CACheck: Detecting and Repairing Cell Arrays in Spreadsheets

Wensheng Dou, Chang Xu, S.C. Cheung, and Jun Wei

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 as a cell array. When a spreadsheet evolves, the cells in a cell array can degenerate due to ad hoc modifications. Such degenerated cell arrays no longer keep cells prescribing the same computational semantics, and are said to exhibit ambiguous computation smells. We propose CACheck, a novel technique that automatically detects and repairs smelly cell arrays by recovering their intended computational semantics. Our empirical study on the EUSES and Enron corpora finds that such smelly cell arrays are common. Our study also suggests that CACheck is useful for detecting and repairing real spreadsheet problems caused by smelly cell arrays. Compared with our previous work AmCheck, CACheck detects smelly cell arrays with higher precision and recall rate.

Index Terms—Spreadsheet, cell array, ambiguous computation smell

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 [45]. Spreadsheet errors can induce great financial losses [43], [54]. Various techniques have been proposed to improve the quality of spreadsheets. Some examples include testing [1], [20], [37], error or smell detection [6], [7], [29], [31], debugging [3], [47], and auditing [11], [12], [16].

A spreadsheet comprises clusters of cells arranged in rows and columns. We refer to a cell cluster that contains cells as a *cell array* when these cells are subject to the same computational semantics. For example, the cells [D2:D7] in Fig. 1(b) implement the semantics of "Total" and uniformly follow a formula pattern of $D_i = B_i + C_i$, where $2 \le i \le 7$. Cells in a cell array are usually copy-equivalent [11], although there are inconsistent cases. In our empirical study, we found 591,413 non-equivalent cell arrays in the 19,963 spreadsheets in the EUSES [19] and Enron [26] corpora, which are the two mostly cited spreadsheet corpora so far. 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 [46]. Spreadsheet software like Excel provides two useful editing 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 [55], and apply it to the new cells in a cell array.

Although these 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 the deduced formula patterns. There are rare records (e.g., copy-paste tracking in XanaSheet [32]) in the new cells documenting that they have been created using these two features, and therefore have to be consistently 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 semantics 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 Fig. 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 is issued by Excel 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 each cell in this cell array [D2:D7] is evaluated to a correct value, it can degenerate into errors upon future updates of entries in columns B and C. For example, the value in D2 would be incorrect if the value of C2 is later updated to a non-zero value. As ambiguous computation smells are vulnerable to errors, their early detection is important. It is particularly the case for those spreadsheets that have liability consequences such as company financial reports.

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

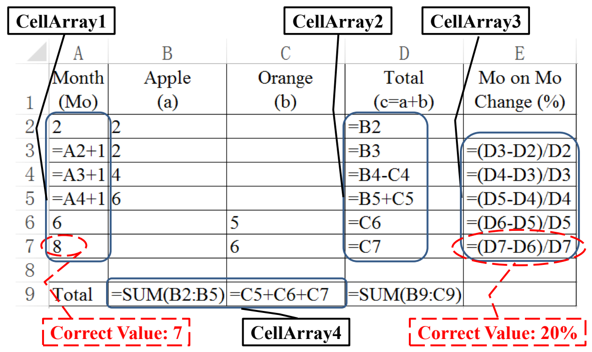
- *W. Dou and J. Wei are with the State Key Laboratory of Computer Science, the Institute of Software, Chinese Academy of Sciences, Beijing, China. E-mails: {wsdou, wj}@otcaix.iscas.ac.cn.*
- *C. Xu is with the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, China. E-mails: changxu@nju.edu.cn.*
- *S.C. Cheung is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China. E-mail: scc@cse.ust.hk*

to issue any warning for the cell array [D2:D7] in Fig. 1(a). Besides, UCheck [4] and dimension inference [8] exploit information about labels and headers in spreadsheets to check the type consistency of formulas. Since each cell in the cell array [D2:D7] in Fig. 1(a) does not have any type inconsistency, the smell cannot be detected by UCheck or dimension inference, either. Some commonly used commercial spreadsheet tools (e.g., Spreadsheet Professional [56], OAK [57], EXChecker [58] and PerfectXL [59]) consider cells D4 and D5 as errors with not quite relevant explanations (e.g., due to the fact that the two cells reference empty cells like C4 and C5). Some tools (e.g., OAK, EXChecker, Spreadsheet Detective [60] and Spreadsheet Auditor [61]) adopt Clermont et al. [11]'s idea to annotate copy-equivalent cells with colors, and assist end users to locate potential inconsistencies. However, they do not automatically detect the smell in the cell array [D2:D7] in Fig. 1(a).

Like semantic bugs in programming languages [39], [51], 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.

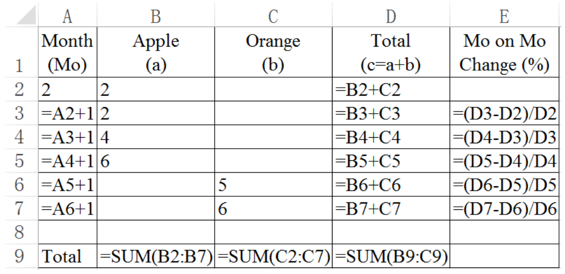
In this article, we focus on *numeric cells* whose numeric value is either computed by a formula or given directly without computation. Examples of these are cells A5 and A6 in Fig. 1(a). We study the automated extraction of cell arrays from numeric cells as well as the automated detection and repairing of those cell arrays that suffer from ambiguous computation smells. Cells that are subject to the same evaluation in a cell array are realized by the same formula pattern. We found that 17.3% of consecutive numeric cells share the same formula pattern while 82.7% do not. The first key challenge is to identify which of these 82.7% are cells belonging to some cell arrays even they do not share the same formula pattern with their neighbors. Once a cell array is identified to suffer from ambiguous computation smells, the repair action is to infer an appropriate formula pattern so that its cells are subject to the same evaluation. The second key challenge is how to infer appropriate formula patterns for repairing smelly cell arrays that suffer from ambiguous computation smells. Our approach automatically extracts computational semantics from cells in a cell array, recovers its formula pattern, and further detects smells in its contained cells. Thus, our approach could detect smells without human judgments. Fig. 1(b) shows a possible repairing of the spreadsheet in Fig. 1(a).

We evaluated our approach (CACheck) from three perspectives. First, we analyzed the EUSES [19] corpus to learn how often smelly cell arrays can occur, and measured the precision and recall rate of our approach for detecting such cell arrays. Second, we compared CACheck with our earlier version AmCheck in precision and recall on the EUSES corpus. Third, we further analyzed a more recent and industrial corpus Enron [26] to see whether we can obtain results similar to those obtained from the EUSES corpus. Our evaluation reports that: (1) 1,586 cell arrays in the EUSES corpus suffer from ambiguous computation smells. They cover 7.8% identified cell arrays. (2) Smelly cell arrays



	A	B	C	D	E
1	Month (Mo)	Apple (a)	Orange (b)	Total (c=a+b)	Mo on Mo Change (%)
2	2	2		=B2	
3	=A2+1	2		=B3	=(D3-D2)/D2
4	=A3+1	4		=B4-C4	=(D4-D3)/D3
5	=A4+1	6		=B5+C5	=(D5-D4)/D4
6	6		5	=C6	=(D6-D5)/D5
7	8		6	=C7	=(D7-D6)/D7
8					
9	Total	=SUM(B2:B5)	=C5+C6+C7	=SUM(B9:C9)	

(a) A spreadsheet with ambiguous computation smells.



	A	B	C	D	E
1	Month (Mo)	Apple (a)	Orange (b)	Total (c=a+b)	Mo on Mo Change (%)
2	2	2		=B2+C2	
3	=A2+1	2		=B3+C3	=(D3-D2)/D2
4	=A3+1	4		=B4+C4	=(D4-D3)/D3
5	=A4+1	6		=B5+C5	=(D5-D4)/D4
6	=A5+1		5	=B6+C6	=(D6-D5)/D5
7	=A6+1		6	=B7+C7	=(D7-D6)/D6
8					
9	Total	=SUM(B2:B7)	=SUM(C2:C7)	=SUM(B9:C9)	

(b) The correct version of the spreadsheet in (a).

Fig. 1. A motivating example: the four cell arrays in (a) are ambiguous; for each cell array, its contained cells do not all follow the same formula pattern, e.g., the cells in CellArray2 do not uniformly follow the formula pattern of $D_i = B_i + C_i$ ($2 \le i \le 7$); the spreadsheet in (b) gives the correct version.

reveal weakness and can cause errors in spreadsheets. 8,139 cells in the 1,586 smelly cell arrays were decided as smelly. They contain either wrong values or formulas. This number (8,139) occupies 25.9% of all cells in 1,586 smelly cell arrays, or 3.3% of all cells in 20,320 cell arrays. (3) CACheck can detect 98.3% of the smelly cell arrays detected by AmCheck, and 401 (out of 1,586) additional smelly cell arrays that are missed by AmCheck. CACheck also has a higher precision (86.8% vs. 71.9%) and recall rate (71.0% vs. 60.3%) than AmCheck. Other existing spreadsheet smell detection techniques (e.g., Excel, UCheck/Dimension [4], [8] and CUSTODES [10]) can detect at most 37.4% of CACheck's detected smelly cells. (4) CACheck has comparable precision (86.8% vs. 87.2%) and recall rate (71.0% vs. 72.7%) of smelly cell array detection on the EUSES and Enron corpora. Our approach can help end users detect and repair such smells, thus improving the quality of their spreadsheets.

We made the following main contributions in this article:

- We empirically study the characteristics of cell arrays in two spreadsheet corpora (EUSES and Enron). This study identifies several key observations on cell arrays.
- We propose a novel approach, CACheck, to detect and repair smelly cell arrays 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 implement CACheck as a tool and evaluate it experimentally on the EUSES and Enron corpora.

Compared with our previous work AmCheck, CACheck detects smelly cell arrays with higher precision (86.8% vs. 71.9%) and recall rate (71.0% vs. 60.3%).

An earlier version of this work (AmCheck) appeared at ICSE 2014 [17]. In this article (CACheck), we extend the earlier version in five aspects. (1) We conduct an empirical study on well-formed cell arrays in the EUSES and Enron corpora (Section 4). The study finds two common structures by which spreadsheet cells with formulas are organized: *homogeneous cell arrays* and *inhomogeneous cell arrays*. In a *row-/column*-based *homogeneous cell array*, the formula of each cell consistently references cells from the same *column/row* as this cell, such as the column-based cell array [D2:D7] in Fig. 1(b). In a *row-/column*-based *inhomogeneous cell array*, the formula of each cell may reference cells from *columns/rows* different from this cell, such as the column-based cell arrays [A2:A7] and [E3:E7] in Fig. 1(b). (2) AmCheck can only detect homogeneous cell arrays. However, CACheck can detect both homogeneous and inhomogeneous cell arrays (Section 5.1), greatly improving its scope. (3) We make several observations on cell arrays in our study. CACheck leverages these observations to filter out wrongly identified cell arrays, improving the precision of cell array identification (Section 5.7). (4) AmCheck was only evaluated on the EUSES corpus, while in this work, we evaluate CACheck on both EUSES [19] and Enron [26] corpora (Section 6), for more comprehensive comparison. (5) Compared with AmCheck, CACheck provides higher precision (86.8% vs. 71.9%) and recall rate (71.0% vs. 60.3%).

The remainder of this article is organized as follows. Section 2 gives a motivating example and explains the use of our technique. Section 3 defines and explains necessary concepts like cell array and ambiguous computation smell. Section 4 presents our empirical study on the EUSES and Enron corpora. Section 5 elaborates on our smell detection and repairing technique. Section 6 evaluates CACheck with the EUSES and Enron corpora. Section 7 discusses related work, and finally Section 8 concludes this article.

2 Motivation

In this section, we illustrate smelly cell arrays using an example spreadsheet stemming from the EUSES corpus [19]. We then explain how to detect and repair such smelly cell arrays.

2.1 Example

Fig. 1(a) shows a spreadsheet that computes monthly harvest of fruits. A cell array, which consists of numeric cells, can exhibit 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 a smell can be introduced to a cell array when end users override the formula in a cell with a plain value. For example, CellArray1 [A2:A7] is subject to the computation of "Month" with an intended formula pattern of $A_i = A_{i-1} + 1$, where $3 \le i \le 7$ (we use subscript to represent row number). Unlike cells A3, A4 and A5, the values of cells A6 and A7 are not computed by formulas. Note that, the first cell A2 in CellArray1 gives a base value, which is not computed by a formula.

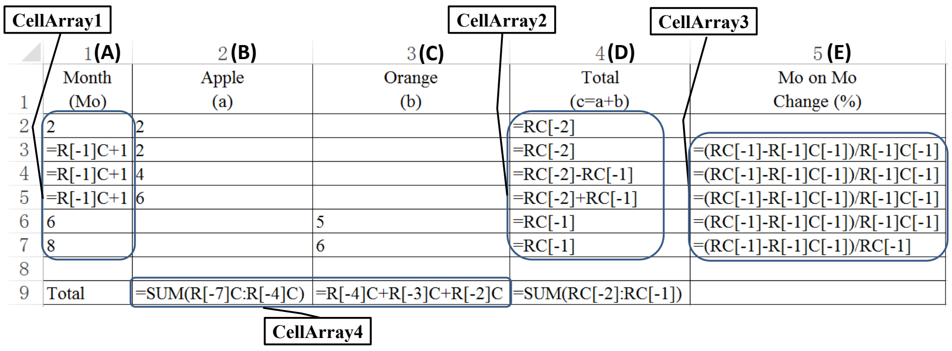
Inconsistent formula smell. This ambiguous computation smell occurs when the cells in a cell array prescribe different formula patterns. Such a smell can be introduced to a cell array when end users specify the formula of a cell in the cell array inappropriately without preserving the cell array's computational 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. Inconsistent formula smells also occur at CellArray3 [E3:E7] and CellArray4 [B9:C9].

Although CellArray2 and CellArray4 in Fig. 1(a) suffer from ambiguous computation smells, the values given by their 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 or auto-fill operations to these cell arrays later. A cell array suffering from ambiguous computation smell likely contains an error (e.g., A7 and E7) if no formula patterns can be found to compute the values in it.

2.2 CACheck Overview

Several technical challenges need to be addressed in the detection and repairing of cell arrays with ambiguous computation smells in spreadsheets. We explain them using the example in Fig. 1(a). First, does a cell (e.g., A3) belong to a cell array? If yes, does this cell belong to a row-based cell array (e.g., [A3:B3]) or column-based cell array (e.g., [A2:A7])? What are other cells for this cell array? Second, do the cells in a cell array prescribe semantically different formula patterns? Note that we consider two formula patterns (e.g., $x + x$ and $2*x$) to be the same if the formulas derived from these patterns offer the same computation. 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 CellArray4. 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., A6) in a cell array may prescribe no formula. The values of these cells (e.g., A7) may even conflict with their appropriate formula patterns.

In our earlier work AmCheck [17], we addressed the challenge of cell array extraction by assuming that a cell array's orientation (row-based or column-based) and its contained cells are determined by its referenced cells, i.e., homogeneous cell arrays (each cell in a row-/column-based cell array references only the cells that share the same column/row as this cell). However, our empirical study reveals that a significant amount (21.0%) of cell arrays are inhomogeneous, which AmCheck fails to extract. For example, AmCheck does not work for CellArray3 [E3:E7] (column-based cell array), because cell E3 references D2, which does not share the same row as E3. Even



	1(A)	2(B)	3(C)	4(D)	5(E)
1	Month (Mo)	Apple (a)	Orange (b)	Total (c=a+b)	Mo on Mo Change (%)
2	2	2		=RC[-2]	
3	=R[-1]C+1	2		=RC[-2]	=(RC[-1]-R[-1]C[-1])/R[-1]C[-1]
4	=R[-1]C+1	4		=RC[-2]-RC[-1]	=(RC[-1]-R[-1]C[-1])/R[-1]C[-1]
5	=R[-1]C+1	6		=RC[-2]+RC[-1]	=(RC[-1]-R[-1]C[-1])/R[-1]C[-1]
6	6		5	=RC[-1]	=(RC[-1]-R[-1]C[-1])/R[-1]C[-1]
7	8		6	=RC[-1]	=(RC[-1]-R[-1]C[-1])/RC[-1]
8					
9	Total	=SUM(R[-7]C:R[-4]C)	=R[-4]C+R[-3]C+R[-2]C	=SUM(RC[-2]:RC[-1])	

Fig. 2. The earlier spreadsheet in Fig. 1(a) is now given in the R1C1 representation style, in which four cell arrays are ambiguous, e.g., cells in CellArray2 do not have semantically equivalent formulas in the R1C1 representation style.

worse, AmCheck would assume [A3:B3], [A4:B4] and [A5:B5] as candidate cell arrays, but we can see that they are not true cell arrays. The true cell array should be CellArray1 [A2:A7]. Note that the cell array [A2:A7] cannot be extracted by AmCheck because it is column-based and each of its cells references a cell in another row (different from this cell). Therefore, AmCheck could miss true cell arrays, and introduce false cell arrays.

Our CACheck adopts new heuristics to extract cell arrays, and does not assume their orientations in advance. The basic idea is that if two adjacent cells share the same input dependence, they would belong to the same cell array. This relaxed constraint helps detect more cell arrays, such as CellArray1 [A2:A7], [A3:B3], [A4:B4] and [A5:B5]. However, which cell arrays are true? We did an empirical study on real-life cell arrays to understand how cell arrays are used (e.g., how cell arrays are structured in spreadsheets). Then, we leverage the observations (e.g., non-equivalent cell arrays rarely overlap) from this empirical study to filter out wrongly identified cell arrays, such as [A3:B3], [A4:B4] and [A5:B5] (i.e., these cell arrays overlap with the true one [A2:A7]).

CACheck 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. CACheck uses component-based program synthesis [22], [36] to construct candidate formula patterns for repairing smelly cell arrays. To achieve this, CACheck needs to cope with the noises induced by conflicting formulas (e.g., D4 and D5) and potential errors (e.g., A7). For example, CACheck can construct a candidate formula pattern (B_i + C_i, where $2 \le i \le 7$) to repair the smelly cell array CellArray2 in Fig. 1(a), and a candidate formula pattern (SUM(X_2, X_3, X_4, X_5) + X_6 + X_7, where $X$ = {B, C}) to repair the smelly cell array CellArray4. It can also use its inferred formula patterns to detect errors (e.g., A7 and E7) in smelly cell arrays.

3 Cell Arrays and Ambiguous Computation Smells

In this section, we introduce the spreadsheet programming model, and explain key concepts such as cell array and ambiguous computation smell for subsequent discussions. To ease presentation, we refer *data cells* to those cells whose numeric values are given directly without computation and *formula cells* to those cells whose numeric values are computed by formulas, unless otherwise specified.

3.1 Spreadsheet Programming Model

A spreadsheet can be modeled as a set of cells with expressions, which are indexed by two-dimensional *cell addresses* (a row index and a column index, e.g., B1 or C2) [5]. The expression of a data cell and a formula cell is given by its numeric value and formula, respectively. A formula references another cell by means of a *cell reference*, which denotes the referenced cell's address. Let R be the set of cell references, EXP be the set of expressions, and V be the set of plain values. A cell's expression exp is either a plain value ($v \in V$), a cell reference ($r \in R$), or a function (φ) applied to one or more expressions. Functions used in spreadsheets include basic operators (e.g., "+", "−", "*", "/") as well as other built-in functions from spreadsheet software (e.g., SUM, AVERAGE and MAX). Formally, a cell's expression exp is:

$$exp = v \mid r \mid \varphi(exp_1, \ldots, exp_n).$$

We define a reference-fetching function $\sigma(exp)$, which returns the set of cell references used in a cell's expression exp. Formally, $\sigma(exp)$ is:

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

Most spreadsheet systems have two built-in styles for representing a cell reference, namely, *A1* and *R1C1 representations* [52], and they can be either *absolute* or *relative*. An *absolute reference* points to a particular cell, and keeps pointing to this cell when it is copied to another cell. A *relative reference* presents the cell address offset between the current cell and the referenced cell, and the offset keeps unchanged when the reference is copied to another cell. In the A1 representation style, a cell at the $X$-th column and y-th row is notated as Xy in relative reference (e.g., B5), or $\$X\$y$ in absolute reference (e.g., \$B\$5). For example, the spreadsheet in Fig. 1(a) uses the A1 representation (all are relative references). On the other hand, in the R1C1 representation style, a cell at n rows below and m columns right to the current cell is notated as R[n]C[m] (in relative reference; [n] (or [m]) can be omitted when n = 0 or m = 0), and a cell at the n-th row and m-th column is notated as RnCm (in absolute reference). For example, the spreadsheet in Fig. 2 uses the R1C1 representation (all are relative references).

In subsequent discussions, we assume that expression exp and function $\sigma(exp)$ use the R1C1 representation, unless otherwise specified.

An interesting observation is that cell formulas prescribing the same formula patterns typically have semantically equivalent R1C1 representations. For example, the formula "B5 + C5" in cell D5 in Fig. 1(a) is "RC[-2] + RC[-1]" in the R1C1 representation, as shown in Fig. 2. 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. Fig. 2 also gives corresponding R1C1 representations for all formulas in the spreadsheet in Fig. 1(a). We can observe that some of them are semantically equivalent and some are similar with each other. We use such features to detect cell arrays and find their contained smells.

3.2 Cell Array

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., [A2:A7], [D2:D7], [E3:E7] and [B9:C9] in Fig. 1(b)) prescribing 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 as its *formula pattern* ($f_{pattern}$). In subsequent discussions, we assume that formula patterns always use the R1C1 representation for ease of presentation. 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:

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

The first condition states that any two cells' expressions in this cell array share the same cell references. The second condition states that any two cells' expressions should be evaluated to the same output value given the same input values to their cell references. For example, two expressions "2 * (R[-2]C + R[-1]C)" and "2 * R[-2]C + 2 * R[-1]C" are semantically equivalent although they are syntactically different. Our CACheck checks well-formedness using constraint solver Z3 [42]. Since a well-formed cell array has all its expressions semantically equivalent, we can take any of them as the cell array's $f_{pattern}$.

Note that the concept of cell array proposed in this article differs slightly from copy equivalence (i.e., two cells' expressions are identical) proposed by Clermont et al. [11], [12], [41]. In order to detect computational semantic smells, we require that: (1) any two cells' expressions in a cell array are evaluated to the same output value given the same input values (i.e., their expressions may be syntactically different but should be semantically equivalent; we adopt Z3 [42] to check expressions' equivalence), and (2) the cell array's contained cells are consecutive in layout. However, for cells that are copy equivalent [11], it is assumed that: (1) their expressions must be identical, and (2) they are not necessary to be topologically adjacent (e.g., they can be separated by some other cells with different expressions).

We can classify cell arrays based on either their orientation or the way they reference other cells in their formulas. Based on their orientation, cell arrays can be classified into row-based and column-based:

Row-based cell array. It comprises consecutive non-empty cells in a row. For example, [B9:C9] in Fig. 1(b) is a row-based cell array.

Column-based cell array. It comprises consecutive non-empty cells in a column. For example, [D2:D7] in Fig. 1(b) is a column-based cell array.

Based on how they reference other cells in formulas, cell arrays can be classified into homogeneous and inhomogeneous:

Homogeneous cell array. A row-/column-based cell array is *homogeneous* if the expression of each contained cell *c* reference only cells in the same column/row as that of *c*. For example, in the cell array [D2:D7] (column-based cell array) in Fig. 1(b), cell D2 references cells B2 and C2 (the same row as D2) as inputs, and cell D3 references cells B3 and C3 (the same row as D3) as inputs. Therefore, the cell array [D2:D7] is homogeneous. The cell array [B9:C9] in Fig. 1(b) is homogeneous, too.

Let *row* represent the row index of a cell *c* or a cell reference *cr*, and *col* represent the column index of a cell *c* or a cell reference *cr*. Formally, a cell array *CellArray* is homogeneous if the following condition holds:

For row-based *CellArray*,

$$\forall c \in CellArray, (\forall cr \in \sigma(c.exp), cr.col = c.col).$$

For column-based *CellArray*,

$$\forall c \in CellArray, (\forall cr \in \sigma(c.exp), cr.row = c.row).$$

Inhomogeneous cell array. A row-/column-based cell array is *inhomogeneous* if it contains a cell *c* whose expression references some cells in a column/row different from that of *c*. For example, in the cell array [A2:A7] (column-based cell array) of Fig. 1(b), cell A3 references cell A2 (different row from cell A3) as its input. Therefore, the cell array [A2:A7] is inhomogeneous. In cell array [E3:E7] (column-based cell array), cell E3 references cells D2 (at a different row from cell E3) and D3 (at the same row as cell E3) as its inputs. Therefore, the cell array [E3:E7] is inhomogeneous, too.

Formally, a cell array *CellArray* is inhomogeneous if the following condition holds:

For row-based *CellArray*,

$$\exists c \in CellArray, (\exists cr \in \sigma(c.exp), cr.col \neq c.col).$$

For column-based *CellArray*,

$$\exists c \in CellArray, (\exists cr \in \sigma(c.exp), cr.row \neq c.row).$$

3.3 Ambiguous Computation Smell

If a cell array is not well-formed, we say that it suffers from an *ambiguous computation smell* or it is *smelly*. 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 expressions. A cell array of

TABLE 1
Statistics of Our Study Subjects

Corpus	Subjects			Cell arrays				
	SS	Processed SS	SS with formulas	SS with CA	Initial CA	CA	SS with CA/ SS with formulas	Average CA per SS with CA
EUSES	4,037	3,737	1,617	1,118	26,393	21,427	69.1%	19
Enron	15,926	15,790	9,137	6,298	1,177,967	569,986	68.9%	91
Total	**19,963**	**19,527**	**10,754**	**7,416**	**1,204,360**	**591,413**	**69.0%**	**80**

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 formula pattern $f_{pattern}$:

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

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 cells A7 and E7 in Fig. 1(a).

4 Empirical Study on Well-Formed Cell Arrays

In this section, we report our findings from an empirical study on well-formed cell arrays in the EUSES [19] and Enron [26] corpora. We aim to understand the use of cell arrays in real-life spreadsheets. We focus on the following three research questions:

RQ1: *How commonly are cell arrays used in spreadsheets?*

RQ2: *How common are homogeneous and inhomogeneous cell arrays? Especially, are inhomogeneous cell arrays common?*

RQ3: *How are cell arrays structured in spreadsheets? Especially, does a cell array often occupy a whole range of consecutive cells? Do cell arrays overlap?*

To answer questions RQ1–3, we conducted an empirical study on the EUSES and Enron corpora. We extracted all well-formed cell arrays from these spreadsheets, and analyzed them statistically for answering these questions. We have also made our tool and empirical results available online for future research [62].

4.1 Subject Selection and Well-Formed Cell Array Extraction

Subject selection. The EUSES corpus consists of 4,037 spreadsheets from 11 categories. These spreadsheets were mainly collected over the web by search engines. Since its creation in 2005, the EUSES corpus has been widely used for spreadsheet research although the corpus may not necessarily represent spreadsheets used in companies. The EUSES corpus is the most cited one among all spreadsheet corpora so far. The Enron corpus is a recent collection that consists of 15,926 spreadsheets, which were extracted from the Enron Email Archive, within the Enron Corporation [38]. The Enron corpus is considered a collection that represents spreadsheets used in a typical enterprise.

In Table 1, columns 2-4 list the statistics of the EUSES and Enron corpora. There are 19,963 spreadsheets (SS) in total. We found that 97.8% (19,527/19,963) of spreadsheets in these two corpora could be parsed by the Apache POI [63] that we used to parse spreadsheets (Processed SS).

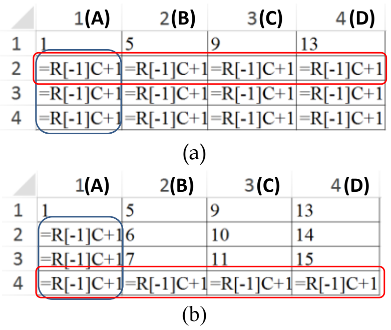


Fig. 3. Spreadsheets with overlapping cell arrays**.**

Some pre-1995 (BIFF5 format) spreadsheets cannot be processed by the Apache POI. We did not attempt to recover these spreadsheets as the recovery process may not be content preserving. Moreover, these spreadsheets may no longer well represent those being used nowadays. Among the spreadsheets that can be parsed, 55.1% (10,754/19,527) of them contain formula cells.

Extracting well-formed cell arrays. We examine consecutive formula cells clustered in a row or column, and consider a cell cluster to be a well-formed cell array if: (1) the cluster is not a subset of any other cell array, (2) each cell in the cluster has a semantically equivalent expression in the R1C1 representation style. The first condition enforces that a cell array's neighboring cells should not have the same formula pattern as this cell array. The second condition enforces that all the cells in a cell array should prescribe the same formula pattern.

Filtering well-formed cell arrays. Cell arrays in a compact region may prescribe the same computational semantics. For example, in Fig. 3(a), cells in the region [A2:D4] all have the same formula pattern. One can obtain a set of three row-based cell arrays [A2:D2], [A3:D3] and [A4:D4], and another set of four column-based cell arrays [A2:A4], [B2:B4], [C2:C4] and [D2:D4]. These two sets of cell arrays overlap. We consider the two sets are semantically equivalent because either of them can represent the computational semantics of the whole region [A2:D4]. Keeping one of them in cell array extraction is enough. The other one can be filtered out.

We use two criteria to filter out equivalent sets of cell arrays. First, the remaining sets contain all the cells of the ones that are filtered out. Second, the number of overlapping cells among the remaining ones is minimized. If there are more than one solution satisfying both criteria, we choose the solution that contains the least number of cell arrays. Based on these criteria, the set of three row-based cell arrays [A2:D2], [A3:D3] and [A4:D4] is selected in the cell array extraction of the spreadsheet in Fig. 3(a). The other set of four column-based cell arrays are filtered out.

TABLE 2
Statistics of Well-formed (Homogeneous and Inhomogeneous) Cell Arrays

Corpus	CA	Homogeneous	Inhomogeneous
EUSES	21,427	16,383 (76.5%)	5,044 (23.5%)
Enron	569,986	450,691 (79.1%)	119,295 (20.9%)
Total	**591,413**	**467,074 (79.0%)**	**124,339 (21.0%)**

TABLE 3
Layout Statistics of Well-formed Cell Arrays

Corpus	CA	Whole1	Whole2	Overlap
EUSES	21,427	12,612 (58.9%)	13.2%	110 (0.5%)
Enron	569,986	344,790 (60.5%)	17.4%	3,487 (0.6%)
Total	**591,413**	**357,402 (60.4%)**	**17.3%**	**3,597 (0.6%)**

For the spreadsheet in Fig. 3(b), the cell array extraction selects cell arrays [A2:A4] and [A4:D4].

Table 1 lists the numbers of cell arrays extracted from the two corpora. We in total extracted 1,204,360 cell arrays (Initial CA), and after filtering, we obtained 591,413 (CA) cell arrays, which occupy 49.1% (CA/Initial CA). In the following, we study these obtained remaining non-equivalent well-formed cell arrays.

4.2 RQ1: How Commonly are Cell Arrays Used in Spreadsheets?

Table 1 also gives how commonly cell arrays are used in real-life spreadsheets. Among 10,754 spreadsheets with formulas (SS with formulas), 7,416 spreadsheets (SS with CA) have used cell arrays. Interestingly, the percentage of spreadsheets that use cell arrays (SS with CA/SS with formulas) is almost the same in the EUSES and Enron corpora. On average, there are 80 (Average CA per SS with CA) cell arrays in a spreadsheet with cell arrays. The Enron corpus (91) uses cell arrays more often than the EUSES corpus (19), and the ratio is about 4.8 (91/19). This suggests that cell arrays are used more often in the Enron corpus. Therefore, we make the following observation:

Observation 1: *Cell arrays are commonly used in real-life spreadsheets. The Enron corpus uses cell arrays more often than the EUSES corpus.*

4.3 RQ2: How Common are Homogeneous and Inhomogeneous Cell Arrays?

In our earlier work AmCheck [17], we assume that each cell in a row-/column-based cell array references only the cells that share the same column/row as this cell. Therefore, AmCheck can only detect homogeneous cell arrays. Thus, we care about how commonly homogeneous and inhomogeneous cell arrays are used in real-life spreadsheets. Table 2 lists the statistics of all cell arrays (CA), homogeneous cell arrays (Homogeneous) and inhomogeneous cell arrays (Inhomogeneous). We observe that 79.0% cell arrays are homogeneous. This result suggests that AmCheck's assumption (i.e., most cell arrays are homogeneous) is reasonable, and AmCheck can detect most (79.0%) cell arrays. Still, we also observe that about 21.0% cell arrays are inhomogeneous. As a result, AmCheck would thus miss all these inhomogeneous cell arrays, whose percentage is not negligible. Therefore, we make the following observation:

Observation 2: *Inhomogeneous cell arrays are also common (21.0%) in real-life spreadsheets. One needs to extend AmCheck in order to detect such cell arrays.*

4.4 RQ3: How are Cell Arrays Structured in Spreadsheets?

Given a well-formed cell array, if neither of its two neighboring cells is a data cell or formula cell (i.e., they are empty or labels), we say that this cell array occupies a whole range of consecutive cells in a row or column. For example, in Fig. 1(b), cell arrays [D2:D7] and [E3:E7] both occupy a whole range of consecutive cells. However, cell array [B9:C9] does not occupy a whole range of consecutive cells, due to the existence of cell D9 (formula cell). Table 3 shows that only 60.4% well-formed cell arrays occupy a whole range of consecutive cells (Whole1). Interestingly, the EUSES and Enron corpora have almost the same ratio.

We further investigated whether a whole range of consecutive cells in a row or column often form a well-formed cell array. To carry out the investigation, we consider those ranges of consecutive numeric cells that contain at least one formula cell and are bound by non-numeric cells at both ends. We extracted 3,056,367 ranges of consecutive cells from the EUSES and Enron corpora. Among them, we found that only 17.3% share the same formula patterns (i.e., they are well-formed cell arrays), while 82.7% do not (Whole2). This leads us to make the following observation:

Observation 3: *Consecutive cells in the same row or column do not necessarily form a cell array.*

The observation suggests that one cannot simply aggregate consecutive cells in the same row or column to form a cell array. We need a more precise way to identify cell arrays.

We are also interested in finding whether cell arrays could possibly overlap with each other in spreadsheets. If two cell arrays have shared cells, we say that they *overlap*. For example, in Fig. 3 (a), two cell arrays [A2:A4] and [A2:D2] share cell A2; in Fig. 3 (b), two cell arrays [A2:A4] and [A4:D4] share cell A4. If two cell arrays overlap, they should have the same formula pattern. As a result, two row-/column-based cell arrays should not overlap. This is because if they overlap, they should be merged into a single cell array.

Cell arrays can overlap in two ways: redundant and non-redundant. As mentioned earlier in cell array extraction (Section 4.1), a cell array is redundant if all of its cells are contained by other cell arrays. Such overlapping is redundant and can be removed. For example in Fig. 3(a), all cells in range [A2:D4] share the same formula pattern, and the three row-based cell arrays ([A2:D2], [A3:D3] and [A4:D4]) and four column-based cell arrays ([A2:A4], [B2:B4], [C2:C4] and [D2:D4]) overlap. Therefore, we need only to extract the three row-based cell arrays. By doing so, our extracted cell arrays do not overlap.

Cell arrays can also overlap in a non-redundant way, which requires a different treatment. For example, in Fig. 3(b), cell arrays [A2:A4] and [A4:D4] overlap on cell A4, but neither of them can represent the other. Therefore, we need to keep both overlapping cell arrays during extraction. Table 3 lists the statistics of overlapping cell arrays in the EUSES and Enron corpora (Overlap). We observe that cell arrays rarely (0.6%) overlap in real-life spreadsheets. Therefore, we make the following observation:

Observation 4: *Cell arrays rarely overlap in real-life spreadsheets.*

4.5 Summary

The above observations enable us to effectively identify cell arrays in real-life spreadsheets, and support follow-up detection and repairing of smells in these cell arrays.

For example, observation 2 motivates us to identify both homogeneous and inhomogeneous cell arrays in spreadsheets. Observation 3 suggests the cell extraction technique should be capable to identify cell arrays that may not occupy whole rows or columns. Observation 4 can be leveraged to filter out wrongly identified cell arrays. The methodology elaborated in the next section embodies these ideas.

5 Detecting and Repairing Smelly Cell Arrays

After analyzing a given spreadsheet, CACheck reports all detected smelly cell arrays with repair suggestions. Fig. 4 shows its architecture. CACheck heuristically extracts cell arrays from a spreadsheet (Section 5.1), and detects whether each of them is smelly via constraint solving (Section 5.2). CACheck infers a cell array's formula pattern $f_{pattern}$ in two steps. First, CACheck uses values and formulas in a cell array to derive constraints associated with its intended formula pattern (Section 5.3). Second, CACheck infers the formula pattern $f_{pattern}$ based on these constraints. In order to expedite the inference process, CACheck combines heuristics (Section 5.4) and program synthesis techniques (Section 5.5). After the inference, CACheck identifies smelly cells in a cell array and their contained conformance errors, if any, based on its inferred $f_{pattern}$ (Section 5.6). Finally, CACheck identifies and removes false positives based on the observations from Section 4, as well as the inferred $f_{pattern}$ (Section 5.7).

5.1 Extracting Cell Arrays

The first challenge of smelly cell array detection is to identify cell arrays from a given spreadsheet, which has no record about which cells were previously prepared by copy-and-paste or auto-fill operations. We observe that a spreadsheet snippet usually provides useful hints about boundaries of cell arrays. Besides, the cells in a cell array often have similar formulas. Their formulas can be similar by means of referencing the same cells or referencing different cells with the same R1C1 representation. Such similarity facilitates our cell array identification and extraction.

We first identify spreadsheet snippets. Related data or formulas in a spreadsheet are often clustered together in a rectangle circumscribed by empty cells or labels [27]. We refer to such rectangles of cells as *snippets*. Examples of spreadsheet snippets in Fig. 1(a) include two rectangles comprising cells [A2:E7] and [B9:D9], respectively.

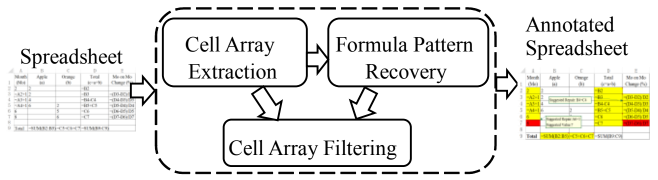


Fig. 4. CACheck's architecture.

To identify snippets, we adopt a cell classification strategy, similar to what Abraham and Erwig [4] proposed. 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 snippet.

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

We next extract cell arrays from the identified snippets. As observed earlier, the cells in a cell array are often similar in how their formulas reference other cells. We capture this similarity by means of dependence similarity.

Given a pair of cells, c_1 and c_2, from a consecutive range of numeric cells in a row or column (e.g., CellArray2 in Fig. 1(a)), if c_1 and c_2 satisfy one of the following four conditions, they are said to have *dependence similarity*:

Condition 1. Either c_1 or c_2 is a data cell. Since one has no idea on how the value in a data cell is computed, this data cell can potentially have any dependence on other cells. Therefore, we consider that a data cell has dependence similarity with any other cell. For example, in Fig. 1(a), cells A6 and A7 are both data cells, and thus they have dependence similarity.

Condition 2. Both c_1 and c_2 are formula cells, and they reference some cells in common. For example, cells E3 and E4 in Fig. 1(a) commonly reference cell D3, and thus they have dependence similarity.

Condition 3. Both c_1 and c_2 are formula cells and they do not reference any cell in common, but they reference some cells in the same way. For example, in Fig. 1(a), cells D3 and D4 reference cells B3 and B4, respectively (D4 also references C4, but it is not important here and so omitted). Although B3 and B4 are not the same cell, they are referenced in the same way (same distance to D3 and D4), and therefore their references are the same in the R1C1 representation, i.e., R[-2]C, as shown in Fig. 2. Thus, cells D3 and D4 also have dependence similarity.

Condition 4. Both c_1 and c_2 are formula cells and they do not satisfy condition 2 or 3, but there exists another cell c_3 from the same consecutive range, such that: (1) c_1 and c_3 satisfy condition 2 or 3, and (2) so do c_2 and c_3. For example, in Fig. 1(a), cells D2 and D7 do not satisfy either condition 2 or 3, but: (1) D2 and D4 satisfy condition 3, and (2) so do D7 and D4. Then cells D2 and D7 satisfy condition 4. As a result, they also have dependence similarity.

Overall, our cell array extraction algorithm works as follows. For each identified spreadsheet 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 already identified cell array, (2) each pair of cells in the cluster has dependence similarity, and (3) at least one cell in the cluster is a formula cell. The algorithm may have three outcomes:

- **Cell array:** A row- or column-based cell array is successfully identified and extracted. Two examples are [D2:D7] and [B9:C9] in Fig. 1(a).
- **Plain value:** A consecutive range of cells in a row or column are all data cells. One cannot tell whether they prescribe the same business concept and are subject to certain computational semantics. We do not consider it as a cell array.
- **Others:** It does not belong to the above two cases. To play safe, we also do not consider it as a cell array.

Note that the cell arrays we now extract can be either well-formed or smelly. Therefore, the above four conditions relax the ones we use to extract well-formed cell arrays earlier in Section 4.1. For example, the above condition 1 can help detect cell arrays that suffer from missing formula smells. Conditions 2, 3 and 4 can help detect cell arrays that suffer from inconsistent formula smells.

5.2 Detecting Smelly Cell Arrays

Next, let us explain how to check whether an extracted cell array is smelly or not.

According to our earlier Definition 1, a cell array is well-formed if: (1) it contains only formula cells, (2) all its expressions share the same cell references, and (3) all its expressions are semantically equivalent. If a cell array does not satisfy any of the above conditions, it suffers from ambiguous computation smells, and is thus *smelly*.

As such, we partition the extracted cell arrays into two groups: well-formed and smelly. In the following, we examine smelly cell arrays to repair their contained smells by recovering their intended formula patterns.

5.3 Extracting Formula Pattern Constraints

To detect and repair a smelly cell array, CACheck needs to recover its formula pattern $f_{pattern}$. To do so, we first extract constraints behind the formula pattern $f_{pattern}$.

Our idea was inspired by component-based program synthesis, which synthesizes a loop-free program from components, input-output pairs and specifications used by this program [22], [36]. The synthesis is based on three assumptions: (1) Existing expressions in a cell array are good hints for inferring its formula pattern $f_{pattern}$; (2) Most cell values should be correct for this cell array, and they can serve as $f_{pattern}$'s input-output pairs; (3) Components in expressions used by this cell array are often those used by this cell array's formula pattern $f_{pattern}$. Under these assumptions, CACheck recovers a smelly cell array's intended formula pattern by extracting its constraints from the cells of this cell array, and combining them appropriately. The extraction process consists of four parts, i.e., extracting input variables, functions, input-output pairs and components from a smelly cell array, as follows:

1) All cell references used by expressions in a cell array are considered as *input variables* for this cell array's $f_{pattern}$. For example, in Fig. 1 (a), input variables for CellArray1 are R[-1]C, input variables for CellArray2 are RC[-2] and RC[-1], input variables for CellArray3 are R[-1]C[-1] and RC[-1], and input variables for CellArray4 are R[-7]C, R[-6]C, …, R[-2]C. The process may extract irrelevant input variables, which could be removed later. Let IV be the set of a cell array's input variables, and x_i be the i-th input variable in IV. After extracting n input variables for $f_{pattern}$, we can model $f_{pattern}$ as a conceptual function like $f(x_1, x_2, \dots x_n)$. Formally, IV is defined as:
$$IV = \sigma(c_1.exp) \cup \sigma(c_2.exp) \dots \cup \sigma(c_m.exp),$$
$$\text{where } c_1, c_2, \dots c_m \in CellArray.$$
2) Existing expressions in the cell array are extracted as *functions*. For example, one can extract four functions from CellArray2, 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 considered as specifications in component-based program synthesis [22], [36].
3) All data (including those calculated by expressions) in the cell array are considered as *input-output pairs*. For example, in Fig. 1 (a), input-output pairs in CellArray2 include <(2, 0), 2>, <(4, 0), 4>, <(6, 0), 6>, <(0, 5), 5> and <(0, 6), 6>.
4) All operators and constants used by expressions in the cell array are considered as *components*. Note that we also consider a constant as a component that returns this constant value. For example, in Fig. 1 (a), components from CellArray1 include "+" and constant (1), components from CellArray2 include "+" and "–", components from CellArray3 include "–" and "/", and components from CellArray4 include "+" and SUM. Some components might be irrelevant, but could be removed later. If CACheck fails to find any operator from a cell array, it would add basic operators (e.g., +, –, *, /) as components.

All thus extracted input variables, functions, input-output pairs and components are *constraints* used for recovering or synthesizing a smelly cell array's intended formula pattern $f_{pattern}$, as explained in the following.

5.4 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 $f(x) = x + 1$ extracted from formula cells in CellArray1 in Fig. 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 extracted from its formula cells. This can significantly reduce the cost of formula pattern inference since program synthesis [22], [36] is typically 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 this function. For example, the value (6) of cell A6 in CellArray1 in Fig. 1(a) can be computed by $f(x) = x + 1$, where

x represents the cell reference R[-1]C. We say that a function *covers* a formula cell if the function is *compatible* with the one extracted from this cell in the sense that both can return the same outputs values given the same input values. For example, function $f(x_1, x_2) = x_1 + x_2$ is compatible with function $f(x_1, 0) = x_1$ extracted from cell D2 in Fig. 1(a). Note that the second parameter needs to bind to zero for both functions to take the same values as inputs. However, function $f(x_1, x_2) = x_1 + x_2$ is not compatible with function $f(x_1, x_2) = x_1 - x_2$ extracted from cell D4 in CellArray2 in Fig. 1(a). This is because their output values are different when x_1 and x_2 are set to 0 and 1, respectively.

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. If only one function can be extracted from a 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 data and formula cells in the cell array (Lines 4–10) is treated as $f_{pattern}$. The `Coverage` method (Lines 13–27) computes the ratio of cells a function can cover against all cells in the cell array. Lines 17-19 (or Lines 21-23) check whether a formula (or data) cell is covered by a function.

5.5 Synthesizing $f_{pattern}$

The $f_{pattern}$ recovery algorithm returns NULL when it fails to identify an appropriate $f_{pattern}$ for a smelly cell array from its extracted functions. When this happens, CACheck would try to synthesize $f_{pattern}$ using component-based program synthesis [22], [36].

Let us first review how component-based program synthesis works for constructing a program. Program synthesis first derives constraints (*constraints$_{ps}$*) for the target program to be synthesized based on a set of components and input-output pairs, which can be generated by specifications [22] or provided by users [36]. 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$_{ps}$*). Then more input-output pairs are used to provide additional constraints to further 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 $f_{pattern}$: (1) Component-based program synthesis requires users to explicitly provide components and input-output pairs. The algorithm addresses this using constraints extracted from cells in a smelly cell array (Section 5.3). (2) Functions extracted from a smelly cell array may not be compatible 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 Fig. 1(a) are not compatible. This can cause our $f_{pattern}$ synthesis to fail. (3) Data cells may contain incorrect values, which cannot be computed by the cell array's $f_{pattern}$ even if it is correct. Such incorrect values can also cause our $f_{pattern}$ synthesis to 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

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

```
Input: IV (input variables), FUNC (functions), IO (input-out-
       put 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
```

group are compatible. The method classifies as many distinct compatible functions into 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 added into any existing group, it creates a new compatible group (Lines 24–25) and iteratively adds in other functions compatible with this new group (Lines 26–30). Note that a function is allowed to be in multiple compatible groups. For example, we can obtain two compatible groups from CellArray2 in Fig. 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 $f_{pattern}$ synthesis, but one has 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 uses 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 [22] by treating functions as specification inputs. It generates a $f_{pattern}$ candidate set for each compatible group. If the functions in a group are not restrictive enough, the set can contain multiple candidates. In other words, all functions in the group collectively constitute only a partial specification for the $f_{pattern}$ synthesis. The algorithm then takes the second step to enrich the specification with additional constraints given by input-output pairs using the `Refine` method (Line 6). For each $f_{pattern}$ candidate set, the method iteratively prunes inappropriate candidates from the set using the input-output pairs from the given cell array while ignoring those that lead to no solution. This relieves us from the need for identifying incorrect

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

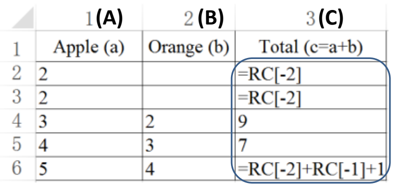
```
Input: IV (input variables), FUNC (functions), IO (input-out-
put pairs), COMP (components), CA (cell array).
Output: F (target formula pattern).
 1:  groups = Classify(FUNC); // Get compatible groups
 2:  pert = 0; F = NULL;
 3:  while groups not EMPTY do
 4:    group = groups.removeOne(); // Retrieve 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:  if (F = NULL) then // Synthesis fails
13:    foreach fn in FUNC do
14:      if (Coverage(fn, CA) > pert) then
15:        pert = Coverage(fn, CA);
16:        F = fn;
17:      end if
18:    end for
19:  end if
20:  return F;
21:
22:  method Classify(FUNC)
23:    groups = EMPTY;
24:    while (∃initFunc∈FUNC. initFunc non-classified) do
25:      newGroup = {initFunc};
26:      foreach func in FUNC\newGroup do
27:        if (!∃fn ∈ newGroup. ∃in. fn(in) ≠ func(in)) then
28:          newGroup.add(func);
29:        end if
30:      end for
31:      groups.add(newGroup); // All in newGroup classified
32:    end while
33:    return groups;
34:  end method
```

data cells and excluding their associated input-output pairs. Details of this pruning process can be found in related work [36]. The `Refine` method would return a $f_{pattern}$ candidate randomly if there are multiple remaining ones for each set as its result. Finally, among all returned $f_{pattern}$ candidates, Algorithm 2 selects the one that covers the most cells in the given cell array as its synthesized $f_{pattern}$ (Lines 7–10).

We note that program synthesis relies heavily on its underlying constraint solver. As in practice, we also set up a timeout limit for solving constraints. The limit, say 5 minutes, is set with respective to each compatible group. Upon timeout, we conservatively select one function, which currently covers the most cells in the given cell array, as its $f_{pattern}$ (Lines 12–19).

For the four smelly cell arrays in Fig. 1(a), their $f_{pattern}$ can be recovered by Algorithm 1 or synthesized in the first step of Algorithm 2. So we consider a more complicated example, cell array [C2:C6], as shown in Fig. 5. From this cell array, one can extract two functions ($f(x_1, 0) = x_1$ and $f(x_1, x_2) = x_1 + x_2 + 1$), four input-output pairs (IO = {<(2, 0), 2>, <(3, 2), 9>, <(4, 3), 7>, <(5, 4), 10>}), and three components (two "+" operators and one constant (1)).

When Algorithm 2 starts, its `Classify` method (Line 1) partitions the above two functions into two different compatible groups: (1) $f(x_1, 0) = x_1$ and (2) $f(x_1, x_2) = x_1 + x_2 + 1$. We use the first compatible group to explain our two-step synthesis, and show this process in Table 4. During synthesis, we need to prune inappropriate $f_{pattern}$ candidates by iteratively adding input-output pairs, which are generated from the first compatible group (`SynFPattern`) or selected from IO (`Refine`). In the first iteration, `SynFPattern` uses an input-output pair <(1, 0), 1> to generate its



	1(A)	2(B)	3(C)
1	Apple (a)	Orange (b)	Total (c=a+b)
2	2		=RC[-2]
3	2		=RC[-2]
4	3	2	9
5	4	3	7
6	5	4	=RC[-2]+RC[-1]+1

Fig. 5. A more complicated smelly cell array for synthesis, in which cells [C2:C6] should uniformly follow a formula pattern of $C_i = A_i + B_i$ ($2 \le i \le 6$).

TABLE 4
Two-step synthesis process

Iteration	IO pair	Candidates
1	<(1, 0), 1>	$f(x_1, x_2)=x_1$; $f(x_1, x_2)=x_1+x_2$; $f(x_1, x_2)= x_2+1$; $f(x_1, x_2)=1$; $f(x_1, x_2)=x_1+x_2+x_2$; $f(x_1, x_2)=x_2+x_2+1$
2	<(2, 0), 2>	$f(x_1, x_2)=x_1$; $f(x_1, x_2)=x_1+x_2$; $f(x_1, x_2)=x_1+x_2+x_2$
3	<(3, 2), 9>	none
4	<(4, 3), 7>	$f(x_1, x_2)=x_1+x_2$

initial set of $f_{pattern}$ candidates. This input-output pair is generated from function $f(x_1, 0) = x_1$ in the compatible group. Then `SynFPattern` generates six $f_{pattern}$ candidates: $f(x_1, x_2) = x_1$, $f(x_1, x_2) = x_1 + x_2$, $f(x_1, x_2) = x_2 + 1$, $f(x_1, x_2) = 1$, $f(x_1, x_2) = x_1 + x_2 + x_2$ and $f(x_1, x_2) = x_2 + x_2 + 1$. Note that if multiple $f_{pattern}$ candidates are equivalent, only one would be generated, e.g., $f(x_1, x_2) = x_1 + x_2$ and $f(x_1, x_2) = x_2 + x_1$ are equivalent and thus only the former remains. Then in the second iteration, in order to prune some $f_{pattern}$ candidates, `SynFPattern` uses another input-output pair <(2, 0), 2>, which is also generated from function $f(x_1, 0) = x_1$ itself. This time `SynFPattern` generates only three $f_{pattern}$ candidates: $f(x_1, x_2) = x_1$, $f(x_1, x_2) = x_1 + x_2$ and $f(x_1, x_2) = x_1 + x_2 + x_2$ (the other three are pruned). Note that now `SynFPattern` can no longer further prune $f_{pattern}$ candidates by only using input-output pairs generated from function $f(x_1, 0) = x_1$. Therefore, Algorithm 2 moves on to the `Refine` method (Line 6). `Refine` would use input-output pairs from IO to further prune $f_{pattern}$ candidates. Since the input-output pair <(2, 0), 2> from IO has been used, in the third iteration, `Refine` uses the second input-output pair <(3, 2), 9> from IO. Unfortunately, none of the remaining three $f_{pattern}$ candidates can satisfy this pair, and thus this pair is ignored (in fact, this pair is a wrong input-output pair). In the fourth iteration, `Refine` uses the third input-output pair <(4, 3), 7> from IO. This time `Refine` generates a unique $f_{pattern}$ candidate: $f(x_1, x_2) = x_1 + x_2$ (the other two are pruned). Now we finish the synthesis process and return $f(x_1, x_2) = x_1 + x_2$ as the formula pattern for the first compatible group.

5.6 Identifying Smelly Cells

CACheck infers a smelly cell array's formula pattern $f_{pattern}$ successfully if it can recover or synthesize $f_{pattern}$ for this cell array. When successful, CACheck uses the inferred $f_{pattern}$ to check whether some cells contained in the cell array are smelly, and repair them if necessary.

We consider a cell in a cell array *smelly* if it is a data cell (i.e., missing formula smell), or it is a formula cell but its expression is not semantically equivalent to the inferred $f_{pattern}$ (i.e., inconsistent formula smell). Then, according to our earlier Definition 2, CACheck can further check whether a smelly cell suffers from a conformance error.

In identifying smelly cells, we notice one subtle but important issue that should be handled specially. In some cell arrays, a cell's value computation depends on another, which further depends on its next, forming a continual chain with all concerned cells in the same cell array. We call such cell arrays as *chained cell arrays*. For a chained cell array, its first several cells in the chain, whose computation does not depend on any other cells, contain initial (plain) values for the whole cell array. We should remove them from consideration of candidates for smelly cells. For example, in Fig. 1(b), each cell (except A2) in cell array [A2:A7] references its immediately above cell in its value computation, and thus this cell array is a chained one. The first cell A2 offers the initial value (2) for the whole cell array. Therefore, cell A2 should not be considered smelly.

5.7 Filtering Cell Arrays

As discussed earlier in Section 5.1, we use relaxed conditions to extract cell arrays, which can be either well-formed or smelly. However, due to this relaxation, our cell array extraction might incur *false positives*, i.e., a consecutive range of cells is mistakenly extracted as a cell array but it is not actually. For example, in Fig. 1 (a), cells [A3:B3] form a consecutive range, and they coincidentally satisfy condition 1 in our extraction. As a result, cells [A3:B3] are extracted as a cell array. However, we consider this cell array a false positive. This is because the two cells do not prescribe the same computational semantics (cell A3 is computed from cell A2, meaning a previous month, while cell B3 represents the amount of apple harvest in March). Similar cases also occur in cells [A4:B4] and [A5:B5]. As such, one needs to filter out such false positives in cell array extraction.

We note that our filtering is based on the earlier observations we made in the empirical study in Section 4, as well as the formula patterns, which are either recovered or synthesized, as discussed in Section 5.4 or 5.5. Therefore, we discuss our cell array filtering after them. We will use four examples of extracted cell arrays for illustration in the following discussions. They are [A2:A7], [A3:B3], [A4:B4] and [A5:B5] from Fig. 1 (a), among which we aim to identify the latter three as false positives.

Let CA be the set of extracted cell arrays. Our cell array filtering aims to select a subset of CA to satisfy certain constraints. Let CA_{select} be the subset ($CA_{select} \subseteq CA$). We collect constraints on CA_{select} , and generate all possible CA_{select} candidates to find one that satisfies these constraints. This process needs to consider the following two requirements:

1) From observation 4 in Section 4, cell arrays rarely overlap (only 0.6% cases). This suggests that, given a pair of extracted cell arrays that overlap (i.e., some of their cells are shared), one of them is probably a false positive. Therefore, we require that all cell arrays in CA_{select} should not overlap. In order not to mistakenly miss cell arrays, CA_{select} should also be maximized with respect to CA. This means that any cell array in $CA - CA_{select}$ must overlap with at least one cell array in CA_{select}. Considering the four extracted cell arrays [A2:A7], [A3:B3], [A4:B4] and [A5:B5], we have two CA_{select} candidates: CA_{select} = {[A2:A7]} or CA_{select} = {[A3:B3], [A4:B4], [A5:B5]}.

2) The first requirement tries to isolate true positives from false positives, but one has no idea which CA_{select} candidate best suits the characteristics of *true positives* (i.e., real cell arrays). We observe that even if a true positive suffers from ambiguous computation smells, its recovered or synthesized formula pattern can cover most of its contained cells. This means that a majority of its cells have correct values, and its contained conformance errors, if any, would be few. For example, in cell array [A2:A7] (true positive), all its cells except one (A7) have correct values. However, cells in a wrongly extracted cell array (false positive) cannot easily be covered by its recovered or synthesized formula pattern since its contained cells are put together in an unreasonable way. Thus one can easily detect conformance errors in a false positive. For example, in cell array [A3:B3] (false positive), its recovered formula pattern is R[-1]C + 1. Then cell B3 contains a conformance error (its value is 2 rather than 3). Similarly, one can also detect one conformance error (at cell B4) for cell array [A4:B4] (false positive) and one conformance error (at cell B5) for cell array [A5:B5] (false positive). As such, taking each CA_{select} candidate as a unit, one can detect one conformance error in the first candidate CA_{select} = {[A2:A7]}, but three conformance errors in the second candidate CA_{select} = {[A3:B3], [A4:B4], [A5:B5]}. Therefore, we should select a CA_{select} candidate, which has the minimal number of conformance errors, to win the best chance of isolating false positives from our consideration.

The above filtering process works on observations and heuristics, but it is highly effective in isolating true positives from false positives. For example, our later experiments reported that the overall removal precision is as high as 97.2%. In the following, we elaborate on the details of this filtering process.

5.7.1 Generating CA_{select} Candidates

According to the first requirement, the cell arrays in each CA_{select} candidate must not overlap. Each CA_{select} candidate is also a maximal subset with respect to CA in the sense that any cell array in $CA - CA_{select}$ must overlap with at least one cell array in CA_{select}. We use the following two constraints to generate such CA_{select} candidates.

Constraint 1: Given any CA_{select} candidate, for any two cell arrays in CA that overlap with each other, at most one of them can be in this CA_{select} candidate. For example, cell arrays [A2:A7] and [A3:B3] overlap. Then they must go to two different CA_{select} candidates. If two cell arrays ca_i and ca_j overlap, we denote this overlapping relationship as $overlap(ca_i, ca_j)$. Then, this constraint can be specified formally as (CA_{select} represents any CA_{select} candidate):

$$\bigwedge_{ca_i, ca_j \in CA} (ca_i \neq ca_j \land overlap(ca_i, ca_j)) \Rightarrow (ca_i \notin CA_{select} \lor ca_j \notin CA_{select}).$$

Constraint 2: Given any CA_{select} candidate, for any cell array ca in $CA - CA_{select}$, there exists at least one cell array in this CA_{select} candidate that overlaps with ca. This constraint makes sure that any cell array in $CA - CA_{select}$ cannot be added into this CA_{select} candidate (i.e., already maximized). This constraint can be specified formally as (CA_{select} represents any CA_{select} candidate):

$$\bigwedge_{ca\in(CA-CA_{select})} \exists ca' \in CA_{select}, overlap(ca, ca').$$

Any CA_{select} candidate must satisfy both constraints 1 and 2. Consider our earlier example: CA = {[A2:A7], [A3:B3], [A4:B4], [A5:B5]}. Its two CA_{select} candidates are thus: CA_{select} = {[A2:A7]} or CA_{select} = {[A3:B3], [A4:B4], [A5:B5]}.

In order to obtain all CA_{select} candidates, a straightforward way is to generate them by enumerating all subsets of CA, and check whether they satisfy both constraints 1 and 2. However, it would be exponentially complex. We choose to speed up this process based on the overlapping relationship. The key idea is to remove those subsets that do not satisfy constraint 1 or 2 as early as possible.

Our CA_{select} candidate generation algorithm (Algorithm 3) takes all extracted cell arrays (i.e., CA) as input, and returns all CA_{select} candidates. Its kernel part is the `Generate` method (Lines 8–27), which first selects those cells arrays that do not overlap with any other cell array into a candidate subset ($curCAs$) (Lines 9–13). If this step already selects all available cell arrays, we successfully find one CA_{select} candidate and add it into the result (Lines 14-18). Otherwise, we still have some remaining cell arrays not selected (in $restCAs$), and we know that they overlap with at least one cell array also not selected yet (in $restCAs$), but never overlap with any other cell array in $curCAs$. Then we consider selecting these remaining cell arrays in an iterative and recursive way. We consider each remaining cell array *ca* in turn as follows (Lines 20-26). (1) We add *ca* into $curCAs$ and at the same time remove those cell arrays overlapping with *ca* from $restCAs$ (Lines 22-23). By doing so, we again are able to possibly select more non-overlapping cell arrays from $restCAs$ into $curCAs$ (Lines 9-13). If this selects all remaining cell arrays, we find a new CA_{select} candidate (Lines 14-18). Otherwise, we consider the new $restCAs$ and $curCAs$ at a finer granularity and restart the `Generate` method recursively (Line 24). (2) When we complete considering the current *ca*, we restore the original $restCAs$ and $curCAs$ (Line 25), and then consider the next *ca* until we complete considering all remaining cell arrays in $restCAs$. In the above steps, we make two intended efforts. (1) We keep adding non-overlapping cell arrays into $curCAs$. This is for avoiding generating those subsets that are not maximal. (2) Whenever we add a cell array into $curCAs$, we also remove its overlapping cell arrays from $restCAs$. This is for avoiding generating those subsets that contain overlapping cell arrays.

Consider our earlier example: CA = {[A2:A7], [A3:B3], [A4:B4], [A5:B5]}. Since each cell array in CA overlaps with at least another cell array, we do not select any cell array into $curCAs$ in the first step (Lines 9-13). Then in the first iteration (Lines 20-26), we select cell array [A2:A7] into $curCAs$, and at the same time remove cell arrays [A3:B3], [A4:B4] and [A5:B5] from $restCAs$, since they all overlap with [A2:A7]. Thus, we generate the first candidate CA_{select} = {[A2:A7]} (this iteration completes quickly as $restCAs$ is empty). In the second iteration, we restart this process by first recovering original $restCAs$ (containing four cell arrays) and $curCAs$ (empty). This time, we select another cell array [A3:B3] into $curCAs$, and at the same time remove cell array [A2:A7] from $restCAs$, since it overlaps with [A3:B3]. Now, $restCAs$ becomes {[A4:B4], [A5:B5]}. During recursively invoking the `Generate` method, we select the two non-overlapping cell arrays ([A4:B4] and [A5:B5]) into $curCAs$ (Lines 9-13). Since now $restCAs$ is empty, we generate the second candidate CA_{select} = {[A3:B3], [A4:B4], [A5:B5]}.

Algorithm 3. CA_{select} candidate generation algorithm.

```
Input: CAs (inputted cell arrays).
Output: candidates (all $CA_{select}$ candidates).
 1: candidates = EMPTY;
 2: curCAs = EMPTY; // The current candidate
 3: Generate(CAs, curCAs);
 4: return candidates;
 5:
 6: // Any two cell arrays from restCAs and curCAs,
 7: // respectively, do not overlap
 8: method Generate(restCAs, curCAs)
 9:   foreach ca in restCAs do
10:     if (GetOverlap(ca, restCAs) = EMPTY) then
11:       restCAs.remove(ca); curCAs.add(ca);
12:     end if
13:   end for
14:   if (restCAs = EMPTY) then
15:     // Duplicated candidates are ignored
16:     candidates.add(curCAs);
17:     return
18:   end if
19:   // All cell arrays in restCAs overlap with others
20:   foreach ca in restCAs do
21:     tmpRestCAs = restCAs; tmpCurCAs = curCAs; // Backup
22:     restCAs.remove(ca); curCAs.add(ca); // Select ca
23:     restCAs.remove(GetOverlap(ca, restCAs));
24:     Generate(restCAs, curCAs);
25:     restCAs = tmpRestCAs; curCAs = tmpCurCAs; // Restore
26:   end for
27: end method
28:
29: // Get cell arrays in restCAs, which overlap with ca
30: method GetOverlap(ca, restCAs)
31:   overlapCAs = EMPTY;
32:   foreach tmp_ca in restCAs do
33:     if (ca ≠ tmp_ca && overlap(ca, tmp_ca)) then
34:       overlapCAs.add(tmp_ca); // ca overlaps with tmp_ca
35:     end if
36:   end for
37:   return overlapCAs;
38: end method
```

We note that Algorithm 3 may generate duplicated CA_{select} candidates. For example, when we go ahead with the third iteration, a duplicated candidate CA_{select} = {[A4:B4], [A3:B3], [A5:B5]} is generated. Since the ordering does not matter for set elements, this candidate is the same as the second one. Algorithm 3 would keep only one copy for duplicated CA_{select} candidates (Lines 15-16). For this example, the algorithm would eventually generate two CA_{select} candidates.

5.7.2 *Selecting CA_{select} Candidates*

With generated CA_{select} candidates, we use the following strategy for final selection.

As mentioned earlier, we aim to minimize the number of conformance errors in CA_{select} candidates, to win the best chance of isolating false positives from our consideration. Let *ca.errors* be the number of conformance errors in a cell array *ca*, and $error(CA_{select})$ be the number of conformance errors in all cell arrays in a CA_{select} candidate. Formally, $error(CA_{select})$ is defined as:

$$error(CA_{select}) = \sum_{ca \in CA_{select}} ca.errors.$$

We select as the final result the CA_{select} candidate that has the minimal $error(CA_{select})$ value. If there are multiple choices, we select the one having the least number of cell arrays. Then the set of cell arrays in this final CA_{select} candidate is our filtering result. Smell detection results are adjusted accordingly with this set (e.g., dropped if the concerned cell arrays are not in this set).

6 Evaluation

We implemented our smelly cell array detection approach as a tool named CACheck. CACheck builds on the Apache POI library [63] to manipulate spreadsheets in Excel files. CACheck loads an Excel file, analyzes its cell arrays, and generates comments explaining whether they contain ambiguous computation smells and what they are, as well as corresponding repairs suggested.

We implemented CACheck in Java 7 and used Z3 [42] as its underlying constraint solver. To be user-friendly, CACheck transforms an inferred formula pattern $f_{pattern}$ back to its A1 representation, e.g., RC[-2] + RC[-1] is transformed to B2 + C2 for cell D2 in Fig. 1(a). For visualization, CACheck marks its detection 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 suggesting their corresponding repairs; (3) Conformance errors are colored in red with comments explaining their reasons. These annotations can assist end users to quickly validate the reported problems. Fig. 6 gives a screenshot of CACheck's detection reports regarding problems identified for our motivating example in Fig. 1(a).

We then evaluate CACheck and study the following research questions (RQ1–3 were studied earlier in Section 4):

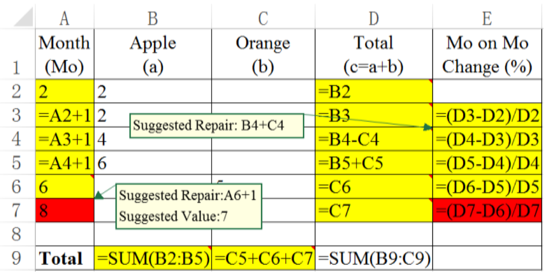
RQ4 (Precision): *Can CACheck detect and repair smelly cell arrays precisely?*

RQ5 (Recall): *Can CACheck detect smelly cell arrays with a high recall rate?*

RQ6 (Comparison): *How is CACheck compared with existing techniques, e.g., AmCheck, Excel, UCheck/Dimension and CUSTODES?*

RQ7 (Consistency): *Can CACheck obtain consistent results on different spreadsheet corpora, such as EUSES and Enron?*

To answer question RQ4, we ran CACheck on all spreadsheets in the EUSES corpus (Section 6.1), and manually validated all detected smelly cell arrays (Section 6.2). To answer question RQ5, we randomly sampled 50 spreadsheets from the EUSES corpus, and manually created the ground truths for them (i.e., manually identifying well-formed and smelly cell arrays). The process of subject selection is explained in Section 6.5.1. We then measured CACheck's recall rate on these 50 spreadsheets (Section 6.5).



	A	B	C	D	E
1	Month (Mo)	Apple (a)	Orange (b)	Total (c=a+b)	Mo on Mo Change (%)
2	2	2		=B2	
3	=A2+1	2		=B3	=(D3-D2)/D2
4	=A3+1	4		=B4-C4	=(D4-D3)/D3
5	=A4+1	6		=B5+C5	=(D5-D4)/D4
6	6		5	=C6	=(D6-D5)/D5
7	8			=C7	=(D7-D6)/D7
8					
9	**Total**	=SUM(B2:B5)	=C5+C6+C7	=SUM(B9:C9)	

Suggested Repair: B4+C4

Suggested Repair:A6+1
Suggested Value:7

Fig. 6. CACheck's screenshot for the spreadsheet in Fig. 1(a).

To answer question RQ6, we ran both CACheck and AmCheck on the EUSES corpus, and manually validated all detected smelly cell arrays to compare their performance, i.e., precision (Section 6.3) and recall rate (Section 6.5). We further compared CACheck with Excel, UCheck/Dimension and CUSTODES in Section 6.4. To answer question RQ7, we additionally ran CACheck on all spreadsheets in the Enron corpus, and manually validated 700 randomly sampled smelly cell arrays. We then compared statistical characteristics of the detected results on the EUSES and Enron corpora. We further sampled 50 spreadsheets from the Enron corpus and created their ground truths. Based on them, we measured CACheck's recall rate, and compared it to that for EUSES spreadsheets (Section 6.6). We have also made our tool, experimental dataset and results available online for future research [62].

6.1 Cell Array Detection for the EUSES Corpus

We ran CACheck on all spreadsheets in the EUSES corpus to detect smelly cell arrays.

Table 5 gives the statistics of cell arrays detected for each category of spreadsheets in the EUSES corpus (Category). It shows the statistics of cell arrays (Cell array) and smelly cell arrays (Smelly cell array). It also lists the number of cell arrays (CA), number of smelly cell arrays (SCA), and number of cell arrays suffering from missing formula smells (MISS), inconsistent formula smells (INCO) and both smells (Both). We observe that smelly cell arrays occur commonly in the EUSES corpus: 15.5% (3,443/22,177) of the detected cell arrays suffer from ambiguous computation smells. Among these smelly cell arrays, 53.7% (1,849/3,443) suffer from missing formula smells, 49.3% (1,699/3,443) suffer from inconsistent formula smells, and 3.0% (105/3,443) suffer from both smells.

Table 5 also gives the statistics of homogeneous and inhomogeneous cell arrays detected in the EUSES corpus. It shows the number of homogeneous cell arrays (Homo), number of inhomogeneous cell arrays (Inho), number of smelly homogeneous cell arrays (SHomo), and number of smelly inhomogeneous cell arrays (SInho).

We observe that 76.3% (16,928/22,177) of the detected cell arrays are homogeneous cell arrays. Out of them, 13.7% (2,324/16,928) suffer from ambiguous computation smells. We also observe that 23.7% (5,249/22,177) of the detected cell arrays are inhomogeneous cell arrays. Out of them, 21.3% (1,119/5,249) suffer from ambiguous computation smells. It seems that inhomogeneous cell arrays are more error-prone.

TABLE 5
Detected Cell Arrays in the EUSES corpus (n.a.: not applicable)

Category	Cell array				Smelly cell array						
	CA	Homo	Inho	Inho/CA	SCA	SHomo	SInho	SCA/CA	MISS	INCO	Both
cs101	39	35	4	10.3%	12	12	0	30.8%	8	4	0
database	3,271	2,302	969	29.6%	448	358	90	13.7%	345	114	11
filby	0	0	0	n.a.	0	0	0	n.a.	0	0	0
financial	7,008	5,573	1,435	20.5%	1,259	869	390	18.0%	502	796	39
forms3	150	114	36	24.0%	16	5	11	10.7%	14	2	0
grades	2,955	2,275	680	23.0%	666	528	138	22.5%	335	354	23
homework	2,702	1,971	731	27.1%	343	137	206	12.7%	214	140	11
inventory	3,903	3,133	770	19.7%	517	287	230	13.2%	322	213	18
jackson	0	0	0	n.a.	0	0	0	n.a.	0	0	0
modeling	2,018	1,394	624	30.9%	182	128	54	9.0%	109	76	3
personal	131	131	0	0.0%	0	0	0	0.0%	0	0	0
Total	**22,177**	**16,928**	**5,249**	**23.7%**	**3,443**	**2,324**	**1,119**	**15.5%**	**1,849**	**1,699**	**105**

Therefore, we draw the following conclusion:

Smelly cell arrays commonly exist in real-life spreadsheets, e.g., in the EUSES corpus. Ambiguous computation smells are also common for inhomogeneous cell arrays, which thus deserve detection.

6.2 CACheck's Precision on Smelly Cell Array Detection for the EUSES corpus

We then investigate CACheck's precision on its smelly cell array detection.

6.2.1 Smelly Cell Arrays

We first partition CACheck's detected smelly cell arrays into seven categories according to how many cells their inferred $f_{pattern}$ can cover in these arrays. The seven categories are: 100%, [90%, 100%), [80%, 90%), [70%, 80%), [60%, 70%), [50%, 60%) and [0%, 50%), which represent different levels of coverage. We then measure CACheck's precision on smell detection for the seven categories of cell arrays. For this purpose, we manually validated all smelly cell arrays detected in the EUSES corpus. For each category, Table 6 lists the number of smelly cell arrays (SCA), number of missing formula smells (M-SCA), and number of inconsistent formula smells (I-SCA). These numbers are also accompanied with corresponding numbers of validated-as-true smelly cell arrays (TP).

We observe that CACheck's inferred $f_{pattern}$ is able to cover all the cells in 1,184 smelly cell arrays (i.e., coverage of 100%), and 90% or more (but not 100%) cells in another 152 smelly cell arrays (i.e., coverage in range [90%, 100%)). This suggests that values and formulas in these 1,336 cell arrays are highly compatible with the inferred $f_{pattern}$. In other words, each of these 1,336 cell arrays that suffer from missing formula or inconsistent formula smells very likely prescribes common computational semantics expressed by the inferred $f_{pattern}$. Then these detected ambiguous computation smells in these cell arrays (1,336/3,443 = 38.8%) are probably true. This provides an alternative for assessing the quality of CACheck's smell detection results. We can use these seven categories to rank the likeliness of a smelly cell array being true (higher coverage, more probably true).

True positives. Out of the 3,443 smelly cell arrays detected, we manually validated them and found that 1,586 (46.1%) of them are true. These 1,586 smelly cell arrays cover 7.8% (1,586/(22,177 – (3,443 – 1,586))) of all identified well-formed and true smelly cell arrays. The precision for missing formula cell arrays (970/1,849 = 52.5%) is higher than that for inconsistent formula cell arrays (660/1,699 = 38.8%). We also observe that homogeneous cell arrays (Homo-SCA; 1,235/2,324 = 53.1%) have a higher precision than inhomogeneous cell arrays (Inho-SCA; 351/1,119 = 31.4%). Fig. 7 shows how the precision changes with different levels of coverage for the detected smelly cell arrays (CACheck), smelly homogeneous cell arrays (CACheck-Homo) and smelly inhomogeneous cell arrays (CACheck-Inho). We observe that the precision roughly decreases with the reduction in coverage. We also observe that there is a sharp decrease when the coverage is lower than 70%. Therefore, we recommend the level of coverage of 70% as a reliable threshold for smelly cell array detection, where the precision is 86.8% (1,386/1,597).

False positives. In Table 6, the values in the SCA/TP column also disclose false positives in smelly cell array detection. We analyzed the causes for the 211 (= 1,597 – 1,386) false positives for the coverage in range [70%, 100%]. There are three main causes: (1) Some spreadsheets use numbers as labels. For example, in financial reports, end users often use years like 2013 and 2014 as labels, which are, however, represented in a number format. Our heuristics in cell array extraction can misinterpret them as data cells. 6.2% (13/211) false positives belong to this case. It should be easy for end users to quickly validate such false positives. (2) Some cells in a row or column have the same computational semantics, but they are separated by empty cells. CACheck thus extracted multiple column- or row-based cell arrays, which should not be separated. 12.3% (26/211) false positives belong to this case. (3) For the remaining 81.5% (172/211) false positives, the concerned cells in these ranges contain complex computational semantics, which CACheck could not effectively recognize or distinguish currently. End users should manually confirm or reject them for such cases.

6.2.2 Smelly Cells

As mentioned earlier, some cells in a smelly cell array are smelly. They suffer from either missing formula smells or inconsistent formula smells. Further, a smelly cell may contain a conformance error if its value does not conform to that computed by the concerned cell array's inferred $f_{pattern}$.

TABLE 6
Detected and Validated Smelly Cell Arrays with Different Levels of Coverage in the EUSES Corpus

Coverage	Smelly cell arrays						Smelly cells		
	SCA/ TP	M-SCA/ TP	I-SCA/ TP	Homo-SCA/ TP	Inho-SCA/ TP	Fail to repair	M-Cell/ TP	I-Cell/ TP	CE/ TP
100%	1,184/ 1,092	679/ 661	525/ 449	877/ 831	307/ 261	9	2,477/ 2,445	2,478/ 2,251	0/ 0
[90%, 100%)	152/ 112	118/ 85	40/ 33	101/ 85	51/ 27	0	1,259/ 1,186	84/ 77	226/ 131
[80%, 90%)	164/ 117	106/ 78	66/ 45	112/ 86	52/ 31	3	804/ 724	218/ 140	398/ 323
[70%, 80%)	97/ 65	59/ 41	49/ 28	64/ 48	33/ 17	5	343/ 315	144/ 88	262/ 201
[60%, 70%)	406/ 74	141/ 31	272/ 47	320/ 65	86/ 9	3	396/ 209	470/ 175	749/ 294
[50%, 60%)	1,042/ 76	440/ 32	607/ 44	619/ 75	423/ 1	5	659/ 133	890/ 128	1,496/ 218
[0%, 50%)	398/ 50	306/ 42	140/ 14	231/ 45	167/ 5	21	1,786/ 202	679/ 66	2,422/ 291
Total	**3,443/ 1,586**	**1,849/ 970**	**1,699/ 660**	**2,324/ 1,235**	**1,119/ 351**	**46**	**7,724/ 5,214**	**4,963/ 2,925**	**5,553/ 1,458**

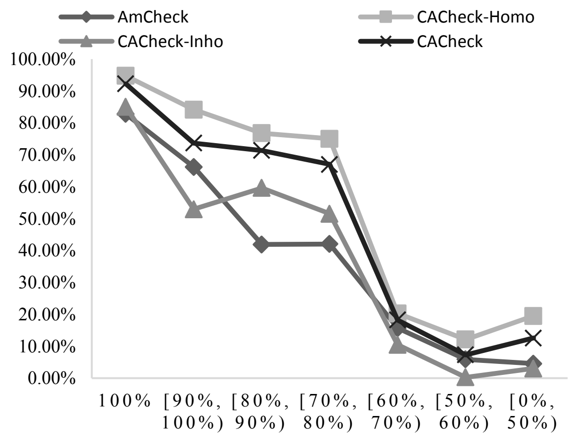


Fig. 7. Precision comparison on smelly cell array detection for different levels of coverage on the EUSES corpus (horizontal axis: coverage category, vertical axis: detection precision).

Table 6 shows the number of detected missing formula cells (M-Cell) and number of detected inconsistent formula cells (I-Cell). These numbers are also accompanied with corresponding numbers of validated-as-true smelly cells (TP) for comparison. We confirmed that a total of 8,139 (5,214 + 2,925) cells are truly smelly. Out of them, 5,214 cells suffer from missing formula smells, and 2,925 cells suffer from inconsistent formula smells. The precision for smelly cell detection is 64.2% (8,139/(7,724 + 4,963)). These 8,139 true smelly cells cover 25.9% of all (31,457) cells from 1,586 true smelly cell arrays, and 3.3% of all (250,245) cells from all (20320 = 22,177 – (3,443 – 1,586)) detected well-formed and true smelly cell arrays.

Besides, as shown in Table 6, CACheck detected a total of 5,553 conformance errors (CE) in the EUSES corpus. We manually validated them, and confirmed that 26.3% (1,458/5,553) detected conformance errors are true ones. We observe that conformance errors occurring in cell arrays with higher levels of coverage also have a higher probability to be true. For example, 73.9% (655/886) detected conformance errors are true in cell arrays with a level of coverage in range [70%, 100%].

6.2.3 Repairability

Table 6 also lists the number of true smelly cell arrays that CACheck failed to repair (Fail to repair). Out of 1,586 true smelly cell arrays, CACheck was able to repair 1,540 (97.1%) of them. It shows that CACheck is effective for repairing smelly cell arrays automatically and correctly. The 46 smelly cell arrays that CACheck failed to repair involve cases of incomplete input variables (26), complex table structures (9), incomplete components (10), and too many wrong cells (1). We roughly observe that such "failed to repair" cases seem to relate to cell arrays with a level of low coverage. For example, 45.7% (21/46) cases occur to cell arrays with coverage below 50%, and this causes a failing rate of 42.0% (21/50). Besides, although 9 cases occur to cell arrays with coverage of 100%, the concerned failing rate is actually below as 0.8% (9/1,092).

Therefore, we draw the following conclusion:

CACheck can effectively detect smelly cell arrays. 70% can be a reliable threshold for effective smelly cell array detection, where it corresponds to a detection precision of 86.8%. Besides, CACheck can repair true smelly cell arrays with a 97.1% successful rate.

6.3 Comparison between CACheck and AmCheck

We now compare our CACheck with its predecessor, AmCheck, published earlier. As mentioned, AmCheck can only detect and repair homogeneous cell arrays, while CACheck can do so for both homogeneous and inhomogeneous ones.

We compared CACheck and AmCheck on the EUSES corpus. We partitioned comparison results into seven categories as earlier, according to different levels of coverage with respect to AmCheck's detected smelly cell arrays (i.e., how many cells their inferred $f_{pattern}$ can cover in these arrays). Table 7 lists the number of smelly cell arrays detected by AmCheck (AmCheck/SCA), and number of validated-as-true smelly cell arrays of AmCheck (AmCheck/TP). As mentioned, all these detected smelly cell are homogeneous. Fig. 7 also compares how the precision

TABLE 7
Comparisons between CACheck and AmCheck on the EUSES Corpus

Coverage	AmCheck		CACheck-Homo		Common	Missed by CACheck			Added by CACheck	
	SCA	TP	SCA	TP	SCA	SCA	TP	Equal smell	SCA	TP
100%	993	822	877	831	836	157	27	8	41	36
[90%, 100%)	133	88	101	85	97	36	6	1	4	3
[80%, 90%)	215	90	112	86	110	105	6	1	2	2
[70%, 80%)	119	50	64	48	63	56	3	3	1	1
[60%, 70%)	440	69	320	65	317	123	6	4	3	2
[50%, 60%)	1,293	76	619	75	608	685	4	3	11	3
[0%, 50%)	974	44	231	45	223	751	2	1	8	3
Total	**4,167**	**1,239**	**2,324**	**1,235**	**2,254**	**1,913**	**54**	**21**	**70**	**50**

changes with different levels of coverage for smelly cell arrays detected by CACheck and AmCheck. We observe that the overall precision for CACheck is 46.1% as earlier calculated, whereas that for AmCheck is only 29.7% (1,239/4,167). If we use 70% as the reliable threshold for coverage-based smelly cell array extraction, we observe that CACheck's precision is 86.8% as earlier calculated, whereas AmCheck's precision is only 71.9% (1,050/1,460).

6.3.1 Inhomogeneous Cell Arrays

As shown in Table 6, CACheck detected 351 true smelly inhomogeneous cell arrays (Inho-SCA/TP). As mentioned earlier, AmCheck was unable to detect any inhomogeneous cell array. As such, AmCheck missed at least 22.1% (351/1,586) true smelly cell arrays in the EUSES corpus.

6.3.2 Homogeneous Cell Arrays

Both CACheck and AmCheck can detect homogeneous cell arrays. Therefore, we are interested in their comparison at the aspect.

Table 7 also lists the number of smelly homogeneous cell arrays detected by CACheck (CACheck-Homo/SCA) and number of validated-as-true smelly homogeneous cell arrays of CACheck (CACheck-Homo/TP). We observe that CACheck's precision is 53.1% (1,235/2,324), whereas AmCheck's precision is only 29.7% (1,239/4,167). Similarly, if we use 70% as the reliable threshold for coverage-based smelly homogeneous cell array extraction, we observe that CACheck's precision is 91.0% (1,050/1,154), whereas AmCheck's precision is only 71.9% (1,050/1,460). This indicates that CACheck is more precise than AmCheck even if we only compare the detection of smelly homogeneous cell arrays. The improvement is mainly attributed to CACheck's filtering rules, which effectively prune invalid cell arrays.

We also concern how different the smelly homogeneous cell arrays detected by CACheck and AmCheck are. Table 7 shows the comparison results. We observe that 97.0% (2,254/2,324) smelly homogeneous cell arrays detected by CACheck could also be detected by AmCheck (Common SCA), but 45.9% (1,913/4,167) smelly cell arrays detected by AmCheck were missed by CACheck (Missed by CACheck/SCA). However, we note that this only compares detection results, but not validation results. Thus we manually validated all these 1,913 missed smelly cell arrays, and found that only 2.8% (54/1,913) of them are true smelly cell arrays (Missed by CACheck/TP). Besides, in 21 out of the 54 true smelly cell arrays (Equal smell), their corresponding smelly cells were already detected in other extracted cell arrays as reported by CACheck. As such, CACheck actually only missed 1.7% (33/1,913) true smelly homogeneous cell arrays. At the same, CACheck successfully filtered out 63.5% ((1,913 – 54) / (4,167 – 1,239)) false positives in AmCheck's detection results with a precision of 97.2% ((1,913 – 54)/1,913). For the 33 indeed missed (true smelly homogeneous) cell arrays, there are two cases: (1) Some cells do not have dependence similarity, and thus CACheck could not decide if they form any cell arrays. 17 cell arrays belong to this case. (2) Some cell arrays overlap with other cell arrays. As including the latter into the set of cell array candidates can introduce less conformance errors than including the former, the former cell arrays were missed. 16 cell arrays belong to this case.

From Table 7, we also observe that CACheck detected 70 additional smelly homogeneous cell arrays (Added by CACheck/SCA) that AmCheck could not detect, and 50 of them were validated as true (Added by CACheck/TP). This is more than what CACheck missed (33). These 50 cell arrays were not decided as cell arrays by AmCheck, as they do not follow AmCheck's cell array extraction heuristics. Nevertheless, CACheck's dependence similarity checking works for them and can precisely capture them. This indicates that CACheck's improved cell array extraction heuristics are effective and could take back AmCheck's missed true homogeneous cell arrays with a precision of 71.4% (50 / 70).

Therefore, we draw the following conclusion:

CACheck detects 401 (351 inhomogeneous and 50 homogeneous) additional true smelly cell arrays that are missed by AmCheck. If one sets 70% as the reliable threshold for coverage-based cell array extraction, CACheck's precision is 86.8%, higher than AmCheck's precision, 71.9%, and CACheck's precision on homogeneous cell array detection is even higher, 91.0%.

6.4 Comparison with other Techniques

We then compare our CACheck with Excel, UCheck/Dimension [4], [8] and CUSTODES [10]. These techniques mainly focus on syntactic smells (e.g., division by zero in Excel, type inconsistency in a formula [4], [8], and outliers in a cell cluster [10]), while CACheck focuses on semantic smells that violate computational semantics of concerned cell arrays, as mentioned earlier. These techniques adopt different mechanisms and may possibly detect smells that CACheck cannot (e.g., division by zero in Excel), and may also detect some smells that CACheck can as well (e.g., a cell suffers from both semantic and syntactic smells). In order to exhibit the differences in the scope of focused smells between these techniques and CACheck, our experimental comparisons rely on checking the applicability of other techniques to detecting those smells detected by CACheck. Specially, we investigate how many of CACheck's 8,139 validated-as-true smells can also be detected by these techniques.

6.4.1 Comparison with Excel

Microsoft Excel has built-in support for detecting syntactic smells (inconsistencies) in spreadsheet cells. However, its detection is subject to a few limitations. First, Excel considers only row-based or column-based ranges of three consecutive cells, and tries to detect smells in such ranges. Second, it detects only those smells that the middle cell's formula expression is syntactically different from those of its two adjacent cells, while the two adjacent cells' formula expressions are identical themselves. Besides, Excel also supports detecting some well-known calculation errors like division by zero. In Table 8, the Excel column shows that Excel can give warnings for only 2.2% (175/8,139) validated-as-true smelly cells detected by CACheck. Therefore, we consider that CACheck's capability is orthogonal to that of Excel's built-in checking mechanisms.

6.4.2 Comparison with UCheck/Dimension

The general idea of UCheck [4] and Dimension[8] is to exploit information in spreadsheets about labels and headers to check the type inconsistency of formulas in spreadsheet cells. UCheck uses the concept of unit to represent the type of a cell, e.g., cell E4 represents the harvest of apples in June, and cell F5 represents the harvest of oranges in July. UCheck defines some rules to enforce that one should not sum E4 and F5 up. Dimension detects smells when it finds that units of measurement are used incorrectly in formulas. For example, one should not add two cells with time and distance types, respectively, together.

CACheck has several advantages over UCheck/Dimension: (1) In the EUSES corpus, CACheck detected 8,139 true smelly cells, whereas UCheck/Dimension detected only 695 true smelly cells as reported in their latest work [9]. (2) CACheck can detect and repair smelly cells by suggesting intended formula patterns and calculated values, whereas UCheck/Dimension can only detect smelly cells without repair suggestions. (3) CACheck can detect and repair missing formula cells, whereas UCheck/Dimension cannot. (4) CACheck can detect and repair smelly cells that do not violate UCheck/Dimension's checking rules, e.g., smelly cells in Fig. 1(a). (5) CACheck does not rely on unreliable header/label information, whereas UCheck/Dimension relies much on such information, which may incur problems when the information is missing or incomplete.

The tool we obtained for experiments has implemented both UCheck and Dimension. So we conducted experiments by not splitting it into two parts, as their latest work [9] did. In Table 8, the UCheck/Dimension column shows that UCheck/Dimension detected only 0.2% (20/8,139)

TABLE 8
True Smelly Cells Detected by Different Techniques in the EUSES corpus

Category	CACheck	Excel*	UCheck/ Dimension*	CUSTODES*
cs101	22	0	0	5
database	3,650	57	1	1,317
filby	0	0	0	0
financial	1,491	36	4	627
forms3	8	0	0	3
grades	1,322	25	10	468
homework	285	29	0	96
inventory	981	22	1	434
jackson	0	0	0	0
modeling	380	6	4	90
personal	0	0	0	0
Total	**8,139**	**175**	**20**	**3,040**

* The numbers in the columns show how many of CACheck's validated-as-true smelly cells could be detected by the corresponding techniques.

validated-as-true smelly cells by checking type inconsistency. Still, we note that if a type inconsistency does not relate to any cell array, CACheck may not be able to detect it. Therefore, CACheck's capability is orthogonal to that of UCheck/Dimension, and they can detect different types of smells.

6.4.3 Comparison with CUSTODES

We also compare our CACheck to another more recent technique CUSTODES [10]. CUSTODES uses strong features (e.g., the same or similar R1C1 expressions and cell references) and weak features (e.g., same labels and font colors) to classify cells into different clusters. It then uses outlier detection to identify smelly cells in each cluster. For example, in Fig. 1(a), CUSTODES can extract {D4, D5, D9} as a cell cluster since the three cells share their cell references in the R1C1 representation. CUSTODES can also extract other four cell clusters for this example: {A3:A6}, {D2:D3}, {D6:D7} and {E3:E7}. We can observe that CUSTODES's concept of cell cluster is different from CACheck's concept of cell array.

Our investigation suggests that CACheck has several advantages over CUSTODES: (1) CUSTODES is learning-based and relies its threshold settings. Although its current implementation used default settings for all experiments, it did not guarantee for its best performance. It may cause false positives or negatives, as we observed in experiments. Nevertheless, CACheck does not have this issue. (2) CACheck can detect and repair smelly cells by suggesting intended formula patterns and corresponding values, whereas CUSTODES can only detect smelly cells without repair suggestions. (3) CUSTODES detects a cell cluster mainly by the equivalence of formulas contained by its cells, while CACheck detects a cell array mainly by the consecutive nature of its cells. Therefore, although CUSTODES can detect cell clusters that contain non-consecutive cells, it may also cause CUSTODES to miss important smelly cells. For example, in Fig. 1(a), cells D2 and D3 contain equivalent formulas in the R1C1 representation, and thus CUSTODES extracts them into a cell cluster. However, this prevents CUSTODES from further considering D4, D5,

TABLE 9
Comparisons between CACheck and AmCheck on the Sampled EUSES Spreadsheets

Category	Statistics of sampled spreadsheets						CACheck			AmCheck		
	SS	Work-sheet	For-mula	CA	SCA	Smelly cell	CA/TP	SCA/TP	Smelly cell/TP	CA/TP	SCA/TP	Smelly cell/TP
cs101	1	1	40	8	5	10	9/5	4/4	6/6	8/4	4/4	6/6
database	7	32	3,345	555	104	378	528/499	70/65	215/197	506/447	99/60	208/143
financial	11	30	1,372	201	74	276	193/184	61/57	213/190	183/168	58/48	187/140
forms3	1	1	52	12	0	0	8/8	0/0	0/0	40/4	36/0	72/0
grades	8	21	1,665	119	24	69	118/115	22/20	70/46	29/29	8/8	34/34
homework	8	15	1,000	88	23	48	69/57	18/11	34/16	68/48	25/10	25/10
inventory	8	11	9,967	68	15	30	67/67	15/15	30/30	163/47	131/15	146/30
modeling	6	17	595	85	7	23	86/85	8/7	24/23	67/67	7/7	23/23
Total	**50**	**128**	**18,036**	**1,136**	**252**	**834**	**1,078/1,020**	**198/179**	**592/508**	**1,064/814**	**368/152**	**701/386**

D6 and D7 into this cell cluster under default threshold settings, and as a result, CUSTODES fails to find that D2 and D3 are actually smelly. Nevertheless, CACheck does not have this problem for this example. (4) CACheck's smelly cell detection is based on a cell array's inferred formula pattern, while CUSTODES uses outlier detection to identify smelly cells. This might cause CUSTODES to misjudge situations and incur false positives. For example, in Fig. 1(a), CUSTODES considers {D4, D5, D9} as a cell cluster, but CUSTODES cannot figure out which of them are smelly since all three cells have different formulas. As a result, CUSTODES cannot identify the cell D4 as smelly, without taking more information under default threshold settings.

To validate our analysis for CUSTODES, we ran CUSTODES's implementation (obtained from its authors) on the EUSES corpus, and checked how many of CACheck's validated-as-true smelly cells could be detected by CUSTODES. In Table 8, the CUSTODES column gives the comparison results. We observe that CUSTODES detected only 37.4% (3,040/8,139) true smelly cells, which were detected by CACheck. Thus, CUSTODES missed a lot that were achieved by CACheck, and this suggests that CACheck has its unique advantages that CUSTODES cannot compare. Still, we note that CUSTODES may also detect certain smelly cells in its extracted cell clusters that may not necessarily contain consecutive cells. Therefore, we consider that our CACheck's capability is orthogonal to that of CUSTODES.

6.5 CACheck's Recall on Smelly Cell Array Detection for the EUSES Corpus

We next measure CACheck's recall rate on its smelly cell array detection for the EUSES corpus. As this measurement involves ground truths for all true smells in spreadsheets, which requires substantial manual effort, we conducted experiments only on a sampled subset of spreadsheets from the EUSES corpus. We also compared the recall rate between CACheck and AmCheck.

6.5.1 Experimental Subjects

Manually building ground truths for all true smells in EUSES spreadsheets is extremely difficult, as we are not authors of these spreadsheets and we also cannot find their corresponding authors. Therefore, we randomly sampled 50 spreadsheets from those with cell arrays for measuring CACheck's recall rate and comparing it with AmCheck (both requiring cell arrays). We manually obtained all well-formed and smelly cell arrays in these sampled spreadsheets as our ground truths.

We asked two postgraduate students to help sample spreadsheets and identify their contained cell arrays. For each randomly sampled spreadsheet from the EUSES corpus, we (authors and the two students) checked each of its contained worksheets for cell arrays individually, and then discussed our findings together. For a worksheet that does not contain any cell array or cannot be understood by any one of us, we removed it from consideration. If a spreadsheet contains at least one worksheet remaining, we kept this spreadsheet. Otherwise, it was also removed from consideration. We repeated this sampling process until 50 spreadsheets were collected.

Table 9 gives the statistics of the 50 sampled spreadsheets from the EUSES corpus. We observe that the 50 sampled spreadsheets (SS) are distributed in eight different categories, including 128 worksheets (Worksheet) and 18,036 formulas (Formula). Our manual inspection of these spreadsheets identifies a total of 1,136 cell arrays (CA). Among them, 252 cell arrays are smelly (SCA). We also identified 834 smelly cells (Smelly cell) from them.

We note that our sampling process might bias those spreadsheets that contain cell arrays. In our early experiments on the whole EUSES corpus, we found that CACheck detected many false smelly cell arrays in only few spreadsheets. For example, CACheck detected 569 (out of the total of 1,857) false smelly cell arrays in only two spreadsheets. AmCheck detected a similar number of false smelly cell arrays in the two spreadsheets. Thus, we need to filter out such spreadsheets clearly different from others

TABLE 10
Missed Cell Arrays on the Sampled EUSES Spreadsheets

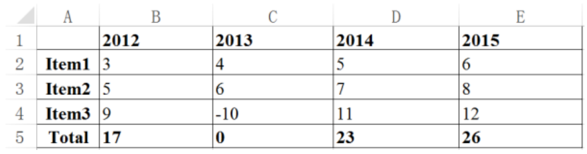
Category	CA	Common causes				CACheck		AmCheck		
		Plain value	Wrong cell	Empty line	Func	Detected	Wrong range	Detected	Wrong range	Inho (Common)
cs101	8	1	2	0	0	5	0	4	0	1(0)
database	555	17	31	6	0	499	2	447	25	33(4)
financial	201	14	3	0	0	184	0	168	0	16(0)
forms3	12	0	0	0	4	8	0	4	0	4(0)
grades	119	3	0	0	0	115	1	29	0	87(0)
homework	88	12	12	0	1	57	6	48	0	17(2)
inventory	68	0	0	0	1	67	0	47	0	21(1)
modeling	85	0	0	0	0	85	0	67	0	18(0)
Total	**1,136**	**47**	**48**	**6**	**6**	**1,020**	**9**	**814**	**25**	**197(7)**

in sampled spreadsheets. Our sampling process used "having cell arrays" as a criterion, since the two spreadsheets do not contain any cell array. Still, our sampling process also filtered out other spreadsheets that contain no cell array. According to our understanding, these spreadsheets likely fall in the category of low coverage, since "containing no cell array" and "detected false (smelly) cell array" are related. Thus, our sampling process might cause CACheck and AmCheck to detect less smelly cell arrays from spreadsheets with coverage in range [0%, 70%), since most of them should have been filtered out. Nevertheless, for coverage in other ranges like 100%, [90%, 100%), [80%, 90%) and [70%, 80%), CACheck's detected smelly cell arrays are distributed as (133, 14, 18, 9) and (1,184, 152, 164, 97), and AmCheck's detected smelly cell arrays are distributed as (139, 7, 36, 10) and (993, 133, 215, 119), respectively, for the 50 sampled spreadsheets and all EUSES spreadsheets. We observe that the two pairs of distribution data are comparable in percentage. As suggested earlier, a coverage of 70% can be a reliable threshold for effective smelly cell array detection in practice. Thus, our sampling process is still reasonable and can reflect CACheck's and AmCheck's performance and comparison in practice.

6.5.2 *Smelly Cell Arrays*

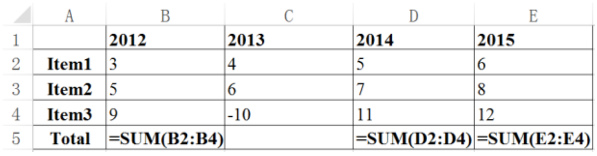
Table 9 also compares the detection results for CACheck and AmCheck on the sampled spreadsheets. For cell array extraction, the precision for CACheck and AmCheck is 94.6% (1,020/1,078) and 76.5% (814/1,064), respectively (CA/TP). For smelly cell array detection, the precision for CACheck and AmCheck is 90.4% (179/198) and 41.3% (152/368), respectively (SCA/TP). For smelly cell detection, the precision for CACheck and AmCheck is 85.8% (508/592) and 55.1% (386/701), respectively (Smelly cell/TP). We observe that CACheck detected more cell arrays and smelly cell arrays with a higher precision.

In this group of experiments, recall rate is our main focus. For cell array extraction, the recall rate for CACheck and AmCheck is 89.8% (1,020/1,136) and 71.7% (814/1,136), respectively. For smelly cell array detection, the recall rate for CACheck and AmCheck is 71.0% (179/252) and 60.3% (152/252), respectively. For smelly cell detection, the recall rate for CACheck and AmCheck is 60.9% (508/834) and 46.3% (386/834), respectively (Smelly cell). We observe that CACheck has largely improved the recall rate as compared to AmCheck, i.e., greatly reducing missed smelly cell arrays and smelly cells.



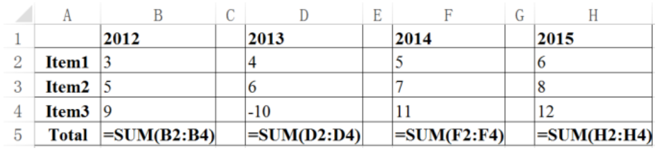
	A	B	C	D	E
1		**2012**	**2013**	**2014**	**2015**
2	**Item1**	3	4	5	6
3	**Item2**	5	6	7	8
4	**Item3**	9	-10	11	12
5	**Total**	**17**	**0**	**23**	**26**

(a) Cell array [B5:E5] with plain values only.



	A	B	C	D	E
1		**2012**	**2013**	**2014**	**2015**
2	**Item1**	3	4	5	6
3	**Item2**	5	6	7	8
4	**Item3**	9	-10	11	12
5	**Total**	**=SUM(B2:B4)**		**=SUM(D2:D4)**	**=SUM(E2:E4)**

(b) Cell array [B5:E5] with an empty cell.



	A	B	C	D	E	F	G	H
1		**2012**		**2013**		**2014**		**2015**
2	**Item1**	3		4		5		6
3	**Item2**	5		6		7		8
4	**Item3**	9		-10		11		12
5	**Total**	**=SUM(B2:B4)**		**=SUM(D2:D4)**		**=SUM(F2:F4)**		**=SUM(H2:H4)**

(c) Cell array [B5:H5] contains empty lines inside.

Fig. 8. Simplified spreadsheet examples with missed cell arrays, extracted from the EUSES corpus.

6.5.3 *False Negatives*

We then further analyze missed cell arrays (false negatives) on the 50 sampled spreadsheets for CACheck and AmCheck.

Table 10 lists missed cell arrays for CACheck and AmCheck, according to our ground truths. CACheck missed 10.2% ((1,136 – 1,020) / 1,136) cell arrays, whereas AmCheck missed 28.3% ((1,136 - 814) / 1,136) cell arrays. This big difference has been caused by AmCheck being unable to extract inhomogeneous cell arrays, as mentioned earlier.

Table 10 also lists four common reasons (Common causes) explaining why false negatives occurred to both CACheck and AmCheck: (1) 4.1% (47/1,136) cell arrays (i.e., 18.7% (47/252) smelly cell arrays) contain plain values only (Plain value). It is difficult to figure out whether the concerned cells with only plain values contain the same formula pattern or not, since there is no clue on how these values are calculated. One example is the cell array [B5:E5] in Fig. 8(a). (2) 4.2% (48/1,136) cell arrays contain empty cells, string cells, or error cells (Wrong cell). Both CACheck and AmCheck split such cell arrays into multiple smaller ones, as separated by such cells. One example is the cell array [B5:E5] in Fig. 8(b). (3) 0.5% (6/1,136) cell arrays contain empty lines inside for layout purposes. Similarly, CACheck and AmCheck split them into multiple smaller ones.

TABLE 11
Comparisons of CACheck's Detected Cell Arrays on the EUSES and Enron Corpora

Corpus	Cell array				Smelly cell array						
	CA	Homo	Inho	Inho/CA	SCA	SHomo	SInho	SCA/CA	MISS	INCO	Both
EUSES	22,177	16,928	5,249	23.7%	3,443	2,324	1,119	15.5%	1,849	1,699	105
Enron	455,159	342,992	112,167	24.6%	58,514	28,815	29,699	12.9%	43,343	16,993	1,822
Total	**477,336**	**359,920**	**117,416**	**24.6%**	**61,957**	**31,139**	**30,818**	**13.0%**	**45,192**	**18,692**	**1,927**

TABLE 12
Detected and Validated Smelly Cell Arrays on the Enron Corpus

Coverage	SCA	Sample		Estimated TP
		SCA	TP	
100%	21,692	100	91	19,740
[90%, 100%)	3,771	100	88	3,318
[80%, 90%)	2,047	100	66	1,351
[70%, 80%)	1,521	100	60	913
[60%, 70%)	3,135	100	20	627
[50%, 60%)	16,107	100	3	483
[0%, 50%)	10,241	100	12	1,229
Total	**58,514**	**700**	**340**	**27,661**

One example is the cell array [B5:H5] in Fig. 8(c). (4) 0.5% (6/1,136) cell arrays contain complex Excel functions (Func), such as STDEV and SUMPRODUCT, which the current CACheck and AmCheck implementations do not support and need extension. Besides the four common reasons, CACheck and AmCheck may extract larger or smaller ranges of consecutive cells rather than expected, leading to missed cell arrays. Table 10 also lists the number of such missed cell arrays (Wrong range). CACheck and AmCheck missed 0.8% (9/1,136) and 2.2% (25/1,136) such cells arrays, respectively.

Finally, we found that among those cell arrays missed by AmCheck, 61.2% (197/322) of them are inhomogeneous ones. This part contributes to a large portion of AmCheck's low recall rate, as compared to CACheck, which was designed to be able to extract inhomogeneous cell arrays.

Therefore, we draw the following conclusion:

Compared with AmCheck, CACheck largely improves the recall rate of cell array extraction (from 71.7% to 89.8%), and that of smelly cell array detection (from 60.3% to 71.0%).

6.6 Cell Array Detection for the Enron Corpus

The previous experiments and comparisons show that our CACheck performs much better than AmCheck, no matter on the precision or recall rate, with respect to the EUSES corpus. In the following, we extend our evaluation on CACheck to another even huge Enron corpus. We also compare CACheck's evaluation results to those on the EUSES corpus.

6.6.1 Cell Array Detection

Table 11 compares CACheck's detected cell arrays on the EUSES and Enron corpora. It studies detected cell arrays (Cell array) and detected smelly cell arrays (Smelly cell array). It lists the number of cell arrays (CA), number of smelly cell arrays (SCA), and number of cell arrays suffering from missing formula smells (MISS), inconsistent formula smells (INCO), and both smells (Both). Table 11 also lists the numbers of homogeneous (Homo and SHomo) and inhomogeneous (Inho and SInho) cell arrays. We observe that the percentage of inhomogeneous cell arrays against all cell arrays is 23.7% and 24.6% (Inho/CA), respectively, for the EUSES and Enron corpora. They are close to each other. Besides, the percentage of smelly cell arrays is 15.5% and 12.9% (SCA/CA), respectively, for the two corpora. They are also close and comparable.

Therefore, we draw the following conclusion:

Smelly cell arrays are also commonly detected in the Enron corpus. The EUSES and Enron corpora have comparable percentages on inhomogeneous cell arrays and smelly cell arrays.

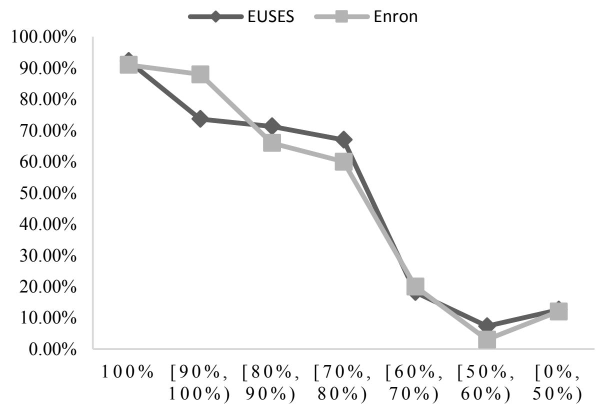


Fig. 9. Precision comparison on smelly cell array detection for different levels of coverage on the EUSES and Enron corpora (horizontal axis: coverage category, vertical axis: detection precision).

6.6.2 Detection Precision

We then investigate CACheck's precision on smelly cell array detection for the Enron corpus.

Following our earlier experimental process for the precision study on the EUSES corpus (Section 6.2), we partition CACheck's detected smelly cell arrays into seven categories according to their levels of coverage (i.e., how many cells in percentage their inferred $f_{pattern}$ can cover in these arrays). Table 12 lists the number of smelly cell arrays (SCA) for each category. Since the total number (58,514) is huge and much more than that (3,443) for the EUSES corpus, it is almost impossible to validate all of them. So we adopted another soft way by sampling and estimation. We randomly sampled 100 smelly cell arrays (Sample/SCA) for each category and manually validated them. This accounts for 700 smelly cell arrays, which occupy 1.2% (700/58,514) of all detected cell arrays. For each category, Table 12 lists the number of validated-as-true smelly cell arrays (Sample/TP).

Fig. 9 compares CACheck's precision on smelly cell array detection for different levels of coverage on the Enron and EUSES corpora. We observe that the two groups of

TABLE 13
Comparisons of CACheck's Recall Rate on the Sampled EUSES and Enron Spreadsheets

Corpus	Statistics of sampled spreadsheets						CACheck (recall rate)		
	SS	Worksheet	Formula	CA	SCA	Smelly cell	CA	SCA	Smelly cell
EUSES	50	128	18,036	1,136	252	834	1,020 (89.8%)	179 (71.0%)	508 (60.9%)
Enron	50	102	27,547	1,175	128	483	1,058 (90.0%)	93 (72.7%)	321 (66.5%)

precision values are close to each other. Based on validation results on sampled smelly cell arrays, we estimate that there are a total of 27,661 smelly cell arrays (Estimated TP in Table 12) in the Enron corpus. If one considers 70% as the reliable threshold for coverage-based cell array extraction, CACheck's precision on smelly cell array detection for the Enron corpus is 87.2% (25,322/29,031), while the precision for the EUSES corpus is 86.8%, as earlier measured. This indicates that CACheck has a comparable precision on smelly cell array detection for the Enron and EUSES corpora.

6.6.3 Detection Recall

We next investigate CACheck's recall rate on smelly cell array detection for the Enron corpus.

Similarly, we followed our earlier experimental process for the recall study on the EUSES corpus (Section 6.5), and randomly sampled 50 spreadsheets from the Enron corpus. However, we found many similar spreadsheets and worksheets in the Enron corpus. This was caused by the fact that the Enron corpus contains many series of different versions of spreadsheets or worksheets, which were extracted from email correspondences [18], [26]. Thus, we only selected one spreadsheet or worksheet if multiple similar ones were found. We believe that this treatment could make our sampled spreadsheets more representative for the Enron corpus.

Table 13 gives the statistics of the sampled spreadsheets from the Enron corpus (columns 2–7 for row Enron). Our sampled 50 Enron spreadsheets (SS) contained 102 worksheets (Worksheet) and 27,547 formulas (Formula). Following our earlier inspection process for the EUSES corpus, our manual inspection of these Enron spreadsheets identified a total of 1,175 cell arrays (CA). Among them, 128 cell arrays are smelly (SCA). We also identified 483 smelly cells (Smelly cell) from smelly cell arrays.

Table 13 also compares CACheck's recall rates on the EUSES and Enron corpora (columns 8–10). We observe that CACheck's recall rates for cell array detection (89.8% vs. 90.0%; CA), smelly cell array detection (71.0% vs. 72.7%; SCA) and smelly cell detection (60.9% vs. 66.5%; Smelly cell) are close to each other for the EUSES and Enron corpora. This suggests that CACheck has a comparable recall rate on smelly cell array detection for the EUSES and Enron corpora.

Therefore, we draw the following conclusion:

CACheck has a comparable precision (86.8% vs. 87.2%) and recall rate (71.0% vs. 72.7%) on smelly cell array detection for the EUSES and Enron corpora.

6.7 Research Questions Revisited

Finally, we revisit our research questions RQ4–7:

RQ4 (Precision): *Can CACheck detect and repair smelly cell arrays precisely?*

- One may use the coverage of 70% as a reliable threshold for CACheck's effective smelly cell array detection, and this corresponds to a satisfactory precision of 86.8%.
- CACheck is able to repair 1,540 (97.1%) of its 1,586 detected true smelly cell arrays.

RQ5 (Recall): *Can CACheck detect smelly cell arrays with a high recall rate?*

- CACheck's recall rate for cell array extraction and smelly cell array detection is 89.8% and 71.0%, respectively, which is both promising.

RQ6 (Comparison): *How is CACheck compared with existing techniques, e.g., AmCheck, Excel, UCheck/Dimension and CUSTODES?*

- CACheck detects 401 (out of a total of 1,586) additional true smelly cell arrays that are missed by AmCheck.
- If one uses the coverage of 70% as a reliable threshold for smelly cell array detection, the precision for CACheck and AmCheck is 86.8% and 71.9%, respectively.
- The recall rate for CACheck and AmCheck on smelly cell array detection is 71.0% vs. 60.3%, respectively.
- Excel, UCheck/Dimension and CUSTODES can detect only 2.2%, 0.2% and 37.4% (out of 8,139) CACheck's validated-as-true smelly cells, respectively.

RQ7 (Consistency): *Can CACheck obtain consistent results on different spreadsheet corpora, such as the EUSES and Enron corpora?*

- The EUSES and Enron corpora have comparable percentages on inhomogeneous cell arrays and smelly cell arrays against all detected cell arrays.
- The precision of smelly cell array detection on the EUSES and Enron corpora is comparable (86.8% vs. 87.2%).
- The recall rate of smelly cell array detection on the EUSES and Enron corpora is also comparable (71.0% vs. 72.7%).

6.8 Threats to Validity

While our experimental evaluation shows that CACheck is promising for detecting and repairing ambiguous computation smells in real-life spreadsheets, we discuss some potential threats in our evaluation.

Representativeness of studied spreadsheets. To generalize the conclusions made in our experimental evaluation, the studied spreadsheets as experimental subjects should be representative. We selected the EUSES and Enron corpora, two well-known and large spreadsheet corpora, which have been well recognized and widely used for spreadsheet-related research studies [9], [25], [31], [53].

Smelly cell array validation. Our experimental evaluation on CACheck's precision on smelly cell array detection, as well as its comparisons to existing work, relies on the validation of its detected smelly arrays. Due to the fact of lack of responsible authors, we manually validated these detected smelly arrays in their concerned spreadsheets. In order to reduce possible mistakes we could introduce, we adopted some helping strategies: (1) Some tables or worksheets are similar to each other, and they help us double check our validation results. (2) We tried to understand the semantics of cell arrays according to their related labels, rather than treating cell array extraction and smelly cell array detection simply as a syntactic process. (3) Some cells in smelly cell arrays, although containing unusual plain values (e.g., 12,498), could be successfully recovered by our inferred formula pattern. Such discovered knowledge (missing formula) helped us double check related cell arrays and their intended formula patterns.

Spreadsheet selection for the recall study. The recall study requires ground truths for all studied spreadsheets. To reduce our efforts, we sampled 50 spreadsheets from the EUSES corpus, and manually identified their contained cell arrays and smelly ones. Our sampling was random, but focusing on those spreadsheets that we can fully understand (i.e., those spreadsheets any one of us could not understand or we do not agree on with each other were discarded). Similar treatments were also applied to the recall study on the Enron corpus. Still, as mentioned earlier, our sampled spreadsheets have comparable distributions of smelly cell arrays with respect to the whole EUSES corpus. We consider that this should alleviate possible threats in our sampling process. If one, who has different domain knowledge, samples spreadsheets from the EUSES corpus in a different way, the corresponding recall rate might be different (not large, we believe). We note that our experimental process can still be similarly applied and thus reusable.

7 RELATED WORK

In this section, we present and discuss related work in recent years. We focus on those pieces of work that concern spreadsheet quality (e.g., spreadsheet errors, auditing, error detection, debugging and testing), and techniques related to our CACheck approach (e.g., program synthesis and semantic bug analysis).

Spreadsheet errors. Spreadsheet errors are common [44], [45], [46]. They can cause serious financial loss [54]. The ambiguous computation smells studied in this article may not cause errors immediately, but would degrade spreadsheets' quality gradually and boost potential errors. Spreadsheets suffering from ambiguous computation smells contain unclear or even conflicting computational semantics, which make them difficult to maintain in a correct way.

Spreadsheet auditing. Auditing is a way to maintain for spreadsheets' quality. To facilitate auditing, Clermont et al. [11], [12], [41] proposed high-level structures (*logical areas*) by aggregating cells to help end users understand large spreadsheets, such as *copy equivalence* (i.e., two cells' formulas are identical), *logical equivalence* (i.e., two cells' formulas differ only in constant values and absolute references), *structural equivalence* (i.e., two cells' formulas differ in constant values and absolute/ relative references, and the same operators or functions are applied in the same order). These logical areas assist end users to better understand conceptual models behind spreadsheets, and to find smelly cells in spreadsheets more easily. Thus, some commercial spreadsheet tools (e.g., OAK [57], EXChecker [58], Spreadsheet Detective [60] and Spreadsheet Auditor [61]) have adopted this idea. The concept of cell array proposed in this article is similar to copy equivalence. However, several differences exist: (1) The auditing technique visualizes logical areas in spreadsheets, and asks end users to spot dangerous parts in them, while CACheck spots smelly cells automatically. (2) The auditing technique requires that, for two cells in a certain kind of logical area, their formulas should have the same operators and functions applied in the same order. This would exclude cell arrays, like [D2:D7] and [B9:C9] in Fig. 1(a), from consideration. (3) CACheck can aggregate smelly cells that do not satisfy requirements of logical areas, such as cells with inconsistent or even missing formulas, while the auditing technique cannot. (4) CACheck can repair smelly cells by suggesting their intended formula patterns, while the auditing technique does not support this. (5) CACheck can rank smelly cell arrays according to their different levels of coverage, while the auditing technique needs end users to audit cells inside or adjacent to logical areas individually.

Spreadsheet error detection and debugging. Various techniques have been proposed to detect and debug errors in spreadsheets. A recent survey [35] provides in-depth reviews of these techniques. It summarizes many main spreadsheet error detection techniques. For example, UCheck [4] and dimension inference [8] use a type system to check unit and dimension errors, respectively. They focus on whether units can be combined correctly into one cell. Smellsheet Detective [14], [15] detects statistical smells, type smells, content smells and functional dependence smells. Hermans et al. proposed visualizing spreadsheets by dataflow graphs [28], and detected inter-worksheet smells in these graphs [30]. They also proposed detecting smells from data clones [31], spreadsheet formulas [29] and lookup functions [25]. Commercial spreadsheet tools (e.g., Spreadsheet Professional [56], OAK [57], EXChecker [58] and PerfectXL [59]) can detect various syntactic errors (e.g., referencing empty cells, division by zero, and so on). These pieces of work focus more or less on syntactic errors, while our CACheck focuses on missing formula and inconsistent formula smells, which concern semantic errors. Our CACheck also detects conformance errors caused by ambiguous computation smells. Its scope is thus orthogonal to existing work. Besides, according to the spreadsheet research survey [35], due to the structure of spreadsheets (e.g., computations are hidden behind the cells), locating spreadsheet errors is typically a hard task. Thus, many debugging techniques have been developed for spreadsheets, e.g., slicing-based debugging [47], spectrum-based fault localization [33], [48], constraint-based fault localization [34], repair-based debugging techniques [3], and so on. These debugging techniques usually depend on users' expectations or judgments about outputs of certain cells. Therefore, our CACheck differs from these debugging techniques in that it does not rely on such expectations or judgments to work.

Spreadsheet modeling and testing. Constructing rigorous models (explicit abstractions) for spreadsheets [2], [13], [27] can help end users reduce chances of introducing ambiguous computation smells. Yet, constructing such models from spreadsheets can be challenging. Its effectiveness depends largely on the correctness of underlying spreadsheets, and ambiguous computation smells can reduce its precision and thus effectiveness. Instead of introducing an explicit abstraction, XanaSheet [32] employs origin tracking techniques to maintain a live connection between the source and destination of copy-paste actions. Whenever a copied formula is edited, the modification can be transformed and replayed on the original and all other copies. Thus, inconsistent modifications of copy-pasted cells could be avoided. Our CACheck also concerns this problem and addresses it by using both heuristics and formula synthesis. Spreadsheet testing (possibly based on models) [1], [5], [20], [37] is a related topic, and its error detection capabilities need to rely on test oracles provided by users (e.g., test formulas extracted from reference [24] or manual confirmation directly from users). Our CACheck extracts partial computational semantics from cell contents and recovers intended formula patterns. Thus, CACheck does not require explicit test oracles to work. Ambiguous computation smells may also affect spreadsheet testing, and CACheck assists spreadsheet testing by detecting and repairing smelly cells.

Program synthesis. Our CACheck is based on component-based program synthesis [22], [36]. Typically, the program synthesis technique [22], [36] can automatically generate loop-free programs based on a user-provided input-output oracle (e.g., input-output pairs [36] or specifications [22]) and components. Regarding our problem (automatically detecting smells in smelly cell arrays without user intervention), the input-output oracle and components are unavailable for synthesizing a smelly cell array's formula pattern. So the original program synthesis technique [22], [36] cannot be directly used by CACheck. Thus, CACheck needs to extract such components and input-output oracle (i.e., input-output pairs and specifications) from smelly cell arrays, and alleviates their noises when adapting the program synthesis technique [22], [36] for spreadsheet smell detection. Program synthesis has also been used for other purposes in the spreadsheet research, e.g., string transformation from examples [21], table transformation [23], and number transformation [50]. In this article, we use program synthesis in a novel way to detect and repair ambiguous computation smells in spreadsheets, by recovering computational semantics aligned to the actual computations in smelly cell arrays.

Semantic bugs. Similar to smells in spreadsheets, semantic bugs are also a dominant cause for software failures [39], [51]. Most semantic bugs require domain knowledge to understand, detect and repair [39]. MUVI [40] and DefUse [45] can detect semantic bugs related to inconsistent updates to correlated multi-variables and dataflow intentions, respectively, in software. They rely on invariant mining and detection techniques. Our CACheck uses a different approach by inferring intended computational semantics by heuristics and program synthesis techniques.

8 Conclusion

In this article, we study the problem of extracting cell arrays and detecting ambiguous computation smells from spreadsheets. Such smells are caused by end users' ad hoc modifications to spreadsheet cells that should stick to certain computational semantics. We propose a novel approach, CACheck, to detect and repair ambiguous computation smells by inferring intended formula patterns for smelly cell arrays in spreadsheets. This also helps detect challenging conformance errors in spreadsheets, which would otherwise be left unnoticed. Our experimental evaluation based on two large-scale spreadsheet corpora reveals that smelly cell arrays are common, and CACheck is capable of detecting smelly cells effectively with a high precision and recall rate.

In future, we plan to study more spreadsheets and identify other types of ambiguous computation smells. For example, in our recall study (Section 6.5.3), we found that a non-negligible proportion (about 18.7%) of true smelly cell arrays have only plain values (i.e., no formula at all for all concerned cells). Our current CACheck is still unable to detect such smelly cell arrays and synthesize formula patterns for them. We plan to further investigate them and come up with an approach to detecting them and inferring their intended formula patterns. Besides, CACheck extracts cell arrays that contain only consecutive cells, and this may prevent it from detecting challenging smelly cell arrays that contain non-consecutive cells. We are also interested in extending CACheck for such cases.

Acknowledgment

The authors wish to thank TSE editors and anonymous reviewers for their valuable comments on improving this article. This work was supported in part by Beijing Natural Science Foundation (4164104), National Key Research and Development Plan (2016YFB1000803), Research Grants Council (General Research Fund 611811) of Hong Kong, National Natural Science Foundation (Grant Nos. 61472174, 91318301, 61321491) of China, and the Collaborative Innovation Center of Novel Software Technology and Industrialization of China.

References

[1] 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), 2006, pp. 43–50.

[2] R. Abraham and M. Erwig, “Inferring Templates from Spreadsheets,” in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006, pp. 182–191.

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

[4] R. Abraham and M. Erwig, “UCheck: A Spreadsheet Type Checker for End Users,” *J. Vis. Lang. Comput.*, vol. 18, no. 1, pp. 71–95, 2007.

[5] R. Abraham and M. Erwig, “Mutation Operators for Spreadsheets,” *IEEE Trans. Softw. Eng.*, vol. 35, no. 1, pp. 94–108, 2009.

[6] D. W. Barowy, D. Gochev, and E. D. Berger, “CheckCell: Data Debugging for Spreadsheets,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages Applications (OOPSLA)*, 2014, pp. 507–523.

[7] M. Burnett and M. Erwig, “Visually customizing inference rules about apples and oranges,” in *IEEE Symposia on Human Centric Computing Languages and Environments (HCC)*, 2002, pp. 140–148.

[8] C. Chambers and M. Erwig, “Automatic Detection of Dimension Errors in Spreadsheets,” *J. Vis. Lang. Comput.*, vol. 20, no. 4, pp. 269–283, 2009.

[9] C. Chambers and M. Erwig, “Reasoning About Spreadsheets with Labels and Dimensions,” *J. Vis. Lang. Comput.*, vol. 21, no. 5, pp. 249–262, 2010.

[10] S.C. Cheung, W. Chen, Y. Liu, and C. Xu, “CUSTODES: Automatic Spreadsheet Cell Clustering and Smell Detection Using Strong and Weak Features,” in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 464-475.

[11] M. Clermont, “A Scalable Approach to Spreadsheet Visualization,” 2003.

[12] M. Clermont and R. Mittermeir, “Auditing Large Spreadsheet Programs,” in *Proceedings of the International Conference on Information Systems Implementation and Modeling*, 2003, pp. 87–97.

[13] 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)*, 2010, pp. 93–100.

[14] 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)*, 2012, pp. 243–244.

[15] J. Cunha, J. P. Fernandes, H. Ribeiro, and J. Saraiva, “Towards a Catalog of Spreadsheet Smells,” in *Computational Science and Its Applications*, 2012, pp. 202–216.

[16] J. S. Davis, “Tools for Spreadsheet Auditing,” *Int. J. Hum.-Comput. Stud.*, vol. 45, no. 4, pp. 429–442, 1996.

[17] W. Dou, S.C. Cheung, and J. Wei, “Is Spreadsheet Ambiguity Harmful? Detecting and Repairing Spreadsheet Smells Due to Ambiguous Computation,” in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 848–858.

[18] W. Dou, L. Xu, S.C. Cheung, C. Gao, J. Wei, and T. Huang, “VEnron: A Versioned Spreadsheet Corpus and Related Evolution Analysis,” in *Proceedings of the 38th International Conference on Software Engineering (ICSE SEIP)*, 2016, pp. 162-171.

[19] M. Fisher and G. Rothermel, “The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[20] 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)*, 1998, pp. 198–207.

[21] S. Gulwani, “Automating String Processing in Spreadsheets Using Input-output Examples,” in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011, pp. 317–330.

[22] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, “Synthesis of Loop-free Programs,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 62–73.

[23] W. R. Harris and S. Gulwani, “Spreadsheet Table Transformations from Examples,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 317–328.

[24] F. Hermans, “Improving Spreadsheet Test Practices,” in *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2013, pp. 56–69.

[25] F. Hermans, E. Aivaloglou, and Bas Jansen, “Detecting Problematic Lookup Functions in Spreadsheets,” in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2015, pp. 153–157.

[26] F. Hermans and E. Murphy-Hill, “Enron’s Spreadsheets and Related Emails: A Dataset and Analysis,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015, pp. 7–16.

[27] 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)*, 2010, pp. 52–75.

[28] F. Hermans, M. Pinzger, and A. van Deursen, “Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams,” in *Proceedings of International Conference on Software Engineering (ICSE)*, 2011, pp. 451–460.

[29] F. Hermans, M. Pinzger, and A. van Deursen, “Detecting Code Smells in Spreadsheet Formulas,” in

Proceedings of International Conference on Software Maintenance (ICSM), 2012, pp. 409–418.

[30] 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)*, 2012, pp. 441–451.

[31] 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)*, 2013, pp. 292–301.

[32] F. Hermans and T. van der Storm, "Copy-Paste Tracking: Fixing Spreadsheets Without Breaking Them," in *Proceedings of the 1st International Conference on Live Coding (ICLC)*, 2015.

[33] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner, "On the Empirical Evaluation of Fault Localization Techniques for Spreadsheets," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2013, pp. 68–82.

[34] D. Jannach and T. Schmitz, "Model-based Diagnosis of Spreadsheet Programs: A Constraint-based Debugging Approach," *Autom. Softw. Eng.*, vol. 23, no. 1, pp. 105–144, 2014.

[35] D. Jannach, T. Schmitz, B. Hofer, and F. Wotawa, "Avoiding, Finding and Fixing Spreadsheet errors – A Survey of Automated Approaches for Spreadsheet QA," *J. Syst. Softw.*, vol. 94, pp. 129–150, 2014.

[36] S. Jha, S. Gulwani, S. A. Seshia, and Ashish Tiwari, "Oracle-guided Component-based Program Synthesis," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010, pp. 215–224.

[37] 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)*, 2000, pp. 230–239.

[38] B. Klimt and Y. Yang, "Introducing the Enron Corpus," in *First Conference on Email and Anti-Spam (CEAS) in Cooperation with AAAI and The International Association for Cryptologic Research and The IEEE Technical Committee on Security and Privacy*, 2004.

[39] 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, pp. 31–44.

[40] 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)*, 2007, pp. 103–116.

[41] R. Mittermeir and M. Clermont, "Finding High-level Structures in Spreadsheet Programs," in *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE)*, 2002, pp. 221–232.

[42] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*, 2008, pp. 337–340.

[43] R. Panko, "Facing the Problem of Spreadsheet Errors," *Decis. Line*, vol. 37, no. 5, pp. 8–10, 2006.

[44] R. R. Panko and S. Aurigemma, "Revising the Panko–Halverson Taxonomy of Spreadsheet Errors," *Decis. Support Syst.*, vol. 49, no. 2, pp. 235–244, 2010.

[45] S. G. Powell, K. R. Baker, and B. Lawson, "A Critical Review of the Literature on Spreadsheet Errors," *Decis. Support Syst.*, vol. 46, no. 1, pp. 128–138, 2008.

[46] K. Rajalingham, D. R. Chadwick, and B. Knight, "Classification of Spreadsheet Errors," *ArXiv08054224 Cs*, 2008.

[47] J. Reichwein, G. Rothermel, and M. Burnett, "Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging," *ACM SIGPLAN Not.*, vol. 35, no. 1, pp. 25–38, 1999.

[48] J. R. Ruthruff, M. Burnett, and G. Rothermel, "Interactive Fault Localization Techniques in a Spreadsheet Environment," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 213–239, 2006.

[49] 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)*, 2010, pp. 160–174.

[50] R. Singh and S. Gulwani, "Synthesizing Number Transformations from Input-Output Examples," in *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, 2012, pp. 634–651.

[51] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug Characteristics in Open Source Software," *Empir. Softw. Eng.*, vol. 19, no. 6, pp. 1665–1705, 2014.

[52] J. Walkenbach, *Excel 2013 Power Programming with VBA*, vol. 13. Wiley.com, 2013.

[53] R. Zhang, C. Xu, S. C. Cheung, P. Yu, X. Ma, and J. Lu, "How Effective can Spreadsheet Anomalies be Detected: An Empirical Study," *J. Syst. Softw.*, 2016.

[54] "European Spreadsheet Risks Interest Group." [Online]. Available: http://www.eusprig.org/horror-stories.htm. [Accessed: 01-Mar-2015].

[55] "How to use the Auto Fill Options button in Excel." [Online]. Available: http://support.microsoft.com/kb/291359. [Accessed: 10-Mar-2015].

[56] "Spreadsheet Professional." [Online]. Available: http://www.spreadsheetinnovations.com/. [Accessed: 09-Apr-2016].

[57] "OAK Operis Analysis Kit." [Online]. Available: http://www.operisanalysiskit.com/. [Accessed: 09-Apr-2016].

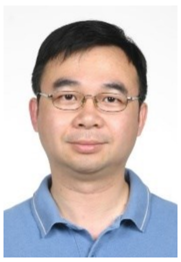
[58] "EXChecker." [Online]. Available: http://www.finsburysolutions.com/exchecker/.

[Accessed: 09-Apr-2016].
[59] “PerfectXL.” [Online]. Available: http://info-tron.nl/en/home-2/. [Accessed: 09-Apr-2016].
[60] “Spreadsheet Detective.” [Online]. Available: http://www.spreadsheetdetective.com/. [Accessed: 09-Apr-2016].
[61] “Spreadsheet Auditor.” [Online]. Available: https://www.spreadsheetauditor.com/. [Accessed: 09-Apr-2016].
[62] “CACheck project.” [Online]. Available: http://www.tcse.cn/~wsdou/project/cellarray. [Accessed: 14-Apr-2016].
[63] “Apache POI - the Java API for Microsoft Documents.” [Online]. Available: http://poi.apache.org/. [Accessed: 13-Feb-2016].

Wensheng Dou received the doctoral degree in computer science and technology from the University of Chinese Academy of Sciences in 2015. He is an assistant professor in the Institute of Software, Chinese Academy of Sciences (ISCAS). His research interests include end-user software engineering, program analysis, testing and debugging, cloud computing, and big data software engineering.

Chang Xu received the doctoral degree in computer science and engineering from The Hong Kong University of Science and Technology. In 2010, he joined Nanjing University, where he is an associate professor with the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology. He participates actively in program and organizing committees of major international software engineering conferences. He co-chaired the FSE 2014 SEES Symposium and MIDDLEWARE 2013 Doctoral Symposium. His research interests include big data software engineering, software testing and analysis, and adaptive and embedded system.

S.C. Cheung received his doctoral degree in Computing from the Imperial College London. In 1994, he joined The Hong Kong University of Science and Technology, where he is a full professor of Computer Science and Engineering. He participates actively in program and organizing committees of major international software engineering conferences. He was the General Chair of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014). He was a director of the Hong Kong R & D Center for Logistics & Supply Chain Management Enabling Technologies. His research interests include program analysis, testing and debugging, big data software, cloud computing, internet of things, and mining software repository.



Jun Wei received the PhD degree in computer science in 1997 from the Wuhan University, China. He was a visiting researcher in the CSE Department, Hong Kong University of Science and Technology, in 2000. He is a professor in the Institute of Software, Chinese Academy of Sciences (ISCAS). His area of research is software engineering and distributed computing, with emphasis on middleware-based distributed software engineering.