Automated Refactoring of Nested-IF Formulae in Spreadsheets

Jie Zhang[1,3], Shi Han[2], Dan Hao[1], Lu Zhang[1], Dongmei Zhang[2]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, China
[2]Microsoft Research, Beijing, China
[3] Department of Computer Science, University College London, London, UK
[1]{jie.zhang,haodan,zhanglucs}@pku.edu.cn, [2]{shihan,dongmeiz}@microsoft.com

ABSTRACT

Spreadsheets are the most popular end-user programming software, where formulae act like programs and also have smells. One well recognized smell is the use of *nested-IF* expressions, which have low readability and high cognitive cost for users, and are error-prone during reuse or maintenance. End users usually lack essential programming language knowledge and skills to tackle or even realize this problem, yet no automatic approaches are currently available.

This paper proposes the first exploration of the nest-if usage status against two large-scale spreadsheet corpora containing over 80,000 industry-level spreadsheets. It turns out the use of *nested-IF* expressions are surprisingly common among end users. We then present an approach to tackling this problem through automatic formula refactoring. The general idea of the automatic approach is two-fold. First, we detect and remove logic redundancy based on the AST of a formula. Second, we identify higher-level semantics that have been represented with fragmented and scattered syntax, and reassemble the syntax using concise built-in functions. A comprehensive evaluation with over 28 million *nested-IF* formulae reveals that the approach is able to relieve the smell of over 90% of *nested-IF* formulae.

1 INTRODUCTION

End-user programming referes to the activities that support end users who are not professional developers to program. Spreadsheets are the most popular end-user programming tools [1]. One of the most important enabling factors is that spreadsheets provide immediate feedback so users can make a change in one place and immediately see the results [2]. Underneath such an advantage, formulae play an important role as end-user friendly programs. However, end-users typically lack essential knowledge and skills of programming, and are easier to write formulae with bad smells [3].

One of the well-recognized spreadsheet smells are *nested-IF* expressions [3, 4]. *IF* functions[1] (i.e., the syntax is $IF(condition, value\_if\_true, value\_if\_false)$) are widely used spreadsheet functions. *Nested-IF* expressions happen when end users write an *IF* function inside another *IF* function. Formulae with *nested-IF* expressions are notorious as being complex, unreadable, error-prone, as well as hard to debug and maintain [3–7]. Although industrial spreadsheet applications allow end users to nest many IF functions, they are also trying to help avoid this bad practice. For example, the documentation of Microsoft Excel IF function [8] lists several serious disadvantages (e.g., hard to ensure 100% accuracy, difficult to maintain, and complex) of using multiple nest-if statements as a caution to end users.

[1] Functions are predefined built-in formulae in spreadsheet systems.

This kind of warning from spreadsheet applications, however, is far from enough. Our investigation against a large-scale real-world industrial spreadsheet corpora[2] reveals that the bad practice of using *nested-IF* expressions is suprisingly common among end users: 30.04% of the worksheets containing *IF* also contain *nested-IF*. If we denote the maximum nesting level inside a *nested-IF* expressions as *if-depth* [3], each spreadsheet includes on average 9 formulae with *if-depth* over 10, while the observed maximum *if-depth* is 64 with multiple instances. The surprising abuse of multiple nest-if statements suggests that end users may lack the consciousness, essential knowledge, or skills to tackle this problem. Automatic support is in great demand.

To tackle this problem, we propose an automated approach to systematically refactoring formulae. The general idea is two-fold. First, there often exists logic redundancy across different condition paths within a *nested-IF*. Reduction of the redundant logic can remove useless parts and simplify the *nested-IF* formula. Second, we realized that in many occasions end users use *nested-IF* functions to achieve some complex bug specific functionality. Thus, some higher-level semantics are often fragmented into hierarchical combinations of *IF* conditions in a *nested-IF*. Reassembling the fragmented syntax from corresponding *IF*-subtrees into built-in functions can shorten the *nested-IF* formula. To analyze and refactor both redundant logic and fragmented syntax, our approach leverages and works on the AST (Abstract Syntax Tree) structure as intermediate representation of *nested-IF* formulae.

The evaluation is conducted on two large spreadsheet corpora, with over 80,000 real-world spreadsheets and over 28 million *nested-IF* formulae. The experimental results lead to the following two key takeaways. First, our approach is generally applicable - over 90% of the *nested-IF* formulae can be refactored. Second, our approach is effective - the *nested-IF* functions in most formulae are completely reduced or transformed with a new *if-depth* of 1.

The main contributions of this paper are shown as follows.

1) The first statistical investigation on the current usage of *nested-IF* formulae in real-world industry-level spreadsheets. We present detailed statistics of *nested-IF* formulae against two corpora, with over 80,000 real-world spreadsheets. We find that *nested-IF* formulae are surprisingly commonly used among end users.

2) The first automated approach to identifying and refactoring *nested-IF* formulae. The approach has high coverage in reducing of smells of *nested-IF* formulae in spreadsheets.

3) A comprehensive evaluation of the proposed automated approach. We evaluated the correctness, applicability, and effectiveness of the approach.

[2] We refer spreadsheet as a file consisting of one or multiple worksheets [11].

[3] E.g, $IF(IF(L1 >= F\$5, L1), IF(L1 <= F\$6, L1, “”), “”)$ has an *if-depth* of 2.

2 COMMON USAGE OF NEST-IF FORMULAE

Nest-if formulae are well known spreadsheet smells, but it remains unknown how commonly they are adopted among end users. In this paper, we conduct the first exploration about the usage status of *nested-IF* formulae.

'The investigation is based on two large-scale spreadsheet corpora. The first corpus is a spreadsheet repository collected by Microsoft, named *MS corpus* in this paper, with over 68,000 real-world spreadsheets (excluding those with technical complications as obstacles for interaction-free processing, e.g., password protected, external reference embedded requiring trust confirmation). The Second corpus is *Enron Spreadsheet Corpus*, introduced by Hermans and Murphy-Hill [11], containing over 15,000 real-world spreadsheets. It is open-source and widely adopted in research [14–16].

We choose these two corpora because of their very large scale: the number of spreadsheets (68,075/15,770) is larger than the other corpora that have been used in previous research (the EUSES Spreadsheet Corpus [17] with 4,037 spreadsheets and the Hawaii Kooker Corpus [18] with 74 spreadsheets). In particular, the MS corpus has high diversity, containing data from various companies across multiple domains. Such large scale and diversity make them more representative of the generalized usage status of spreadsheet formulae. The detailed information is listed in Table 1.

Table 1: Details of the MS corpus and Enron corpus

	Corpus	
	MS	Enron
Average size of spreadsheets	1,211.3 KB	113.4 KB
# spreadsheets	68,075	15,770
# worksheets	149,170	79,983
# spreadsheets with formulae	37,109	9,120
# spreadsheets with *IF* functions	14,425	2,020
# *IF* functions	138,085,568	3,420,790

Based on these two corpora, we investigate the number of *nested-IF* formulae with different *if-depth*. We scan each formula in every spreadsheet, and checks if it contains the *IF* function. If so, we then check the *if-depth* of the formula. If the *if-depth* is bigger than 1, the formula is a nest-if formula. Finally, we count the number of formulae with different depth ranges.

The results are shown in Table 2. The table indicates a **surprisingly heavy usage of *nested-IF* formulae**. For example, for the MS corpus, over 20% formulae with IF function also contain *nested-IF* functions. Over 12% formulae have an *if-depth* of over 5. Over 75 thousand formulae even have an *if-depth* of more than 15. In addition, we found that 30.04% of the worksheets containing *IF* also contain *nested-IF* (not listed in the table). Each spreadsheet includes on average 9 formulae with *if-depth* over 10. What's worse, the observed maximum *if-depth* is 64 with multiple instances.

The *nested-IF* formulae are less common in the Enron corpus than in the MS corpus, but still over 15% formulae with IF function also contain *nested-IF* functions, with over 5,032 formulae containing *nested-IF* functions.

The heavy usage of *nested-IF* formulae indicates that end-users usually lack the awareness or enough knowledge to avoid the smell. As a result, the readability, maintainability, and correctness of spreadsheet may be seriously affected [8]. Automated approaches are thereby in great need to help tackle this problem.

Table 2: Usage status of *nested-IF* formulae

MS corpus	Formula Number
# formulae with IF function	138,085,568
# formulae with *nested-IF* function	27,689,299
if-depth in range (1,5]	24,250,194
if-depth in range (5,10]	2,815,521
if-depth in range (10,15]	548,129
if-depth in range (15,65]	75,455
Enron Corpus	**Formula Number**
# formulae with IF function	3,420,790
# formulae with *nested-IF* function	532,241
if-depth in range (1,5]	527,209
if-depth in range (5,10]	5,032

3 AUTOMATIC REFACTORING

3.1 Overview

Our approach identifies optimizable *nested-IF* expressions and performs refactoring by analyzing the AST structure of each formula to replace basic-level and counter-intuitive syntax with non-redundant and high-level syntax. The approach contains three basic steps: target identification, redundancy removal, and syntax reassembling. In this section, we first present a high-level overview of the 3-step approach, then introduce the details of the two key algorithms for redundancy removal and syntax reassembling, respectively.

Step1: Target identification. First, we need to identify whether a formula has *nested-IF* functions. We achieve this by parsing each formula and generating its AST. AST is a tree representation of the abstract syntactic structure of source code written in a programming language. In spreadsheet related research, AST is usually adopted to indicate formula complexity [16]. The larger depth (height) of the AST, the higher complexity of the formula. The major rationale behind using AST is the desirable structural mapping between AST and *nested-IF* as follows. An *IF* function typically contains three parts: 1) condition, 2) true-branch expression, and 3) false-branch expression. Therefore, the ASTs of *nested-IF* expressions are binary trees, with the true- and false-branch expressions being the two child-nodes of the condition node. Consequently, with AST, it is easy to locate *nested-IF* in a formula as well as convenient to conduct further analysis based on the tree structure.

Along each path of AST, we record the number of *IF* functions, and regard the largest one across all paths as the *if-depth* of the formula. A *nested-IF* is identified in a formula when its *if-depth* is greater than 1, and will be passed to the subsequent steps for refactoring analysis. Otherwise, if the *if-depth* equals 0 (i.e., no *IF* in this formula) or 1 (i.e., no *nested-IF* in this formula), our algorithm will bypass the formula directly.

Step2: Redundancy removal. An *IF* expression can essentially be mapped to an if-else branching statement in professional programming. Once the condition on some node remains deterministic due to its preceding evaluation at some ancestor node on AST, it will become a redundant condition and one of its child branches must be dead code. Such redundant conditions are spreadsheet smells that require removal, since they introduce unnecessary complications to the spreadsheet data and make formulae more complex. We conduct such redundancy removal first, because its existence may also obscure the AST structure from well understood patterns and thus put negative impact on syntax reassembling. More details of this step can be found in Section 3.2.

Step3: Syntax reassembling. We have observed that single and higher-level semantics are often fragmented by end user into lower-level syntax pieces with *nested-IF*s. We then manually checked if there are built-in functions predefined in spreadsheets with higher-level syntax but identical semantics. The goal of this step is to conduct reverse inference against such a smell, i.e., to recognize and reassemble such semantic-fragmented AST regions into their more concise forms via pattern matching and replacement. More details of this step can be found in Section 3.3.

3.2 Redundancy Removal

In this section, we introduce how we identify and remove redundant conditions in a *nested-IF* formula (Step 2). The procedure is presented with the help of an example flow in Figure 1.

1) *Nested-IF* expression extraction. First, we extract outmost *nested-IF* expressions from each formula. By outmost we mean the highest hierarchy in a nested branching logic or on an AST. For example, for formula $SUM(IF(C1, V1, V2), IF(C2, V3, IF(!C2, V4, V5)), IF(C3, V5, IF(IF(C4, V6, V7))))$, there are two target *nested-IF* expressions: $IF(C2, V3, IF(!C2, V4, V5))$ and $IF(C3, V5, IF(IF(C4, V6, V7)))$.

2) Branch collection. Based on the AST of each extracted *nested-IF*, we create a dictionary *dicConBranch* as the key structure to help detect and remove redundant logic. As shown in Figure 1, for each entry in the dictionary, its key is the condition of an AST node such as $C1$ or $C2$; the *dBranchList* value stores a tuple of two AST sub-trees corresponding to true and false branches respectively. In addition, each entry also has a *nBranchList* value for the negation of the key condition such as $!C1$, and stores the tuple of true and false branches accordingly. The dictionary is constructed by visiting each condition node on the AST. When the same condition (or negation) is hit for multiple times, the AST sub-tree tuples at each hitting site are appended to the *dBranchList* (or *nBranchList*).

3) Redundancy identification and removal. Intuitively, if any entry stores more than 1 tuple in *dBranchList* and *nBranchList* collectively, it indicates existence of redundant branches on the AST about the condition at key. We iterate such inspection against *dicConBranch* to detect and remove redundancies. Each detected redundancy site corresponds to one redundant *IF* expression that can be replaced with either the true branch (the condition is deterministic as true) or the false branch (the condition is deterministic as false). Thus, under each situation, we generate the redundant *IF* expression according to the condition and its branch list and make replacement.

3.3 Syntax Reassembling

After removing redundancies, if the resultant formula still contains *nested-IF* expressions, in this third step we further analyze the AST to detect and reassemble fragmented semantics into built-in functions.

We summarize these functions based on our case analysis. First, we sampled around 100 (0.1%) spreadsheets from the MS corpus. Second, we manually analyzed the *nested-IF* expressions one by one and summarized their semantics. Third, we examined the predefined functions in spreadsheets[4] to check if some of them own similar semantics as those we summarized.

We finally matched seven pre-defined functions from our sampled dataset, as listed in Table 3. As of the composing of this paper, there might be other function candidates that remain out of our knowledge. Nonetheless, our proposed algorithm framework should be extensible for easy incorporation of new patterns.

Correspondly, we have identified seven categories of patterns corresponding to seven types of built-in spreadsheet functions. The basic patterns (with *if-depth* of 5 in all examples) are illustrated in Figure 2. Based on specific structures of each pattern, their pattern matching algorithms share the preceding general procedure and differ in minor details. Additionally, we find another pattern that does not match any function, but can also be transformed accordingly to remove nested IF. We call this pattern the "USELESS" pattern. For example, expression $IF(A = B, A, B)$ actually equals A or B. We put the checking order of this patter just before the IFS pattern.

For ease of presentation, we unify the condition redundancy, the USELESS pattern, and the 7 functions all as "patterns". More details of each pattern can be found on our homepage[5].

We then conduct iterative pattern-matching and replacement. For each remaining *nested-IF* after step 2, we further construct a *threePartList* as the key structure to facilitate pattern matching. Each *threePartList* consists of three lists for condition, true branch, and false branch, respectively. For example, for expression $IF(C1, IF(C2, IF(C3, V1, V2), V2), V2)$, the condition part is $[C1, C2, C3]$, the true branch part is $[IF(C2, IF(C3, V1, V2), V2), IF(C3, V1, V2), V1]$, and the false part is $[V2, V2, V2]$.

Subsequently, based on *threePartList*, we infer the semantic of the *IF* expression through pattern matching. If we could find matched patterns, we transform the formula using the corresponding function, and replace the *nested-IF* expression with the transformed one. Following the order introduced in Table 3, we probe each pattern in sequence. Once a pattern is matched, the probe jumps to the next iteration from the first pattern again. This iteration terminates with zero pattern match. Note that the patterns *CHOOSE*/*MATCH*/*LOOKUP* have higher priority than the pattern *IFS* during the matching, because they are more comprehensible and enable more concise expressions. In the future, we may consider to provide all alternative refactoring recommendations for end users to choose from.

4 EVALUATION

4.1 Research Questions

In this paper, we investigate the following three research questions.

RQ1: Is our refactoring correct? This question aims to check the correctness of our approach.

[4] Most mainstream spreadsheet tools such as Excel and Google Sheets support these functions.

[5] https://github.com/sei-pku/nestif

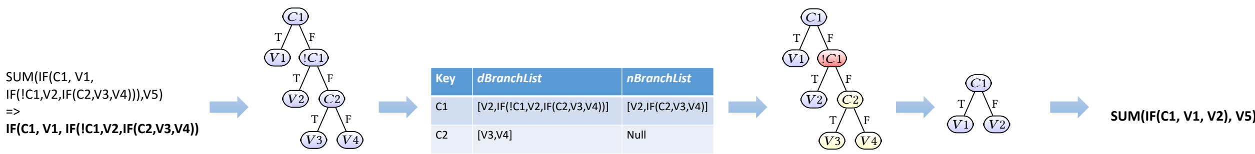

Figure 1: The process of redundancy removal.

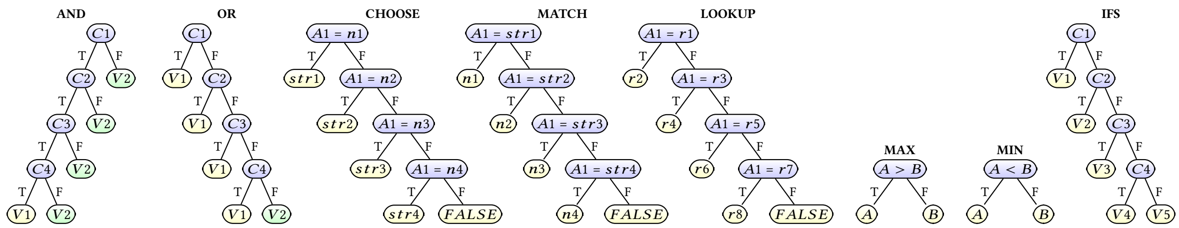

Figure 2: Typical AST of function AND,OR,CHOOSE,MATCH,LOOKUP, MAX,MIN, and IFS. $stri$ **represents a string;** ni **represents a number;** ri **represents a reference** ($0 < i < 5$)**.**

Table 3: The alternative functions we identified

Name	Explanation	Transformation Examples
AND	Returns TRUE if all of the arguments evaluate to TRUE.	$IF(C1, IF(C2, IF(C3, V1, V2), V2), V2) \rightarrow IF(AND(C1, C2, C3), V1, V2)$
OR	Returns TRUE if any argument evaluates to TRUE.	$IF(C1, V1, IF(C2, V1, IF(C3, V1, V2))) \rightarrow IF(OR(C1, C2, C3), V1, V2)$
CHOOSE	Returns a value from a list using a given position or index.	$IF(A1 = 1, str1, IF(A1 = 2, str2, IF(A1 = 3, str3))) \rightarrow CHOOSE(A1, str1, str2, str3)$
MATCH	Returns a number representing a position in an array.	$IF(A1 = str1, 1, IF(A1 = str2, 2, IF(A1 = str3, 3))) \rightarrow MATCH(A1, \{str1, str2, str3\}, 0)$
LOOKUP	Perform a vertical/horizontal lookup (corresponding to function VLOOKUP and HLOOKUP) by searching for a value in the first column/row of a table and returning the value in the same row/column in the $index$ position.	$IF(A1 = C1, D1, IF(A1 = C2, D2, IF(A1 = C3, D3, IF(A1 = C4, D4)))) \rightarrow VLOOKUP(A1, C1 : D4, 2, FALSE)$
MAX/MIN	Return the largest/smallest value.	$IF(A > B, A, B) \rightarrow MAX(A, B)$; $IF(A < B, A, B) \rightarrow MIN(A, B)$
IFS	Run multiple tests and return a value corresponding to the first TRUE result.	$IF(C1, V1, IF(C2, V2, IF(C3, V3, IF(C4, V4)))) \rightarrow IFS(C1, V1, C2, V2, C3, V3, C4, V4)$

RQ2: What is the general performance of our approach in terms of refactor coverage? This question aims to check how many *nested-IF* formulae our approach could handle.

RQ3: What is the general performance of our approach in terms of refactor effectiveness? This question aims to check how much *if-depth* our approach could decrease.

All the experiments are conducted on the two spreadsheet corpora, the MS corpus and the Enron corpus, introduced in Section 2. Note that inside one spreadsheet many formulae may be created by dragging one formula down or to the right to repeat its calculation. As in previous work [11, 14, 15], we remove these formulae by clustering the formulae based on their R1C1 notation[6]. We then pick one formula from each cluster to form the new formula set. We call this new set the "Unique Set" and the original set the "Total Set". The experimental results are presented on these two types of formula sets for each corpus respectively.

Next, we present the experimental setup as well as the results to answer each of the research questions.

[6] The R1C1 notation will stay the same even if the formula is dragged down or right.

4.2 RQ1: Correctness

To answer the first research question, we conduct manual inspection and formula replacement to check the correctness of formula refactoring.

For manual inspection, considering that there are over 28 million formulae and it is impossible to check all the refactoring one by one, we randomly select 2000 formula pairs $< F_o, F_r >$ (F_o represents the original formula, F_r represents the refactored formula) as the check targets. The first three authors then check each pair and record their judgements.

For formula value comparison, we scan all Excel files and replace the original *nested-IF* formulae with the refactored ones. For each formula pair $< F_o, F_r >$, we get a responding value pair $< V_o, V_r >$. We thus record whether V_o equals to V_r. To automatically achieve the above process, we use *ClosedXML*, which is a powerful .NET library enabling users to create and modify Excel files.

The checking results indicate a 100% correctness of our approach. This result reveals the reliability of the refactoring results. We achieve highly correct refactoring because of the strict matching of the *nested-IF* patterns. For those *nested-IF* formulae which does

Table 4: Results of refactor coverage.

Corpus	Formula Set	Original	Refactored	Coverage
MS	Total	27,689,299	27,645,688	99.84%
	Unique	19,260,407	19,243,407	99.91%
Enron	Total	533,023	515,958	96.80%
	Unique	231,978	215,694	92.98%

not match our patterns, we skip them and regard them as those uncovered by our approach.

4.3 RQ2: Coverage

For the original total set of *nested-IF* formulae O_{total} and original unique set O_{unique}, we conduct automatic refactoring following the refactoring procedure introduced in Section 3. Correspondingly, we get the refactored set R_{total} (out of O_{total}) and R_{unique} (out of O_{unique}). The refactor coverage can then be calculated as $\frac{\#R_{total}}{\#O_{total}} * 100\%$ and $\frac{\#R_{unique}}{\#O_{unique}} * 100\%$ (# represents the number).

Table 4 presents the results of refactor coverage. Column "Original" lists the number of original *nested-IF* formulae; Column "Refactored" lists the number of *nested-IF* formulae that our approach could refactor; Column "Coverage" represents the proportion of refactored formula. From this table, our approach is able to handle most of the *nested-IF* formulae for both corpora, with a refactor coverage of over 90% on both the Total set and the Unique set.

There are around 33,000 *nested-IF* formulae that cannot be automatically refactored. We manually checked a sample of them and realized that our approach could not deal with two types of *nested-IF* formulae. In the first type, the condition part of the outmost IF expression contains another IF expression and does not match our patterns even if being treated as a whole, such as $IF(AND(IFsubexpression1, IFsubexpression2) = TRUE, value1, value2)$. In the second type, although the inner *IF* expression lies in the branches of the outer expression, it is wrapped with other non-IF functions, and thus the AST is quite complex, such as $IF(Condition, SUM(IFsubexpression1, IFsubexpression2), value)$.

The refactor coverage of Enron corpus is slightly lower than the MS corpus. This is because Enron corpus has larger proportion of formulae with the tough patterns we mentioned above.

4.4 RQ3: Effectiveness

We present the depth reduce performance of our approach from two aspects: the relative *if-depth* reduction (the depth reduction rate) and the final *if-depth* of formulae after our refactoring.

4.4.1 Relative Depth Reduce. We present the results of relative depth reduction during the refactoring: $DepReduce_{ratio} = DepReduce_{num}/DepOriginal$, the ratio of reduced depth against the original depth. For ease of presentation, we divide $DepReduce_{ratio}$ into four ranges: (0%, 25%] [7], (25%, 50%], (50%, 75%], and (75%, 100%]. Due to space limit, we only present the disctribution of each range for the MS corpus, as shown in Figure 3.

From the figure, different $DepReduce_{ratio}$ have different distributions. Most refactoring falls into Range (25%, 50%] and Range (50%, 75%] on both corpora, while no refactoring falls into Range (0%, 25%].

To conclude, from the two charts, in our approach most refactorings could reduce by more than a half of the *if-depth*, indicating that our approach is effective.

4.4.2 Depth After Refactoring. Except for the relative depth reduction results, we check whether the refactored formulae still have large *if-depth* by investigating the new *if-depth* dep_r of each refactored formula F_r. The results are shown in Table 5.

From the table, most of the refactoring yields a new *if-depth* of 0 or 1[8]. This observation indicates that our approach is able to completely remove the *nested-IF* functions for most formulae.

Two reasons contribute to this performance in reducing *if-depth*. First, some of our patterns, if matched perfectly, could remove the *nested-IF* expressions completely, such as the CHOOSE pattern and the MATCH pattern. Some other patterns may most probably keep just one if expression. For example, when we use the AND pattern to deal with formula $IF(A1, IF(A2, IF(A3, V1, V2), V2), V2)$, the refactored one $IF(AND(A1, A2, A3), V1, V2)$ has an *if-depth* of 1. Second, our approach repeats the process of refactoring until no *nested-IF* expressions could be handled. This repeat ensures the thoroughgoing refactoring.

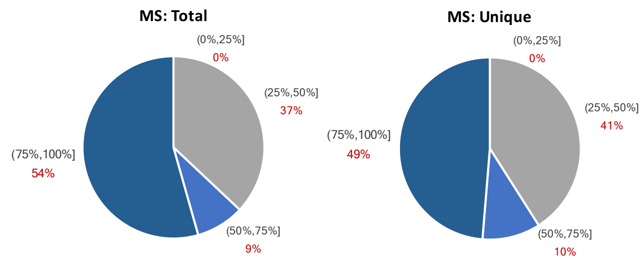

Figure 3: Distribution of refactoring with different depth reduction ratios.

Table 5: Number of formulae with different new depth

Corpus	New Depth	Total	Unique
MS	0	13,906,460 (50.30%)	8,723,082 (45.33%)
	1	13,717,158 (49.62%)	10,498,258 (54.56%)
	2	22,070 (0.08%)	22,067 (0.11%)
Enron	0	224,915 (43.59%)	72,852 (33.78%)
	1	291,043 (56.41%)	142,842 (66.22%)

5 RELATED WORK

The research work most related to ours includes smell detection and refactoring in spreadsheets. The former is related to the motivation of this paper: why *nested-IF* formulae are bad smells. The latter is related to the approach of this paper: how to refactor spreadsheets to reduce smells. We next introduce these two aspects one by one.

5.1 Smell Detection

Same as code smells [19], spreadsheets smells refer to some characteristics that may cause problems. Smells have different levels:

[7] $0\% < DepReduce_{ratio} <= 25\%$

[8] *if-depth* of 0 and 1 are equally effective in relieving nested IF smells, because either of them avoid the smell completely.

formula-level, cell level, and structural level. We mainly introduce the formula-level ones.

Abreu et.al. [5] combines 15 smells to indicate potential faults. They treat conditional complexity as one of the key smells. The results indicate that this smell only can detect 6 spreadsheet faults.

Hermans et.al. [4] regard conditional complexity as one of the five smells, because even in traditional professional programming, conditional complexity is a threat to code readability. However, according to their results derived from EUSES, on average each spreadsheet only has 3 formulae containing at least one condition, while from our corpus, we find that on average each spreadsheet has 1,193 formulae containing conditions; from the corpus of Enron, the number is 217. The reason for this huge difference may be that EUSES contains a lot of toy spreadsheets created by users who rarely use spreadsheet formulae.

Hermans et.al. [4] also mention that end users already know the bad effects of conditional complexity. Our survey results confirm this statement: around half of the participants think that formulae with high conditional complexity are more complex and error-prone; 70.55% think that they are harder to understand.

Another work of Hermans et.al. [6] present an overview of software engineering approaches applied to spreadsheets. They claim that most spreadsheets contain formulae with multiple IF conditions, which is an obvious spreadsheet smell.

5.2 Formula Refactoring

Badame and Dig [12] are the first to propose refactoring in the spreadsheet domain. A tool – ReeBook – is presented, with which seven refactoring patterns are presented. These seven patterns target at different smells. For example, pattern *MAKE CELL CONSTANT* aims to make formulae less error prone and more readable by adding the $ symbol. However, their approach is disperse and can handle only simple formulae. For example, one of their refactoring patterns is called "REPLACE AWKWARD FORMULA", which only focus on the SUM function (e.g., replace $B5 + C5 + D5 + E5$ with $SUM(B5 : E5)$). They evaluate their approach on EUSES corpus and find that their refactoring can be applied to many formulae. However, they only present the number of formulae that are "potential candidates" for each pattern, while not presenting the actual number of successfully refactored formulae. Thus, the refactor coverage and effectiveness are unknown.

Hermans et.al. [4] defined different refactoring according to their smells. The results indicate that their refactoring approach is able to relieve the smells of 87% formulae. However, their approach does not support automated refactoring.

Later on, Hermans and Dig [13] combine the two approaches above and present BumbleBee, which is a refactoring tool allowing a formula to be refactored based on the defined transformation rules. Several patterns such as MAXMIN and OR are also mentioned in the paper. However, the formula can be refactored only when the transformation rule is defined, while according to our survey, only 20.99% of participants may have the knowledge of defining transformation rules. The work of Hoepelman [20] expand this work and introduces more refactoring support.

To sum up, currently several works aim to tackle the challenges brought by spreadsheet smells, while no automatic and high-coverage refactoring approach is available. We propose to systematically tackling the *nested-IF* formulae refactoring problem, which is able to handle most of the formulae with high depth-reduce effectiveness.

6 CONCLUSION

We proposed an investigation into the usage status of *nested-IF* formulae and found that *nested-IF* formulae are surprisingly heavily used among end users. Accordingly, we presented a spreadsheet formula refactoring approach to automatically relieving the smells of *nested-IF* functions. Evaluation on two very large real world spreadsheet corpora indicates that the refactor effectiveness is impressive: most of the *nested-IF* formulae can be refactored.

In the future, we plan to make our approach a spreadsheet plug-in. When an end user finishes writing a formula with *nested-IF* functions, the plug-in may identify whether the formula can be refactored. If so, it alerts that these nested IFs are bad smells, and provides refactor suggestions.

REFERENCES

[1] wiki. End user development. https://en.wikipedia.org/wiki/End-user_development, 2015.
[2] Margaret M Burnett and Christopher Scaffidi. 10. end-user development.
[3] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575, 2015.
[4] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575, Apr 2015.
[5] R. Abreu, J. Cunha, J. P. Fernandes, P. Martins, A. Perez, and J. Saraiva. Smelling faults in spreadsheets. In *Proc. ICSME*, pages 111–120, Sept 2014.
[6] F. Hermans, B. Jansen, S. Roy, E. Aivaloglou, A. Swidan, and D. Hoepelman. Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets. In *Proc. SANER*, volume 5, pages 56–65, March 2016.
[7] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *Proc. ICSE*, pages 403–414. IEEE Press, 2015.
[8] office. IF function – nested formulas and avoiding pitfalls . https://support.office.com/en-us/article/IF-function-%e2%80%93-nested-formulas-and-avoiding-pitfalls-0b22ff44-f149-44ba-aeb5-4ef99da241c8?ui=en-US&rs=en-US&ad=US, 2015.
[9] reddit. Is it a good or bad practice reducing nested if statements. https://www.reddit.com/r/csharp/comments/33puzj/is_it_a_good_or_bad_practice_reducing_nested_if/, 2015.
[10] reddit. Never use nested IFs again. https://www.reddit.com/r/excel/comments/2slys1/never_use_nested_ifs_again/, 2015.
[11] Felienne Hermans and Emerson Murphy-Hill. Enron's spreadsheets and related emails: A dataset and analysis. In *Proc. ICSE*, pages 7–16. IEEE Press, 2015.
[12] Sandro Badame and Danny Dig. Refactoring meets spreadsheet formulas. In *Proc. ICSM*, pages 399–409. IEEE, 2012.
[13] Felienne Hermans and Danny Dig. Bumblebee: a refactoring environment for spreadsheet formulas. In *Proc. ICSE*, pages 747–750. ACM, 2014.
[14] T. Schmitz and D. Jannach. Finding errors in the enron spreadsheet corpus. In *Proc. VL/HCC*, pages 157–161, 2016.
[15] Bas Jansen. Enron versus euses: A comparison of two spreadsheet corpora. *arXiv preprint arXiv:1503.04055*, 2015.
[16] Thomas Reschenhofer, Bernhard Waltl, Klym Shumaiev, and Florian Matthes. A conceptual model for measuring the complexity of spreadsheets. *arXiv preprint arXiv:1704.01147*, 2017.
[17] Marc Fisher and Gregg Rothermel. The euses spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM, 2005.
[18] Salvatore Aurigemma and Raymond R Panko. The detection of human spreadsheet errors by humans versus inspection (auditing) software. *arXiv preprint arXiv:1009.2785*, 2010.
[19] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.
[20] DJ Hoepelman. Tool-assisted spreadsheet refactoring and parsing spreadsheet formulas. 2015.