_2.exp(input).$$

Here, we use *input* to denote any possible input values to cells c_1's and c_2's expressions. If a cell array does not satisfy any of the above conditions, it suffers from ambiguous computation smells.

4.3 Extracting Formula Cell Constraints

Given a smelly cell array, we expect AmCheck to detect any existence of conformance errors as defined in Definition 2. To do that, AmCheck needs to recover an R1C1 expression from existing cells to represent this cell array's $f_{pattern}$.

Our construction technique is inspired by component-based program synthesis, which synthesizes a loopless program from components, input-output pairs and specifications used by this program [14][21]. The construction is based on three assumptions: (1) Components in formulas used by a cell array are often components used by this cell array's $f_{pattern}$; (2) Most values should be correct for this cell array, and they can serve as input-output pairs; (3) Existing formulas in the cell array are good hints of $f_{pattern}$. Under these assumptions, AmCheck recovers an R1C1 expression by extracting its constraints from cells of a smelly cell array, and combining them appropriately. The extraction process consists of four parts, i.e., extracting input variables, components, input-output pairs and functions from a smelly cell array, as follows:

1) All cells referenced by formulas in a cell array are considered as input variables for this cell array's $f_{pattern}$. For example, input variables for CellArray1 are RC[-2] and RC[-1], input variables for CellArray2 are RC[-2] and RC[-1], and input variables for CellArray3 are R[-7]C, R[-6]C, …, R[-2]C (in Figure 1 (a)). The process may extract irrelevant input variables, which can be removed at a later stage. Let *IV* be the set of all input variables of a cell array. After extracting *n* input variables for $f_{pattern}$, we can model $f_{pattern}$ as a function $f(x_1, x_2, \dots x_n)$. Formally, *IV* is defined as follows:

$$IV = \sigma(c_1.exp) \cup \sigma(c_2.exp) \dots \cup \sigma(c_m.exp),$$
$$\text{where } c_1, c_2, \dots c_m \in CellArray.$$

2) All operators used by formulas in the cell array are considered as components. For example, components from CellArray1 include "*", components from CellArray2 include "+" and "–", and components from CellArray3 include "+" and SUM. Some components may be irrelevant, but we can skip them at a later stage. If AmCheck fails to find any operator from a cell array, it would add basic operators (e.g., +, –, *, /) as components.

3) All data in the cell array are considered as input-output pairs. For example, in CellArray2, <(1, 0), 1>, <(2, 0), 2>, <(3, 0), 3>, <(4, 1), 5>, <(0, 5), 5> and <(0, 6), 6> are considered as input-output pairs.

4) Existing formulas in the cell array can be modeled as functions. For example, one can extract from CellArray2 four functions, namely, $f(x_1, 0) = x_1$, $f(x_1, x_2) = x_1 + x_2$, $f(x_1, x_2) = x_1 - x_2$ and $f(0, x_2) = x_2$. These functions are treated as specifications in component-based program synthesis [14][21].

All these extracted input variables, components, input-output pairs and functions are constraints used for recovering $f_{pattern}$.

4.4 Recovering $f_{pattern}$

We observe that a cell array's $f_{pattern}$ can exist in functions extracted from the cell array's formula cells. For example, function $f(x_1, x_2) = x_1 * x_2$ extracted from formula cells in CellArray1 in Figure 1(a) is a good candidate for recovering CellArray1's $f_{pattern}$. This observation enables us to recover a cell array's $f_{pattern}$ based on a candidate set of functions obtained from its formula cells. This can significantly reduce the cost of formula pattern inference since program synthesis [21] is expensive. We aim to select a function that contains all input variables and covers all cells in a cell array as its $f_{pattern}$. We say that a function *covers* a data cell when the cell's value can be computed by the function. For example, the value of F5 in CellArray1 in Figure 1(a) can be computed by $f(x_1, x_2) = x_1 * x_2$. We say that a function *covers* a formula cell if the function is *compatible* with the one extracted from the cell's formula in the sense that both of them can produce the same outputs given the same inputs. For example, function $f(x_1, x_2) = x_1 + x_2$ is compatible with function $f(x_1, 0) = x_1$ extracted from D2 in Figure 1(a). Note that the second parameter binds to zero for the two functions to have the same inputs. However, $f(x_1, x_2) = x_1 + x_2$ is incompatible with another function $f(x_1, x_2) = x_1 - x_2$ extracted from D4 in CellArray2 in Figure 1(a) because their outputs are different when x_1 is 0 and x_2 is 1.

Algorithm 1 gives our $f_{pattern}$ recovery algorithm. The algorithm returns NULL if it fails to recover any $f_{pattern}$ from functions extracted from a given cell array, which contains at least one formula cell. If only one function can be extracted from a given cell array, it is treated as the cell array's $f_{pattern}$ (Lines 1–3). Otherwise, a function that can cover (by the Coverage method) all values and formulas in the given cell array (Lines 4–10) is treated as the $f_{pattern}$. The Coverage method (Lines 13–27) computes the ratio of the number of cells a function can cover over the total number of cells in the cell array. Lines 17-19 (and Lines 21-23) check whether a formula (and data) cell is covered by a function.

Algorithm 1. $f_{pattern}$ recovery algorithm.

```
Input: IV (input variables), FUNC (functions), IO (input-
       output pairs), CA (cell array).
Output: F (target formula pattern) or NULL.
 1: if (FUNC.length == 1)
 2:   return FUNC.get(0)
 3: end if
 4: foreach fn in FUNC do
 5:   if fn contains all input variables in IV then
 6:     if (Coverage(fn, CA) == 100%) then
 7:       return fn
 8:     end if
 9:   end if
10: end for
11: return NULL
12:
13: method Coverage(fn, CA)
14:   coveredCells = 0
15:   foreach cell in CA do
16:     if (cell.type == FORMULA) then
17:       if (!∃input. fn(input)≠cell.exp(input)) then
18:         coveredCells ++
19:       end if
20:     else  // Plain value case
21:       if (fn(cell.input) == cell.value) then
22:         coveredCells ++
23:       end if
24:     end if
25:   end for
26:   return coveredCells / CA.length
27: end method
```

4.5 Synthesizing $f_{pattern}$

The $f_{pattern}$ recovery algorithm returns NULL when it fails to identify an appropriate $f_{pattern}$ of a smelly cell array from its extracted functions. When this happens, AmCheck synthesizes the $f_{pattern}$ using component-based program synthesis [14][21].

Let us review the basic mechanism of component-based program synthesis to construct a program before explaining the $f_{pattern}$ synthesis algorithm. Program synthesis first derives constraints ($constraints_{ps}$) for a program to be synthesized based on a set of components and input-output pairs (generated by specifications [14] or provided by users [21]). It then solves $constraints_{ps}$ to synthesize the program. If the input-output pairs provided are not sufficiently restrictive, multiple candidate programs can be synthesized (all satisfying $constraints_{ps}$). More input-output pairs can be used to provide additional constraints to strengthen $constraints_{ps}$ until a unique program is synthesized.

Algorithm 2 gives the pseudo-code of our $f_{pattern}$ synthesis algorithm. There are three challenges in synthesizing the $f_{pattern}$: (1) Component-based program synthesis [14][21] requires users to explicitly provide components and input-output pairs [21]. The algorithm addresses this using constraints extracted from values and formulas in a smelly cell array (§4.3). (2) Functions extracted from a smelly cell array's formulas can be incompatible with one another. For example, two functions $f(x_1, x_2) = x_1 + x_2$ and $f(x_1, x_2) = x_1 - x_2$ extracted from CellArray2 in Figure 1(a) are incompatible. Such incompatibility can make our $f_{pattern}$ synthesis fail. (3) Data cells may contain incorrect values, which cannot be computed by an appropriate $f_{pattern}$ of the cell array. Such incorrect values can also make our $f_{pattern}$ synthesis fail.

To tackle the second challenge, Algorithm 2 classifies extracted functions into compatible groups using the Classify method (Line

1) such that all functions in each group are compatible. The method classifies as many distinct compatible functions in each group as possible. The Classify method classifies functions by adding them iteratively into compatible groups. When it comes across a function f that cannot be classified into existing groups, it creates a new compatible group (Lines 16–17) and iteratively adds other functions compatible with this new group to this new group (Lines 18–22). Note that a function can be classified into multiple compatible groups. For example, we can obtain two compatible groups from CellArray2 in Figure 1(a): (1) $f(x_1, 0) = x_1$, $f(x_1, x_2) = x_1 + x_2$ and $f(0, x_2) = x_2$; (2) $f(x_1, 0) = x_1$ and $f(x_1, x_2) = x_1 - x_2$.

To tackle the third challenge, Algorithm 2 synthesizes $f_{pattern}$ candidates in two steps. The two-step synthesis is motivated by two observations: (1) the inclusion of input-output pairs derived from incorrect data cells can result in unsuccessful synthesis of $f_{pattern}$ candidates, but we have no prior knowledge of which data cells are incorrect; (2) the additional constraints of input-output pairs are useful for pruning inappropriate $f_{pattern}$ candidates. In the first step, the algorithm utilizes the constraints provided by functions in each compatible group to synthesize $f_{pattern}$ candidates with the SynFPattern method (Line 5). The method is implemented to follow the component-based synthesis technique [14] by treating functions as specification inputs. It gives an $f_{pattern}$ candidate set for each compatible group. If the functions in one group are not restrictive enough, the set can contain multiple candidates. In other words, the functions in the group collectively constitute only a partial specification for $f_{pattern}$ synthesis. The algorithm, therefore, includes a second step to enrich the specification with additional constraints given by the input-output pairs using the Refine method (Line 6). For each $f_{pattern}$ candidate set, the method iteratively prunes inappropriate candidates in the set using the input-output pairs of the given cell array while ignoring those pairs that lead to no solution. This relieves us from the need to identify incorrect data cells and exclude their associated input-output pairs. Details of this pruning process can be found in related work [21]. The Refine method returns an arbitrary one of $f_{pattern}$ candidates left behind in each set as its result. Finally, among all returned $f_{pattern}$ candidates, Algorithm 2 selects the one that covers cells in the given cell array most as the synthesized $f_{pattern}$ (Lines 7–10).

AmCheck infers an $f_{pattern}$ successfully if it can recover or synthesize the $f_{pattern}$ for a given smelly cell array. End users can use the inferred $f_{pattern}$ to repair the cells that it can cover in the cell array. Remaining cells that it cannot cover are error-prone.

5. IMPLEMENTATION

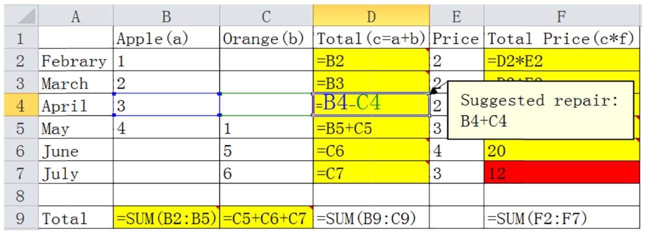
Our tool implementation of AmCheck, also named AmCheck for convenience, uses the Apache POI[1] library to read and modify Excel files. AmCheck loads an Excel file, analyzes its spreadsheet smells, and generates related comments explaining these ambiguous computation smells and corresponding repairs.

We implemented AmCheck in Java 7 and used Choco [22] as its underlying constraint solver. For user friendliness, AmCheck transforms an inferred $f_{pattern}$ back to a human-readable format, e.g., $x_1 + x_2$ is transformed into B2 + C2 for cell D2 in Figure 1(a). For visualization, AmCheck marks analysis results by three annotations: (1) Cell arrays that suffer from ambiguous computation smells are colored in yellow; (2) Spreadsheet comments are added to smelly cells for describing their corresponding repairs; (3) Conformance errors are colored in red with spreadsheet comments explaining their reasons. These annotations can assist end users to validate these reported results. Figure 3 gives a screenshot of the reported results for our motivating example in Figure 1(a).

[1] Apache POI: http://poi.apache.org/.

Algorithm 2. $f_{pattern}$ synthesis algorithm.

```
Input: IV (input variables), COMP (components), FUNC (func-
       tions), IO (input-output pairs), CA (cell array).
Output: F (target formula pattern).
 1:  groups = Classify(FUNC);  // Get compatible groups
 2:  pert = 0; F = NULL
 3:  while groups not EMPTY do
 4:    group = groups.removeOne();  // Retrieve one group
 5:    formulas = SynFPattern (IV, COMP, group)
 6:    formula = Refine(IV, COMP, formulas, IO)
 7:    if (formula≠NULL && Coverage(formula, CA)>pert) then
 8:      pert = Coverage(formula, CA);  // Measure percentage
 9:      F = formula
10:    end if
11:  end while
12:  return F
13:
14:  method Classify(FUNC)
15:    groups = EMPTY
16:    while (∃initFunc∈FUNC. initFunc non-classified) do
17:      newGroup = {initFunc}
18:      foreach func in FUNC\newGroup do
19:        if (!∃fn∈newGroup. ∃in. fn(in)≠func(in)) then
20:          newGroup.add(func)
21:        end if
22:      end for
23:      groups.add(newGroup);  // All in newGroup classified
24:    end while
25:    return groups
26:  end method
```



	A	B	C	D	E	F
1		Apple (a)	Orange (b)	Total (c=a+b)	Price	Total Price (c*f)
2	Febrary	1		=B2	2	=D2*E2
3	March	2		=B3	2	
4	April	3		=B4-C4	2	
5	May	4	1	=B5+C5	3	
6	June		5	=C6	4	20
7	July		6	=C7	3	12
8						
9	Total	=SUM(B2:B5)	=C5+C6+C7	=SUM(B9:C9)		=SUM(F2:F7)

Suggested repair:
B4+C4

Figure 3. AmCheck screenshot for the example in Figure 1(a).

6. EVALUATION

We evaluate AmCheck and study the following research questions:

RQ1: *How common are ambiguous computation smells in real-life spreadsheets?*

RQ2: *Can AmCheck detect and repair ambiguous computation smells precisely?*

RQ3: *Do end users find AmCheck useful for improving the quality of their spreadsheets in terms of detecting and repairing ambiguous computation smells?*

RQ4: *Are ambiguous computation smells harmful?*

To answer questions RQ1–2, we conducted an empirical study on the EUSES corpus [11]. We used AmCheck to detect ambiguous computation smells in this corpus, and manually validated 700 of them randomly selected from all results to determine whether they are true smells (§6.1). To answer question RQ3–4, we conducted a case study on real-life spreadsheets prepared by finance officers for research project budgeting in the Institute of Software, Chinese Academy of Sciences. In this study, we used AmCheck to detect ambiguous computation smells in these spreadsheets. We then interviewed finance officers who created or maintained these spreadsheets to understand why and how these smells have arisen from their spreadsheets (§6.2).

Table 1. Statistics of the EUSES corpus (n.a.: not applicable).

Category	Spreadsheets				Spreadsheets with ambiguous computation smells			
	Total	Processed	With formulas	With arrays	Smelly	Missing	Inconsistent	Percentage
cs101	8	8	8	7	3	2	1	42.9%
database	678	632	202	103	56	45	29	54.4%
filby	45	2	2	0	0	0	0	n.a.
financial	720	692	358	245	126	81	79	51.4%
forms3	26	19	18	10	4	3	1	40.0%
grades	588	557	285	201	88	66	42	43.8%
homework	576	535	278	163	54	35	30	33.1%
inventory	699	643	278	173	75	47	44	43.4%
jackson	13	0	0	0	0	0	0	n.a.
modeling	679	653	192	88	38	29	21	43.2%
personal	5	5	5	3	0	0	0	0%
Total	**4,037**	**3,746**	**1,626**	**993**	**444**	**308**	**247**	**44.7%**

6.1 Empirical Study on the EUSES Corpus

We ran AmCheck on all spreadsheets in the EUSES corpus and got experimental results about ambiguous computation smells.

6.1.1 *Experimental Subjects*

The EUSES corpus consists of 4,037 real-life spreadsheets from 11 categories. Since its creation in 2005, it has been widely used for spreadsheet research and evaluation. Table 1 gives the statistics of the corpus. It lists the number of spreadsheets (Total), the number of our processed spreadsheets (Processed), the number of spreadsheets with formulas (With formulas), and the number of spreadsheets with cell arrays (With arrays) for each category (Category). We find that only 92.8% (3,746/4,037) of spreadsheets in the corpus could be parsed by the Apache POI. 43.4% (1,626/3,746) of the processed spreadsheets contain cells with formulas. Out of them, 61.1% (993/1,626) contain cell arrays.

6.1.2 *Ambiguous Computation Smells*

Table 1 also gives the number of spreadsheets suffering from ambiguous computation smells (Smelly), the number of spreadsheets with missing formula smells (Missing), the number of spreadsheets with inconsistent formula smells (Inconsistent), and the percentage of smelly spreadsheets against all those with cell arrays. Note that a smelly spreadsheet can suffer from both types of smells simultaneously. As shown in Table 1, 44.7% (444/993) of the spreadsheets with cell arrays and 27.3% (444/1,626) of the spreadsheets with formulas suffer from at least one kind of ambiguous computation smell. This discloses that ambiguous computation smells are common in real-life spreadsheets.

Table 2 lists the number of cell arrays (CA), the number of well-formed cell arrays (WCA), and the number of smelly cell arrays (SCA). It also lists the number of cell arrays suffering from missing formula smells (MISS), inconsistent formula smells (INCO), and both smells (BO). It omits filby and jackson since they do not contain cell arrays. We observe that ambiguous computation smells occur commonly in the corpus: 21.6% (3,535/16,385) of identified cell arrays suffer from ambiguous computation smells. Among these smelly cell arrays, 75.3% (2,663/3,535) suffer from missing formula smells, 31.5% (1,113/3,535) suffer from inconsistent formula smells, and 6.8% (241/3,535) suffer from both smells. Therefore, we draw the following conclusion:

Ambiguous computation smells commonly occur in real-life spreadsheets, with missing formula smells occurring more often than inconsistent formula smells.

6.1.3 *Quality of AmCheck Analysis*

AmCheck is based on cell array identification and $f_{pattern}$ inference. This process partly relies on heuristics, so AmCheck may be imprecise and cause false positives. Thus, we are interested in its analysis quality.

Table 2. Smelly cell arrays in the EUSES corpus.

Category	CA	WCA	SCA	MISS	INCO	BO
cs101	31	21	10	7	3	0
database	2,252	1,714	538	438	113	13
financial	4,899	4,221	678	415	294	31
forms3	229	62	167	166	127	126
grades	1,893	1,426	467	381	111	25
homework	2,162	1,456	706	555	176	25
inventory	3,210	2,589	621	433	205	17
modeling	1,585	1,237	348	268	84	4
personal	124	124	0	0	0	0
Total	**16,385**	**12,850**	**3,535**	**2,663**	**1,113**	**241**

We partition detected smelly cell arrays into seven categories according to how much the inferred $f_{pattern}$ can cover cells in these arrays. These seven categories are: {100%, [90%, 100%), [80%, 90%), [70%, 80%), [60%, 70%), [50%, 60%) and [0%, 50%)}. Table 3 lists the number of smelly cell arrays (SCA) and the number of cells with conformance errors (CE1) for each category. We observe that the $f_{pattern}$ inferred by AmCheck is able to cover all cells in 903 smelly cell arrays (i.e., a coverage of 100%) and 90% or more (but not 100%) of cells in another 108 smelly cell arrays. This suggests that values and formulas in these 1,011 cell arrays are highly compatible with the inferred $f_{pattern}$. In other words, each of these 1,011 cell arrays that suffer from missing formulas or different formula patterns very likely prescribes a well-defined computational semantic expressible by the $f_{pattern}$ inferred by AmCheck. It is thus very likely that detected ambiguous computation smells in these cell arrays (1,011/3,535 = 28.6%) are probably true. This provides an alternative for assessing the quality of AmCheck's automatically detected spreadsheet smells. We can use these seven categories to rank the likeliness of a smelly cell array being true. Those cell arrays falling into the 100% category are considered most likely.

Next, we evaluate the precision of AmCheck's smell detection for the seven categories. We randomly selected 100 smelly cell arrays (CA) in each category and manually validated their quality. This accounts for 700 smelly cell arrays, which occupy 19.8% (700/3,535) of the whole. In each category, Table 3 lists the number of true smelly cell arrays (TP), the number of true smelly cell arrays that AmCheck can repair (RE), the number of true smelly cell arrays warned by Excel 2010 (EX), and the number of cells with true conformance errors (CE2). We explain these data below.

True positives and repairability. Out of the 700 sampled smelly cell arrays, we manually confirmed that 319 (45.6%) of them are true smelly cell arrays. We observe from the CA and TP columns

Table 3. Smelly cell arrays with different coverages.

	All		Sampled				
	SCA	CE1	CA	TP	RE	EX	CE2
100%	903	0	100	95/ 100	95/ 100	2/ 100	0/ 0
[90%, 100%)	108	133	100	73/ 100	73/ 100	7/ 100	94/ 121
[80%, 90%)	197	338	100	53/ 100	52/ 100	3/ 100	85/ 133
[70%, 80%)	120	242	100	46/ 100	46/ 100	0/ 100	101/ 190
[60%, 70%)	211	363	100	38/ 100	36/ 100	0/ 100	103/ 192
[50%, 60%)	917	1,392	100	9/ 100	9/ 100	0/ 100	37/ 135
[0%, 50%)	1,079	6,013	100	5/ 100	5/ 100	0/ 100	14/ 652
Total	**3,535**	**8,481**	**700**	**319/ 700**	**316/ 700**	**12/ 700**	**434/ 1,423**

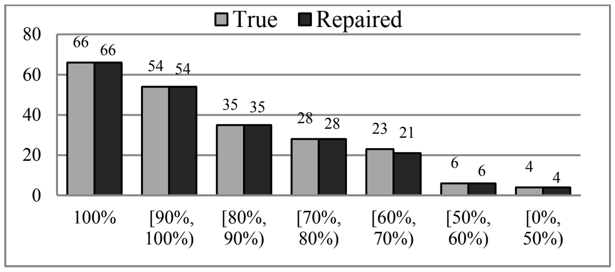
in Table 3 that the number of true smelly cell arrays decreases with the reduction in coverage. We recommend to select 80% as a coverage threshold for reliable detection of smelly cell arrays with an experimental precision of 73.7% (221/300).

AmCheck is able to repair 316 (99.1%) of the 319 true smelly cell arrays. It shows that AmCheck is effective for detecting and repairing smelly cell arrays automatically. Figure 4 elaborates the effectiveness with respect to the two different types of smells under various coverage categories. For each coverage category, it gives the number of true smelly cell arrays (True) and the number of true smelly cell arrays that can be repaired (Repaired) using an inferred $f_{pattern}$ for missing formula smells (Figure 4(a)) and inconsistent formula smells (Figure 4(b)). There are totally 10 cell arrays suffering from both missing formula and inconsistent formula smells. They are counted in both charts. AmCheck is able to repair 214 out of the 216 missing formula smells, and 112 out of the 113 inconsistent formula smells. The three smelly cell arrays that AmCheck failed to repair involves sophisticated library functions, incomplete input variables and complex structures. There is no enough evidence to suggest that AmCheck's repairing effectiveness and coverage categories are correlated.

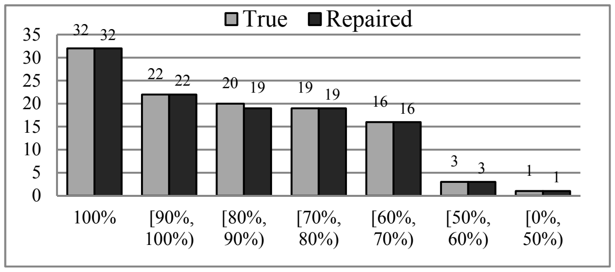
We draw the following conclusion:

AmCheck can effectively repair true ambiguous computation smells. One may use a coverage threshold of 80% for reliable detection of ambiguous computation smells.

False positives. The differences of values in the CA and TP columns in Table 3 give the number of false positives, which arise mostly because the cells concerned do not form a cell array as identified by AmCheck using the heuristics in §4.1. Two main causes are found: (1) Some spreadsheets use numbers as labels. For example, in financial reports end users often use years like 2011 and 2013 as labels but represented in a number format. Our heuristics can misinterpret them as data cells. 69.2% (92/133) of false positives in the coverage range of [70%, 100%] belong to this case. It should be easy for end users to recognize such false positives. (2) The heuristics incorrectly identify some cells with distinct formula patterns as those subject to the same computational semantics (but actually not). 13.5% (18/133) of false positives in the coverage range of [70%, 100%] belong to this case. End users can manually confirm or reject them in our current AmCheck implementation. For the remaining 23 (17.3%) false positives, the concerned cell arrays contain data cells with varying complex semantics, which AmCheck cannot effectively distinguish. We leave this to future work.



(a) Cell arrays with missing formula smells.



(b) Cell arrays with inconsistent formula smells.

Figure 4. Sampled smelly cell arrays with different coverages (x-axis: coverage category, y-axis: the number of cell arrays concerned).

6.1.4 *Conformance Errors*

Cells that do not conform to the inferred $f_{pattern}$ in a cell array are considered as conformance errors. In Table 3, AmCheck detected a total of 8,481 conformance errors (CE1) in the EUSES corpus. We manually validated the 1,423 detected conformance errors in the 700 sampled cell arrays. We confirmed that 30.5% (434/1,423) of them are true conformance errors (CE2), and all of them occur at the 319 true smelly cell arrays. Note that there can be multiple conformance errors in one smelly cell array. We also observe that conformance errors occurring at the cell arrays with higher coverage are more likely to be true. For example, 70.5% (179/254) of detected conformance errors are true in the coverage range of [80%, 100%].

6.1.5 *Comparison with Excel*

Although Microsoft Excel has built-in support for inconsistency detection in cell arrays, its detection is subject to a few limitations. First, Excel considers only cell arrays of three adjacent cells. Second, it detects only those smells that a cell's formula is syntactically different from those of its two adjacent cells when these two cells' formulas are identical. The EX column in Table 3 shows that Excel detected inconsistencies in only 3.8% (12/319) of the true smelly cell arrays.

6.1.6 *Runtime Overhead*

Running AmCheck over the whole EUSES corpus took about 116 minutes. It thus took about 1.7 seconds to analyze one spreadsheet on average. This good performance is attributed to the use of our two-stage analysis ($f_{pattern}$ recovery and $f_{pattern}$ synthesis). If we use $f_{pattern}$ synthesis only to infer formula patterns, it would take 861 minutes (12.8 seconds for each spreadsheet on average). Our two-stage analysis significantly reduces the computational time by 86.5%. Among the 3,535 detected smelly cell arrays, 93.2% (3,296/3,535) of them have their $f_{pattern}$ successfully recovered at the first stage, and the remaining 6.8% (239/3,535) have their $f_{pattern}$ synthesized at the second stage.

6.2 Case Study on Real-life Spreadsheets

To evaluate the usefulness of AmCheck to end users, we conducted a case study on ten real-life spreadsheets currently used by finance officers. We studied three questions: (1) Why do ambiguous computation smells occur in spreadsheets? (2) Is AmCheck helpful for detecting and repairing ambiguous computation smells? (3) How harmful are ambiguous computation smells?

Setup. We conducted our study in the Institute of Software, Chinese Academy of Sciences. We collected ten real-life spreadsheets professionally prepared by finance officers for research project budgeting. These spreadsheets involve seven departments or units, and have been used for more than three years. Most spreadsheets are maintained by more than two officers.

We invited three officers who have participated in maintaining these spreadsheets for an interview. We explained ambiguous computation smells to these officers, and told them our plan of using their spreadsheets in the study. We gave a list of AmCheck's detected ambiguous computation smells in these spreadsheets as well as suggested repairs with brief explanations. We let the officers study these smells before our interview.

In the interview, we asked the officers three questions: (1) Can you explain how these ambiguous computation smells arise? (2) Do you agree that these smells are indeed problems? (3) Are the suggested repairs helpful?

Overall results. For each spreadsheet, Table 4 lists the number of cell arrays (CA), the number of well-formed cell arrays (WCA), the number of smelly cell arrays (SCA), the number of cell arrays suffering from missing formula smells (MISS), the number of cell arrays suffering from inconsistent formula smells (INCO), and the number of cells with conformance errors (CE). The numbers in brackets are confirmed data by officers. We observe that 80.0% (8/10) of the spreadsheets suffer from ambiguous computation smells. We detected 55 smelly cell arrays (40 suffering from missing formula smells and 17 suffering from inconsistent formula smells; some suffering from both smells), and 23 cells contain conformance errors. The results reveal that ambiguous computational smells are common to financial spreadsheets that have been rigorously prepared and maintained by end users for over three years. 40% of the spreadsheets contain even confirmed errors. 20.4% (55/270) of identified cell arrays suffer from ambiguous computation smells, and this percentage is comparable to that for the EUSES corpus (21.6%). The ratio 2.4 (40/17) between number of smelly cell arrays suffering from missing formula smells and number of smelly cell arrays suffering from inconsistent formula smells is also comparable to that for the EUSES corpus, 2.4 (2,663/1,113). Ratios of true positives are higher than those in the EUSES corpus. This is because the aforementioned false positives are rare in these spreadsheets. Therefore, we draw the following conclusion:

The ratio of smelly cell arrays in real-life spreadsheets is comparable to that for the EUSES corpus.

Causes of ambiguous computation smells. From the interview with officers, we obtained interesting answers about how missing formula smells arose. They recalled: "I copied some data from another place, and did not notice that they can be computed from related cells", "I just wrote down these data as plain values", or "after I used the auto-fill feature to generate these cells (in a cell array), I noticed that there was an error of 'division by zero', and so I set its value to 0". From these answers, missing formula smells have been usually caused by carelessly ignoring necessary computation.

Table 4. Detected smelly cell arrays in the case study.

ID	CA	WCA	SCA	MISS	INCO	CE
1	12	12	0 (0)	0 (0)	0 (0)	0 (0)
2	24	24	0 (0)	0 (0)	0 (0)	0 (0)
3	17	9	8 (8)	7 (7)	3 (3)	4 (4)
4	32	12	20 (20)	14 (14)	6 (6)	8 (8)
5	32	29	3 (3)	2 (2)	1 (1)	0 (0)
6	32	29	3 (3)	2 (2)	1 (1)	0 (0)
7	10	9	1 (0)	1 (0)	0 (0)	1 (0)
8	32	29	3 (3)	2 (2)	1 (1)	0 (0)
9	50	45	5 (3)	3 (1)	2 (2)	1 (1)
10	29	17	12 (10)	9 (9)	3 (1)	9 (7)
Total	**270**	**215**	**55 (50)**	**40 (37)**	**17 (15)**	**23 (20)**

Table 5. Undetected cell arrays in the case study (numbers in brackets are false positives).

ID	CA	Undetected cell arrays					
		DRC	PV	INS	CC	CON	DS
1	12	5			1		4
2	24						
3	17		6				
4	32			2		1	
5	32				1		8
6	32						9
7	10 (1)						
8	32						9
9	50 (2)			1			
10	29 (2)		2				1
Total	**270 (5)**	**5**	**8**	**3**	**2**	**1**	**31**

When coming to the question of how inconsistent formula smells arose, they recalled: "I copied formulas from another spreadsheet, and I really do not know why smells occur", or "I wrote formulas by myself and did not use any auto-fill feature, but I missed some input cells". For example, function SUM has been used inconsistently when some cells contain a value of zero. Another example is: a cell array should have a formula pattern of R[-3]C * R[-2]C * R[-1]C, but some of its cells have a formula like R[-3]C * R[-1]C. This occurs when R[-2]C's value happens to be 1. From these interview answers, we draw the following conclusion:

Ambiguous computation smells have been often caused by carelessly using the auto-fill and copy-and-paste features. AmCheck can effectively detect and repair such smells.

Harmfulness of ambiguous computation smells. Interestingly, the officers were surprised by so many detected ambiguous computation smells. Although some concerned cells had temporarily correct values, they still decided to take our repair suggestions. They said: "these cells (suffering from ambiguous computation smells) are risky, and are difficult to maintain in future".

Although the studied spreadsheets have been verified by our financial officers carefully during their daily jobs, conformance errors still exist. When such errors were presented to them, they said: "the final result is balanced, and it should be impossible!". After we repaired the conformance errors, we noticed that the final result now became unbalanced. Weird! Finally, we figured out why: there was another missing formula smell in a related cell. Although this missing formula (not related to cell arrays) could not be detected by the current AmCheck implementation, its reported smell warning enables end users to eventually find the hidden problem successfully. These conformance errors have been caused by careless updates to cells in cell arrays. With AmCheck's repairs, our officers realized that existing values of some cells are indeed faulty. From these interview interactions, we draw the following conclusion:

Ambiguous computation smells are harmful and have caused data discrepancies in real-life spreadsheets.

6.3 Discussions

While the evaluation shows that our AmCheck is promising for detecting and repairing ambiguous computation smells in real-life spreadsheets, we discuss some of its limitations.

6.3.1 False Negatives

False negatives in detecting smelly cell arrays are mainly caused by undetected cell arrays. With help from finance officers, we further measured such undetected cell arrays in our case study.

For each spreadsheet, Table 5 lists the number of detected cell arrays (CA) and the number of undetected cell arrays (Undetected cell arrays). AmCheck misses 15.9% (50 / (270 – 5 + 50)) of all cell arrays in total. The recall rate of cell array extraction in AmCheck is 84.1% ((270 – 5) / (270 – 5 + 50)). Several reasons caused such false negatives: (1) Some cells reference other cells at different rows or columns (DRC); (2) Some cell arrays contain plain values only (PV); (3) Some cell arrays have in-line space (INS); (4) Some cell arrays contain constant cells without being labeled with "$" (a special annotation indicating a constant cell) (CC); (5) Some cell arrays contain conditional formulas (CON); (6) Some cells reference other cells in different spreadsheets (DS). AmCheck needs extensions for handling these cases.

6.3.2 Threats to Validity

One threat to internal validity of our evaluation is that we were unable to validate analysis results of spreadsheets in the EUSES corpus by their original users. As such, we validated the results by ourselves partially and manually in due diligence. Another threat to external validity of our evaluation concerns the representativeness of spreadsheets in the EUSES corpus and collected in our case study. We chose the EUSES corpus because it is by far the largest corpus that has been widely used for evaluation by previous spreadsheet research studies. Spreadsheets collected in our case study are those used in practice and maintained by professional finance officers. We made best effort in choosing representative and real-life experimental subjects.

7. RELATED WORK

In this section, we review related work in recent years.

Spreadsheet errors. Spreadsheet errors are common [27][28][29]. They can cause serious financial losses. Ambiguous computation smells may not cause errors immediately but degrade spreadsheet quality gradually. Spreadsheets suffering from ambiguous computation smells contain unclear computational semantics, which make them difficult to maintain in a correct way.

Detecting faults in spreadsheets. Various techniques have been proposed to detect faults in spreadsheets. UCheck [4] and dimension inference [7] use the type system to check unit faults and dimension faults, respectively. They focus on whether units can be combined correctly into one cell. Smellsheet Detective [9][10] detects statistical smells, type smells, content smells and functional dependency smells. Hermans et al. proposed visualizing spreadsheets by dataflow graphs [17], and detected inter-worksheet smells in these graphs [18]. They proposed detecting smells from data clones [20] and in spreadsheet formulas [19]. In these pieces of work, Hermans et al.'s [19] and Smellsheet Detective [10] focus on syntactic faults, while our work focus on missing formula and inconsistent formula smells, which concern semantic faults. Our work also detects conformance errors caused by ambiguous computation smells. Its scope is thus orthogonal to existing work.

Program synthesis. Our work is based on component-based program synthesis [14][21]. Typically, end users should provide components and input-output pairs for program synthesis [21]. Regarding our problem, we automatically extract such components and input-output pairs from spreadsheets and alleviate their noises. Program synthesis has also been used for other purposes in spreadsheet research, e.g., string transformation from examples [13], table transformation [15], and number transformation [32]. In this paper, we apply program synthesis in a novel way to detect and repair ambiguous computation smells in spreadsheets by recovering hidden computational semantics.

Modeling and testing for spreadsheets. Construction of rigorous models for spreadsheets [1][8][16] can help end users reduce chances of introducing ambiguous computation smells. Inferring such models from spreadsheets can be challenging. Its effectiveness depends on correctness of spreadsheets, and ambiguous computation smells can reduce its precision. Our work addresses the problem effectively by using both heuristics and formula pattern synthesis. Spreadsheet testing [5][12][23] is another interesting topic, whose most challenge may be the lack of test oracles. Our work extracts partial computational semantics from cell contents and recovers hidden formula patterns. It does not require explicit test oracles. Ambiguous computation smells may also mislead spreadsheet testing, and our work can assist the testing by repairing smelly spreadsheets.

Semantic faults in software. Similar to smells in spreadsheets, semantic faults are also dominant root causes of software failures [24][33]. Most semantic faults require domain knowledge to understand, detect and repair [24]. MUVI [25] and DefUse [31] can detect semantic faults related to inconsistent updates to correlated multi-variables and dataflow intentions, respectively, in software. They rely on invariant mining and detection techniques. Our work uses a different approach by inferring hidden computational semantics by heuristics and program synthesis techniques.

8. CONCLUSION

In this paper, we study ambiguous computation smells in spreadsheets, which are caused by end users' ad hoc modifications to spreadsheet cells that should stick to certain computational semantics. We propose a novel approach, AmCheck, to detect and repair ambiguous computation smells by inferring formula patterns for smelly cell arrays in spreadsheets. This also helps detect challenging conformance errors in spreadsheets. Our evaluation with real-life spreadsheets reports that ambiguous computation smells are common and harmful, and end users do care about such smells and conformance errors caused by such smells, and value AmCheck for its ability of automatically detecting and repairing smelly spreadsheets.

We have identified several future research directions. First, detecting smelly cell arrays can be improved by more precise cell array extraction and formula pattern inference. Second, AmCheck can be improved by implementing more features like handling conditional formula patterns. Third, we plan to conduct more real-life case studies and investigate mechanisms to prevent ambiguous computation smells in spreadsheets.

9. ACKNOWLEDGMENTS

We sincerely thank Chang Xu, Yepang Liu and Rongxin Wu for their valuable comments and careful proofreading. The work was supported in part by Research Grants Council (611811) of Hong Kong, and by National High Technology Research & Development 863 Program (2012AA011204) and National Natural Science Foundation (61173005, 61100038) of China.

10. REFERENCES

[1] R. Abraham and M. Erwig. Inferring templates from spreadsheets. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 182–191. 2006.

[2] R. Abraham and M. Erwig. AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 43 –50. September 2006.

[3] R. Abraham and M. Erwig. GoalDebug: A Spreadsheet Debugger for End Users. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 251–260. 2007.

[4] R. Abraham and M. Erwig. UCheck: A spreadsheet type checker for end users. *J. Vis. Lang. Comput.*, 18(1):71–95, February 2007.

[5] R. Abraham and M. Erwig. Mutation Operators for Spreadsheets. *IEEE Trans. Softw. Eng.*, 35(1):94 –108, February 2009.

[6] M. Burnett and M. Erwig. Visually customizing inference rules about apples and oranges. In *IEEE Symposia on Human Centric Computing Languages and Environments (HCC)*, pages 140–148. 2002.

[7] C. Chambers and M. Erwig. Automatic detection of dimension errors in spreadsheets. *J. Vis. Lang. Comput.*, 20(4):269–283, August 2009.

[8] J. Cunha, M. Erwig, and J. Saraiva. Automatically Inferring ClassSheet Models from Spreadsheets. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 93–100. 2010.

[9] J. Cunha, J.P. Fernandes, P. Martins, J. Mendes, and J. Saraiva. SmellSheet detective: A tool for detecting bad smells in spreadsheets. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 243–244. 2012.

[10] J. Cunha, J.P. Fernandes, H. Ribeiro, and J. Saraiva. Towards a Catalog of Spreadsheet Smells. In *Computational Science and Its Applications*, pages 202–216. Springer Berlin Heidelberg, 2012.

[11]M. Fisher and G. Rothermel. The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. *SIGSOFT Softw Eng Notes*, 30(4):1–5, May 2005.

[12] G. Rothermel, L. Li, C. Dupuis, and M. Burnett. What you see is what you test: a methodology for testing form-based visual programs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 198–207. April 1998.

[13] S. Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN*, 46(1):317–330, January 2011.

[14] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. *SIGPLAN*, 46(6):62–73, June 2011.

[15] W.R. Harris and S. Gulwani. Spreadsheet table transformations from examples. *SIGPLAN*, 46(6):317–328, June 2011.

[16] F. Hermans, M. Pinzger, and A. van Deursen. Automatically extracting class diagrams from spreadsheets. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*, pages 52–75. Springer-Verlag, 2010.

[17] F. Hermans, M. Pinzger, and A. van Deursen. Supporting professional spreadsheet users by generating leveled dataflow diagrams. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 451–460. 2011.

[18] F. Hermans, M. Pinzger, and A. van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 441–451. 2012.

[19] F. Hermans, M. Pinzger, and A.V. Deursen. Detecting Code Smells in Spreadsheet Formulas. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 409–418. 2012.

[20] F. Hermans, B. Sedee, M. Pinzger, and A. van Deursen. Data clone detection and visualization in spreadsheets. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 292–301. 2013.

[21] S. Jha, S. Gulwani, S.A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ACM/IEEE 32nd International Conference on Software Engineering (ICSE)*, pages 215–224. 2010.

[22] N. Jussien, G. Rochart, and X. Lorca. The CHOCO constraint programming solver. In *CPAIOR workshop on Open-Source Software for Integer and Contraint Programming (OSSICP)*. 2008.

[23] K.J. Rothermel, C.R. Cook, M.M. Burnett, J. Schonfeld, T.R.G. Green, and G. Rothermel. WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 230 –239. 2000.

[24] L. Lu, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, and S. Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. 2013.

[25] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R.A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP)*, pages 103–116. 2007.

[26] R. Panko. Facing the problem of spreadsheet errors. *Decis. Line*, 37(5)2006.

[27] R.R. Panko and S. Aurigemma. Revising the Panko–Halverson taxonomy of spreadsheet errors. *Decis. Support Syst.*, 49(2):235–244, May 2010.

[28] S.G. Powell, K.R. Baker, and B. Lawson. A critical review of the literature on spreadsheet errors. *Decis. Support Syst.*, 46(1):128–138, December 2008.

[29] K. Rajalingham, D.R. Chadwick, and B. Knight. Classification of Spreadsheet Errors. *CoRR*, 2008.

[30] J. Reichwein, G. Rothermel, and M. Burnett. Slicing spreadsheets: an integrated methodology for spreadsheet testing and debugging. *SIGPLAN*, 35(1):25–38, December 1999.

[31] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition?: DefUse: definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 160–174. 2010.

[32] R. Singh and S. Gulwani. Synthesizing Number Transformations from Input-Output Examples. In *Computer Aided Verification (CAV)*, pages 634–651. January 2012.

[33] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empir. Softw. Eng.*, :1–41, 2013.

[34] J. Walkenbach. *Excel 2013 Power Programming with VBA*. Wiley. com, 2013.

[35] How to use the Auto Fill Options button in Excel. http://support.microsoft.com/kb/291359 .