ColDeco: An End User Spreadsheet Inspection Tool for AI-Generated Code

Kasra Ferdowsi *†, Jack Williams ‡¶, Ian Drosos ‡¶, Andrew D. Gordon ‡¶,
Carina Negreanu ‡¶, Nadia Polikarpova *, Advait Sarkar ‡¶, and Benjamin Zorn §¶
* *UC San Diego, La Jolla, USA*; ‡ *Microsoft Research, Cambridge, UK*; § *Microsoft Research, Redmond, USA*
* {*kferdows, npolikarpova*}*@ucsd.edu*; ¶ {jack.williams, t-iandrosos, adg, cnegreanu, advait, ben.zorn}@microsoft.com

***Abstract*—Code-generating large language models (LLMs) are transforming programming. Their capability to generate multi-step solutions provides even non-programmers a mechanism to harness the power of coding. Non-programmers often use spreadsheets to manage tabular data, as they offer an intuitive understanding of data manipulation and formula outcomes. Considering that LLMs can generate complex, potentially incorrect code, our focus is on enabling user trust in the accuracy of LLM-generated code. We present ColDeco, the first end-user inspection tool for comprehending code produced by LLMs for tabular data tasks. ColDeco integrates two new features for inspection with a grid-based interface. First, users can *decompose* a generated solution into intermediate *helper columns* to understand how the problem is solved step by step. Second, users can interact with a filtered table of *summary rows*, which highlight interesting cases in the program. We evaluate our tool using a within-subjects user study (n=24) where participants are asked to verify the correctness of programs generated by an LLM. We found that while all features are independently useful, participants preferred them in combination. Users especially noted the usefulness of helper columns, but wanted more transparency in how summary rows are generated to assist with understanding and trusting them. Users also highlighted the application of ColDeco in collaborative settings for explaining and understanding existing formulas.**

I. Introduction

In the past two years, large language models (LLMs) [1], [2], [3], [4] have emerged as a practical tool for synthesizing code from natural language. Their commercialization in assistive features such as GitHub Copilot [5], [6] is transforming programming for professional programmers. Still, these tools rely on the expertise of professional programmers to evaluate the (often incorrect) output of the model. The promised value in these tools is only realised through the interactive evaluation and repair of such generated fragments by an expert programmer. However, due to their potential for empowerment, the question arises: how do we design tools specifically to help less skilled users understand and debug AI-generated programs?

A. Background: AI for end-user programming

Without a formal education in programming, most individuals are unfamiliar with and have difficulties with the abstract concepts such as variables, functions, parameters, etc., that make up the *notional machine* with which a programmer understands how a program functions [7]. Spreadsheets provide mechanisms for individuals unfamiliar with these concepts to have a direct and concrete understanding of their data and transformations on it (such as sums, etc.) due, among other things, to their simplified models of control flow and naming [8], [9], [10], [11]. Spreadsheets can provide a graduated experience that allows a range of individuals, from non-programmers to professional programmers, to solve problems using the tools with which they are most comfortable [12], [13]. Our research investigates the application of AI in synthesizing code solutions for *end-user programmers*, people like the many spreadsheet users who need to code as part of their work but who are not professional programmers [14].

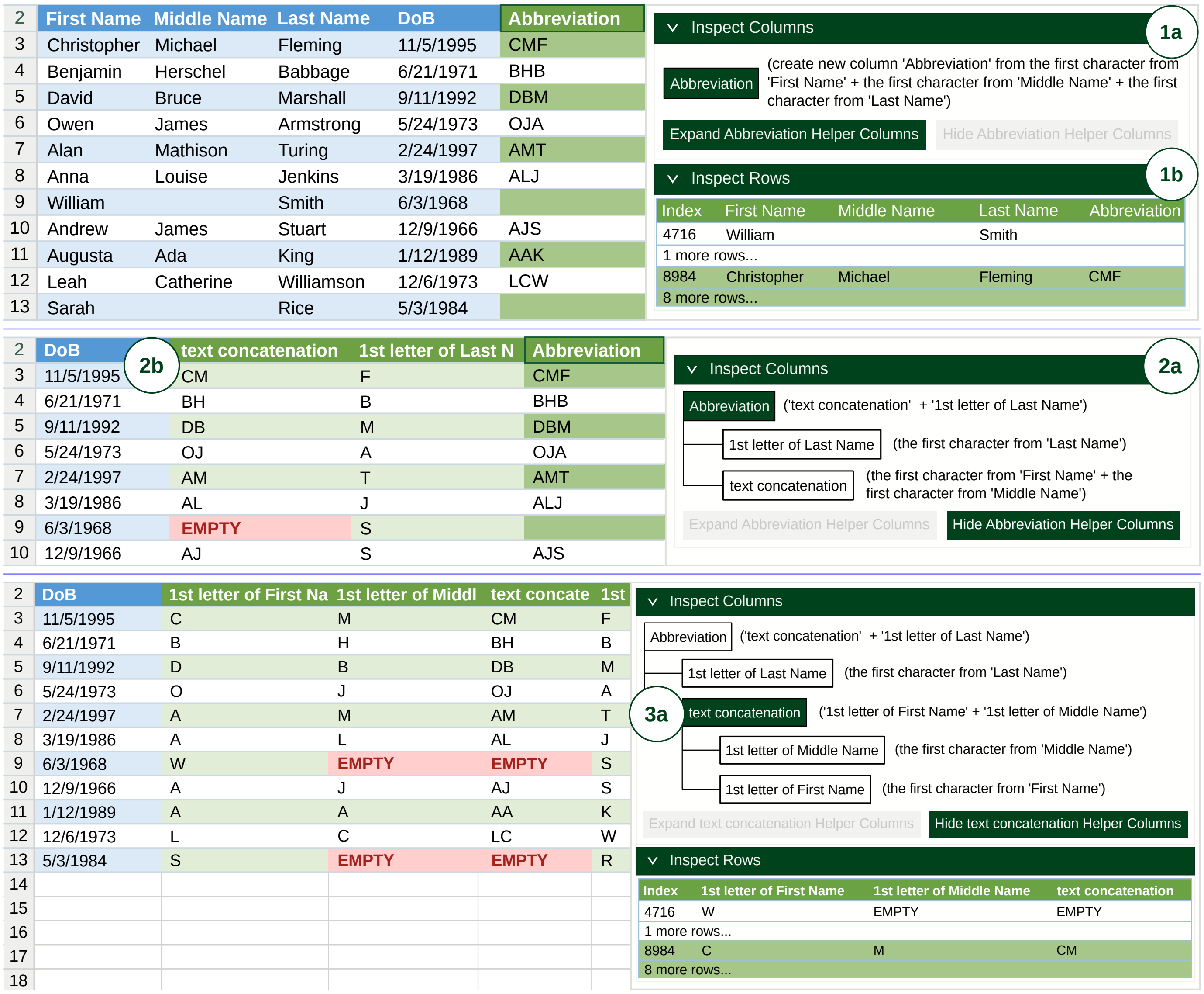
There have been many successful attempts to empower spreadsheet users to define computations without having to learn a formal programming language. The most widely deployed example is FlashFill, a programming-by-example string transformation feature that ships commercially [15]. More recently, commercial spreadsheets have introduced AI-powered features for data analytics by synthesizing pivot tables or charts via automatic recommenders or natural language queries [16], [17]. These tools give rise to a new challenge: *How do we enable end users to evaluate the correctness of machine-generated computations without inspecting the underlying code?*

The only recent work we are aware of that targets this challenge for AI-powered spreadsheet programming is *grounded abstraction matching* (GAM), a new interaction style, which explains AI-generated code to end users in natural language [18]. While GAM helps the user confirm that the model's understanding of the problem matches their intent, it does not necessarily help them discover and diagnose errors, when the intent leads to unexpected behavior on the given data.

B. This paper: ColDeco, an inspection tool for end users

To assist end-user programmers with discovering and diagnosing errors in AI-generated code, ColDeco augments natural-language descriptions in the style of GAM with two complementary features, illustrated in Fig. 1.

† Work done while at Microsoft Research.



	First Name	Middle Name	Last Name	DoB	Abbreviation
3	Christopher	Michael	Fleming	11/5/1995	CMF
4	Benjamin	Herschel	Babbage	6/21/1971	BHB
5	David	Bruce	Marshall	9/11/1992	DBM
6	Owen	James	Armstrong	5/24/1973	OJA
7	Alan	Mathison	Turing	2/24/1997	AMT
8	Anna	Louise	Jenkins	3/19/1986	ALJ
9	William		Smith	6/3/1968	
10	Andrew	James	Stuart	12/9/1966	AJS
11	Augusta	Ada	King	1/12/1989	AAK
12	Leah	Catherine	Williamson	12/6/1973	LCW
13	Sarah		Rice	5/3/1984	

Inspect Columns

Abbreviation (create new column 'Abbreviation' from the first character from 'First Name' + the first character from 'Middle Name' + the first character from 'Last Name')

Expand Abbreviation Helper Columns · Hide Abbreviation Helper Columns

Inspect Rows

Index	First Name	Middle Name	Last Name	Abbreviation
4716	William		Smith	
1 more rows...				
8984	Christopher	Michael	Fleming	CMF
8 more rows...				

	DoB	text concatenation	1st letter of Last N	Abbreviation
3	11/5/1995	CM	F	CMF
4	6/21/1971	BH	B	BHB
5	9/11/1992	DB	M	DBM
6	5/24/1973	OJ	A	OJA
7	2/24/1997	AM	T	AMT
8	3/19/1986	AL	J	ALJ
9	6/3/1968	EMPTY	S	
10	12/9/1966	AJ	S	AJS

Inspect Columns

Abbreviation ('text concatenation' + '1st letter of Last Name')

1st letter of Last Name (the first character from 'Last Name')

text concatenation (the first character from 'First Name' + the first character from 'Middle Name')

Expand Abbreviation Helper Columns · Hide Abbreviation Helper Columns

	DoB	1st letter of First Na	1st letter of Middl	text concate	1st
3	11/5/1995	C	M	CM	F
4	6/21/1971	B	H	BH	B
5	9/11/1992	D	B	DB	M
6	5/24/1973	O	J	OJ	A
7	2/24/1997	A	M	AM	T
8	3/19/1986	A	L	AL	J
9	6/3/1968	W	EMPTY	EMPTY	S
10	12/9/1966	A	J	AJ	S
11	1/12/1989	A	A	AA	K
12	12/6/1973	L	C	LC	W
13	5/3/1984	S	EMPTY	EMPTY	R
14					
15					
16					
17					
18					

Inspect Columns

Abbreviation ('text concatenation' + '1st letter of Last Name')

1st letter of Last Name (the first character from 'Last Name')

text concatenation ('1st letter of First Name' + '1st letter of Middle Name')

1st letter of Middle Name (the first character from 'Middle Name')

1st letter of First Name (the first character from 'First Name')

Expand text concatenation Helper Columns · Hide text concatenation Helper Columns

Inspect Rows

Index	1st letter of First Name	1st letter of Middle Name	text concatenation
4716	W	EMPTY	EMPTY
1 more rows...			
8984	C	M	CM
8 more rows...			

Fig. 1: Initial view (top). Decomposing "Abbreviation" (middle). Decomposing "text concatenation" (bottom).

First, users can *decompose* a generated solution into intermediate helper columns [19] to understand how the problem is solved step by step (see Sec. II-D). Creating helper columns is a common spreadsheet practice for manually simplifying complex formulas. By automating helper column creation, we encourage users to explore the concrete impact of different parts of the code on their data.

Second, users can view a filtered table of *summary rows*, which highlights interesting cases in the program (see Sec. II-C). We introduce an analysis that captures all the unique behaviors that the code displays on the user's data, allowing them to quickly understand the effect of different paths through the code.

In general, a debugging experience involves both finding the source of the error (diagnosis) and fixing the error (repair). In ColDeco, we apply the familiar concepts of helper columns, grouping, and filtering in spreadsheets to assist the diagnostic aspect of debugging; we leave considerations of repair for future work (see Sec. VII).

Our paper makes the following contributions:

- We present ColDeco, the first end-user inspection tool for comprehending code produced by LLMs for tabular data tasks. ColDeco is an Excel add-in that provides three interrelated interactive components: (1) a decomposition of the solution into intermediate values which are added back to the table in the form of helper columns, (2) summary rows highlighting distinct behaviors in the solution, and (3) a description of the solution expressed in natural language.
- We evaluate this approach with a user study ($n = 24$) of spreadsheet users of varying expertise completing several debugging tasks. We find that users were able to inspect the results of code generated by an LLM and to find faults when this code was incorrect.

Participants regard all three components as useful. Spreadsheet users also indicated that ColDeco may provide value in collaboration by affording users the ability to explain code solutions to colleagues, and better understand complex formulas themselves. We received design feedback for future intelligent user interfaces for end-user debugging of LLMs.

II. ColDeco, by example

A data analyst named Kim is cleaning up a spreadsheet of responses to an online form. As a part of their task, they want to create a column of people's initials to use as a part of a unique identifier. Kim has experience with Excel, including using some formulas, but they are not very comfortable with string manipulation functions. So, rather than write a formula themself, they decide to use an LLM to generate the solution and verify it using ColDeco.

A. Generating the solution

Kim is using a spreadsheet environment that integrates a query box to create new columns using natural language. Kim writes their instruction: *Create a column "Abbreviation" concatenating the first character of each part of the name*, and the spreadsheet queries the LLM to generate code that will compute a new calculated column. The calculated column is added to the spreadsheet and ColDeco is automatically opened as a side pane (Fig. 1, top). By convention, additional calculated columns are formatted in green. A cursory glance at the first few rows in the table seems to confirm that the output is reasonable, but Kim would like to make sure the solution is correct, so they turn to ColDeco's inspection features to understand how the output was generated.

B. A first inspection

The ColDeco pane has two views, *Inspect Columns* (1a) and *Inspect Rows* (1b). The first view provides information about the calculated columns (here: "Abbreviation"), including a description of the calculation generated using natural language templates (1a). The description reveals that the calculation is taking the first letter from each of the "First Name", "Middle Name", and "Last Name" columns, and appears sensible to Kim.

The *Inspect Rows* view clusters the rows from the input table by their behavior in the calculation and depicts one example—a *summary row*—from each cluster (1b). Kim notices there are two clusters, and while the second cluster, with nine rows, behaves correctly, the first cluster, with two rows, is missing the output in the "Abbreviation" column. What initially appeared to be a correct solution is not producing the right output on some of the rows.

C. Analyzing the summary rows

Kim makes a further inspection of the summary rows to understand the problem. The *Inspect Rows* view (1b) only depicts a subset of the columns from the original table, including the output column ("Abbreviation") and the columns referenced by the calculation ("First Name", "Middle Name", and "Last Name"). Since this view only shows the columns that affect the output, Kim can more easily see that the empty output seems to be related to the empty middle names.

D. Inspecting the helper columns

To confirm their hypothesis, Kim looks back at the *Inspect Columns* view, and clicks *Expand Abbreviation Helper Columns* to break down the output into its helper columns (Fig. 1, middle).

This brings about the *tree view* (2a), which visualizes the structure of the computation. The tree view shows that the output is comprised of two helper columns, one combining the first letter of "First Name" and "Middle Name", and the other just getting the first letter of "Last Name". The names of helper columns are generated using natural language templates similarly to column descriptions. By looking at the values in the table (2b), Kim sees that the "1st letter of Last Name" column is correct for all rows, but for those without a middle name, the "text concatenation" column is showing a red `EMPTY`, indicating that something went wrong there. ColDeco automatically highlights cells that contain errors, where `EMPTY` is analogous to Excel's `#VALUE!`.

Investigating further, Kim selects the "text concatenation" column, and clicks *Expand* once more (Fig. 1, bottom). This creates two new helper columns, for the first letter of the first and middle names respectively. Both the *Inspect Columns* and *Inspect Rows* views are synchronized with the grid, such that selecting a column in one of them will update the other views. Kim selects the "text concatenation" column to update the summary rows (3a).

With the intermediate values visible, Kim sees that indeed the program correctly computes the first letter of the first and last names. But for rows without a middle name, computing the first letter fails, and that error propagates, causing the output to be empty as well.

III. Related Work

To the best of our knowledge, ColDeco is the first end-user inspection tool for debugging LLM-generated code for data-centric tasks. The only other work we are aware of with the goal of helping spreadsheet users harness the power of LLMs is *grounded abstraction matching* (GAM) [18]. GAM generates natural-language descriptions of LLM-generated programs, which enable end users to both confirm the model's understanding of their request and to iteratively refine the request if needed. While GAM and ColDeco have a similar target audience, and in fact, our tool incorporates natural-language descriptions similar to those in GAM, the main focus of ColDeco is on two novel mechanisms—helper columns and row summaries—that enable the user to concretely see the effect of the generated code on their data. Apart from GAM, the two most relevant lines of work are those on (1) inspecting synthesized code, and (2) debugging tools for spreadsheets.

A. Inspecting LLM-Generated Code

There is a growing body of work studying how programmers interact with LLM-powered coding assistants [20], [6], [21], [22], [23], [24], [25], [26], and, in particular, how they evaluate and debug LLM-generated code. These studies show that programmers spend a significant proportion of their time inspecting generated code [22], [24] and often have trouble understanding [21] and debugging [22] such code, or evaluating its correctness [26]. Moreover, both programmers [22], [26] and data scientists [25] use multiple sources of information when evaluating AI-generated code, including inspecting the code itself as well as executing the code and inspecting its behavior on concrete inputs. While these studies do not focus on end users, they generally motivate the need for better tool support for understanding and debugging LLM-generated code; in the context of end-user programming there is an additional challenge that the user has no option to inspect the code.

B. Code Inspection in Program Synthesis

Research on more traditional (search-based) program synthesis has explored multiple ways to help users inspect and disambiguate generated programs, for example, by displaying intermediate values [27] or generating informative examples [28], [29]; unlike ColDeco, these tools assume that the programmer can always fall back on examining the code. A separate line of research focuses on finding relevant data [30], [31], [32] akin to summary rows. These use different techniques, and are complementary to ColDeco. A more closely related work to ours is FlashProg [33], which introduces user interactions for disambiguating multiple synthesized solutions for end-user tasks. Like ColDeco, FlashProg aims to improve user confidence in the synthesis result without seeing the code, but it is closely tied to the underlying synthesis algorithm and does not support decomposing solutions.

C. Debugging Tools for Spreadsheets

The two core features of ColDeco, helper columns and summary rows, are inspired by previous work in end-user programming and spreadsheet research. Automatically created helper columns have been used before to debug user-written formulas [34] or inspect the behavior of code "foraged" from the web [35]. The main difference between these and ColDeco is our interaction model, where the user interactively decomposes the program via the tree view. Our summary rows take inspiration from templates [36], Lish [37], gradual structuring [38], object spreadsheets [39], and calculation view [40]. Each of these addresses the challenge of comprehending and manipulating a large dataset by abstracting it into a smaller structure, which can be a single exemplar row or formula. ColDeco finds a new application for these ideas —evaluating and debugging AI-generated code.

IV. ColDeco: Design and Implementation

In this section we give more detail about the tool's implementation and describe the technical approach underpinning the two core features of ColDeco—helper columns and summary rows. Throughout this section we use the same running example as in Sec. II (Fig. 1).

A. Implementation

We implement ColDeco as an Excel *taskpane add-in*, written in React. While the design of ColDeco is agnostic to the way the code is generated and even the underlying programming language (as users do not see the code), for our implementation we use the Python pandas API[1] and obtain the code by querying the OpenAI Codex model [1][2]. For our running example, the code produced from the user query is:

```
df["Abbreviation"] = df["First Name"].str[0] \
  + df["Middle Name"].str[0] \
  + df["Last Name"].str[0]
```

Here `df` is a *dataframe* representing the entire table; the assignment above adds a new column `"Abbreviation"` to the dataframe; the values in the new column are computed row-wise from values of three existing columns (the iteration over the rows is implicit in this case, but in more complex programs it becomes explicit).

A server running Pyodide[3] evaluates the Python code and returns the updated dataframe `df` to the ColDeco client, which projects the dataframe into the grid.

B. Decomposition into helper columns

Recall from Sec. II that a ColDeco user can interact with the tree view to iteratively decompose the computation into helper columns. The core technical concept underpinning this feature is the *column decomposition* algorithm, which breaks down a complex column assignment into multiple simpler assignments, akin to A-Normal Form transformation [41]. In our case, one step of column decomposition for the program above yields:

```
df["$fresh1"] = df["First Name"].str[0] \
  + df["Middle Name"].str[0]
df["$fresh2"] = df["Last Name"].str[0]
df["Abbreviation"] = df["$fresh1"] \
  +  df["$fresh2"]
```

You can see how the original right-hand side *rhs* of the form $(e_1 + e_2) + e_3$ was split into its immediate subexpressions, $e_1 + e_2$ and e_3, each of which became a helper column. The decomposition is a little more involved when *rhs* explicitly iterates over rows using the pandas' `apply` function, as in the following example:

```
df["B"] = df.apply(lambda x: \
  "gt10" if x["A"] > 10 else "leq10", axis=1)
```

[1]https://pandas.pydata.org/

[2] Codex was discontinued after our user study. However ColDeco can use any code-generating LLM.

[3]https://pyodide.org/

In this case, our algorithm decomposes the body of the `lambda` abstraction, and then re-wraps each sub-expression into a separate `lambda` and `apply`.

Decomposition is only defined when *rhs* is *non-atomic*, *i.e.* it contains at least one sub-expression whose output is a non-constant column, not already present in the table. For example, the expression `df["Last Name"].str[0]` is atomic, because none of its sub-expressions satisfy our criteria: `df` is not a column, `"Last Name"` and `0` are constants, and `df["Last Name"]` is already present.

After the program has been decomposed into multiple assignments, ColDeco post-processes them to replace fresh column names with descriptive names, computed based on the expression and the original column names. It then constructs the tree view by building a dependency tree over the column assignments. Within the tree view, the button to *Expand* a column is only enabled when the currently selected column is non-atomic. The tree view allows users to hide helper columns, reversing the decomposition; we implement this by caching the original expression and restoring it when required.

C. Summary rows

When an input table is large, it can be difficult to verify the program's output against the user intent. Summary rows facilitate such verification by highlighting a small number of rows that represent unique program behaviors.

The core idea behind summary rows is to *abstract* each row into a vector of *tags* and then cluster the rows by their tag vector. The tags are computed for every value in the row and reflect whether this value: is `NaN`[4], is `""`, is `True`, is positive, or is one of the enumeration values, where a column is considered to be an enumeration if it has at most three distinct string values. For example, in row 9 of Fig. 1, the value `NaN` in "text concatenation" is abstracted into the singleton tag `[isNaN]`, while the value `"S"` in its neighboring column is abstracted into an empty tag.

To compute the tag vector of a row, we string together the value tags from all of its columns. To this end, we consider the fully decomposed set of columns (as in Fig. 1 (bottom)), independently of the current state of the tree view. This has the benefit that rows get clustered not only based on the input and output values of the computation, but also the intermediate values. In our example, clustering the rows by their tag vectors results in two clusters: one includes rows 9 and 12, where the "Middle Name" column is tagged with `[isEmpty]` and three of the derived columns are tagged with `[isNaN]`, and the rest of the rows where all the tags in the vector are empty.

Once the rows have been clustered, ColDeco constructs the *Inspect Rows* view by picking one row per cluster, and additionally displaying the size of each cluster. As we mentioned in Sec. II, ColDeco also restricts the columns depicted in this view to those that participate in the computation.

[4]`NaN` is the pandas error value, displayed as `EMPTY` in ColDeco.

V. User Study

We conducted an hour-long within-subjects study with 24 participants to answer the following research questions:

RQ1 Do ColDeco's features enable users to correctly diagnose generated code outputs in spreadsheets?
RQ2 How does decomposing the output into helper columns affect users' ability to diagnose the output?
RQ3 What are users' perception of the usefulness of each of the features for inspecting ColDeco's output?

A. Participants

For this study, we recruited 24 participants, 10 women and 14 men, across 12 professions. 19 participants reported having "a lot of experience" with spreadsheet software, but all had at least some experience. All participants also had some experience writing spreadsheet formulas, with 8 using "a variety of different functions" and the others only using "a few basic functions such as `SUM` and `AVERAGE`". For traditional programming languages, 18 participants were at least "moderately experienced" programmers, with the others knowing only enough for "small infrequent tasks" or with little to no experience.

B. Tasks

Each participant was asked to solve 4 tasks inspired by the Wrex study [42], and Excel questions on StackOverflow[5]. Each task includes a table of data[6], a task description, and a pre-written query for ColDeco[7]:

A1) Given a table of purchase data, create a column containing the total amount paid after discounts and reimbursements for each entry.
A2) Given a table of TV shows, create a column containing "Yes" if a show's popularity is $\geq 1,000$ or it has a vote average of ≥ 8.0 with at least 10,000 votes.
B1) Given a table of event dates and locations, create a column containing the duration of each event in hours.
B2) Given a table of books, create a column which rounds the price of each book such that the last digit of the rounded price is the nearest 4, 5, or 9.

The query given for A1 and B1 resulted in a correct solution, while A2 and B2 had bugs affecting a small set of the rows. We grouped the tasks into two pairs (A1, A2) and (B1, B2) of approximately the same difficulty (A1 and B1 are simpler and A2 and B2 are more complex).

For each task, participants were asked to use a pre-written query to generate an output, and use ColDeco's inspection features to *diagnose* if the output matches the task's description. To finish the task, they were asked if the output is correct (their "diagnosis"), their confidence in their diagnosis, and (if they diagnosed it as incorrect) what query they would try next to get a correct output.

[5]https://stackoverflow.com/questions/tagged/excel-formula
[6]Taken from [42], [43], [44] or created for the study.
[7]You can find the the study material in the technical report [45].

Since the focus of the study is validating ColDeco's output, not completing tasks, we controlled for the variability in input queries by providing users with the query to use, and did not ask them to try to get to a correct output. Asking them about the query they would try next enabled us to confirm that participants diagnosed the correct cause for incorrect outputs.

C. Study protocol

We study two configurations of ColDeco, `HC` and `No-HC`, which differ in the availability of the *helper columns* feature (the `No-HC` version does not have the *Expand* button in the *Inspect Columns* view). We chose to isolate the effects of the helper columns in particular, since ColDeco has multiple interacting features and column decomposition is the most novel aspect. We did not compare against a traditional "control" condition, since we were not aware of any comparable tools, and we believe that access to ColDeco's features would likely be trivially better than having access to no debugging tools at all.

We had four randomly-assigned groups of 6 participants, based on the condition they were assigned to first (`HC`-first vs. `No-HC`-first) and the task pair (A-first vs. B-first).

The study was conducted remotely via video conferencing, with participants controlling ColDeco on the investigator's machine. Each study session began with a tutorial of the tool (using the example covered in Sec. II), followed by a warm-up where users were asked to repeat the tutorial steps and ask any questions. They then performed the first pair of tasks, followed by a mid-study survey, the second pair of tasks and the post-study survey, ending with a semi-structured interview.

Users had at most 7 minutes to perform each task. If any task exceeded that limit, the investigator informed the participant that they were out of time, and moved to the next task or survey. Participants were encouraged to think-aloud throughout the study.

The participants who saw the `No-HC` condition first were given the column decomposition portion of the tutorial after the mid-study survey, while those in `HC` were simply informed that they will not have access to the *Expand* button for the second pair of tasks. Each group only answered questions about column decomposition in the survey immediately following their use of the feature.

VI. Results

Users completed task A1 with a mean time of 4.37 minutes (SD=1.42), A2 in 4.63 minutes (SD=1.46), B1 in 3.38 minutes (SD=1.40) and B2 in 4.67 minutes (SD=1.23), excluding time-outs. Since this was a think-aloud study, we do not compare times across conditions.

A. Correctness and confidence

Users across both conditions performed well (Fig. 2) and were confident (Fig. 3) on the simpler tasks, but they did not perform as well in A2 and B2. We attribute this to time

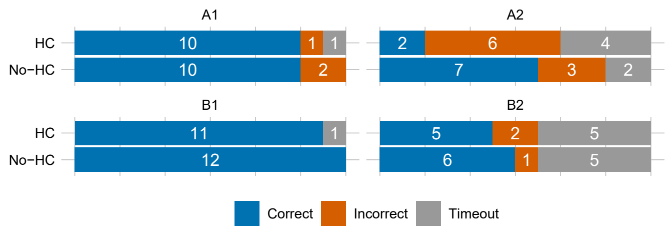


Fig. 2: The number of participants whose diagnosis for each was correct, incorrect, or they ran out of time.

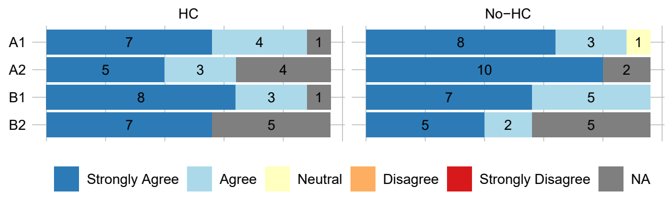


Fig. 3: Participants' agreement with the sentence "I am confidence about my answer".

pressure, where users were not able to inspect the outputs with the same level of detail, and it is possible that the additional cognitive load caused by the more complicated tasks impacted users' performance. Given these results **we answer RQ1 with a tentative yes**: without the artificial time constraint imposed by the study, users are able to correctly diagnose generated code, though future studies are needed to confirm that this generalizes to longer and more complex tasks.

B. The effect of column decomposition

Fig. 2 shows a lower performance on A2 for the `HC` group —six incorrect diagnoses vs. three for `No-HC`—although this difference was not statistically significant (Fisher's Exact Test). The significantly lower confidence (Wilcoxon Rank Sum Test, $p = 0.046$) for A2 in this group (Fig. 3), and the fact that we don't see this effect in B2, suggests that this is not due to decomposition misleading users. Rather, we hypothesize that it is due to the program used in A2: it was a chain of if-else statements, with a bug in the final case. Users needed to decompose the program four times before reaching the values they were interested in. So the complexity of the interaction, combined with relatively little experience with the tool may have overwhelmed and frustrated users.

So **to answer RQ2**, we did not find evidence of significant improvements resulting from the presence of helper columns based only on the quantitative results. We believe, however, this is due to the limitations in our study design (the combination of time pressure and the complexity of the task); indeed, our qualitative evaluation of helper columns, discussed below, is much more positive.

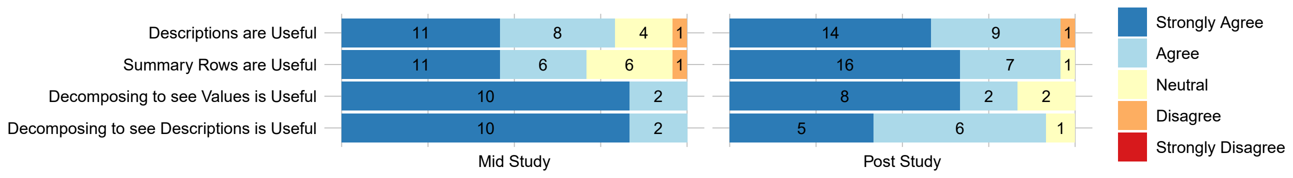


Fig. 4: The results from the surveys. For each question, participants were asked to select their agreement with the statement on a five-level Likert scale. Note that participants in each condition only answered questions about decomposition in the survey immediately following their use of the feature, so the last 2 rows show 12 responses each.

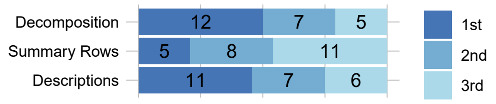


Fig. 5: The number of participants who ranked each feature in a particular position. Some ranked two or all three features equally.

C. Survey results

Fig. 4 presents participants' answers to the mid- and post-study surveys. There was no significant difference between the conditions, so we report the responses to shared questions in aggregate. In the post-study survey, users rated both the natural-language descriptions (M = 4.50, SD = 0.722) and summary rows (M = 4.625, SD = 0.576) highly. We also found a significant *improvement* in the ratings for summary rows between the mid- and post-study surveys (Wilcoxon Signed Rank test, p = 0.016). As we discuss in VI-E3, this suggests a learning effect for this feature, which we found to be harder to learn than others.

Since participants only had access to column decomposition in half of the study, we included questions about that feature only in the survey immediately following that half (the mid-study survey for `HC`-first, the post-study survey for `No-HC`-first). Despite the lower performance noted above, participants rated decomposition highly, for inspecting both the values (M = 4.5, SD = 0.798) and the descriptions (M = 4.33, SD = 0.651) of helper columns.

D. Ranking the features

As a part of the post-study interview, we asked participants to rank the three features (see Fig. 5). We found diversity of preferences, with no clear preference shared by all. Two participants, P21 and P23, ranked decomposition and natural-language descriptions in combination as their first choice. P1 similarly ranked them together, but in last place. P4 considered all features equally useful. The summary rows tended to be ranked lower than others.

Given the ranking and survey results, **we can answer RQ3**: participants found all three features useful, and there was no clear preference for any one feature. However, the summary rows feature was ranked lower than others, which we attribute to its steeper learning curve.

E. Qualitative analysis

We transcribed the semi-structured interviews and participants' comments, and the first author performed an inductive thematic analysis of the transcript (through open-coding) [46]. Here, we discuss notable themes from the analysis, and how they pertain to the results above.

1) Decomposition affords better transparency and analysis: Participants were positive about decomposition, and many commented on its utility during the survey and while ranking the features. Several participants stated that decomposition afforded them a better understanding of the steps involved in the output (P7, 14, 17, 19, 23), with P19 referring to it as a "show-your-work button", and P23 saying that decomposition "makes it less [...] like a black box". Others (P15, 19, 21, 22) noted its role in finding the precise cause of an error, saying that it helps "drill the formula down" (P22) and "pinpoint exactly which part of the prompt is not working well" (P15).

2) ColDeco for collaboration: Participants were excited to use ColDeco for collaboration with their colleagues. P11 and P15 noted that the helper columns would help them *explain* their work to someone else. P6 and P19 commented that natural-language descriptions would help them *understand* complex formulas written by others. And P6 and P15 considered using natural-language descriptions to automatically *document* their spreadsheets.

3) Difficulties with summary rows: We found that users tended to struggle with forming a usable mental model of the summary rows, instead preferring to manually inspect the values in the grid. P6 mentioned that "I don't really understand it, so I wanted to look at the table myself", and P20 stated "I still don't know what it means." Users' comments indicate that this is not due to the nature of the information presented in the summary rows view, but rather that lack of transparency makes it hard to understand how it works and how it can be useful.

Confirming the learning effect noted in Sec. VI-C, some users mentioned that they needed practice to use summary rows effectively. P2 noted that "I feel like once I start using it I might get a grasp of what's happening there", and P14 only used it for their last task, saying "I think I [didn't] understand [summary rows] before this [...] Maybe I got used to it, because it's my fourth time using this program." Others mentioned that more transparency would help their understanding. P4 suggested that "it would be helpful in [summary rows], when it's showing different categories, to specify what the differences are", and similarly P5 wanted "a drop-down that would give you a description". P3, P7 and P21 called for the ability to click on groups to see more example rows for that group, while P21 suggested using a different color to distinguish each behavior.

The usefulness of the information presented in the summary rows view is further confirmed by users who *did* form a usable mental model of the view, and noted its effectiveness in finding errors, especially in combination with other features. P12 found it very helpful "because it brings the different outcomes and behaviors to the front of the screen very quickly." And P16 and P22 mentioned that it would be the first feature they would use, as it lets them quickly check for multiple or incorrect behaviors.

4) Decomposition design suggestions: Alongside the broadly positive response, some participants noted certain limitations with the current design of column decomposition and suggested improvements (P4, 12, 17, 19, 24). The main issue was that, with the more complex tasks, the number of times the output could be decomposed grew, resulting in a large number of helper columns and a more complex tree view. For instance, P24 noted that "As I kept expanding, I kept seeing the other columns for the other cases [...] so it got confusing which ones were related to *just* [...] ones I was actually interested in at that time". As a solution to this, P19 mentioned wanting the ability to selectively expand subexpressions, imagining a design in which they could "highlight part of the formula that I'm interested in and say 'Show me this as a helper column' ".

F. Threats to validity

The most notable threat to *internal* validity of our study is the time limit for each task, which led to some participants timing out, particularly for the more complex tasks A2 and B2. However, users were quite effective at using the tool despite the time limit, which suggests that given more time, users' effectiveness and perceptions may improve. Another threat to internal validity is Participant Response Bias [47]. Following recommendations from [48], we tried to mitigate inflation in subjective ratings and qualitative feedback by presenting multiple designs within ColDeco to the participants.

Our most significant threat to *external* validity is that a majority of our participants had moderate-to-high programming experience. Our participants nevertheless present an important subset of spreadsheet users, as they have a variety of professions and formula experience. Another threat to external validity is that the tasks and data may not represent Excel's real-world use cases. To address this, we used real questions from StackOverflow to inform our tasks, and used real-world data where applicable.

VII. Discussion

ColDeco was built to help us understand how to help users diagnose faults in AI-generated code solutions for tabular data problems. Our study identified the following areas for improvement.

Handling complicated tasks. Users found that for a more complex task, the number of helper columns shown could become overwhelming. An improvement would provide a program slicing [49] capability that could prune the helper columns to only show immediately relevant columns to the calculation of a particular value.

Explaining summary rows. Some users did not understand the meaning or purpose of the row summaries. Based on this feedback, improvements include better documentation, generating natural language explanations of the groupings, or including automatically generated insights about core differences between the groupings.

Handling different kinds of input and output data. Our prototype supports diagnosing errors for a limited, but important, subset of Excel tables (single flat column-major tables). Because code-generating AI can produce code solutions for a wider variety of contexts, it is important to consider how the approaches we outline generalize. Similarly we focused on code that generates columns of results but more general outputs, such as single values and new tables, must be considered.

VIII. Conclusion

We present ColDeco, a new spreadsheet user experience designed to give users confidence that the code being generated by the LLM is correct. To the best of our knowledge, ColDeco is the first end-user inspection tool for comprehending code produced by LLMs for tabular data tasks. ColDeco provides *summary rows*, which highlight collections of rows that exhibit distinct behaviors in the code, and *helper columns*, which map the results of subcomputations back on the table.

We evaluate ColDeco using a within-subjects user study. In both quantitative and qualitative measures, our subjects found row summaries and helper columns were valuable in understanding the AI-generated code solutions. We found that while all three features are independently useful, participants preferred them in combination. Users especially noted the usefulness of helper columns and natural language explanations, but wanted more transparency in how summary rows are generated to assist with understanding and trusting them.

Topics for future work include understanding the application of ColDeco in collaborative settings for explaining and understanding existing formulas. This aspect of using ColDeco was highlighted by our users and further investigation is needed. While ColDeco focused on detecting and diagnosing potential errors, a natural and important extension of this capability is the ability to repair errors once they are found. Integrating repair into the ColDeco experience is an important topic for further study.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR,* vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[2] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311,* 2022.

[3] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago *et al.*, "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814,* 2022.

[4] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068,* 2022.

[5] N. Friedman. (2021, Jun) Introducing GitHub Copilot: your AI pair programmer. GitHub. [Online]. Available: https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/

[6] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, "What is it like to program with artificial intelligence?" *arXiv preprint arXiv:2208.06213,* 2022.

[7] B. Du Boulay, "Some difficulties of learning to program," *J. Educational Computing Research,* vol. 2, no. 1, 1986.

[8] C. Lewis and G. M. Olson, "Can principles of cognition lower the barriers to programming," in *Empirical studies of programmers: second workshop*, vol. 2, no. 15. Ablex Publishing Corporation, Norwood, New Jersey, 1987, pp. 248–263.

[9] B. Shneiderman, "Direct manipulation: A step beyond programming languages," *Computer,* vol. 16, no. 8, pp. 57–69, 1983.

[10] A. Kay, "Computer software," *Scientific American,* vol. 251, no. 3, pp. 52–59, 1984. [Online]. Available: http://www.jstor.org/stable/24920344

[11] A. Sarkar, S. S. Ragavan, J. Williams, and A. D. Gordon, "End-user encounters with lambda abstraction in spreadsheets: Apollo' s bow or Achilles' heel?" in *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2022, pp. 1–11.

[12] B. A. Nardi and J. R. Miller, "The spreadsheet interface: A basis for end user programming," in *INTERACT.* North-Holland, 1990, pp. 977–983.

[13] P. Dourish, *The stuff of bits: An essay on the materialities of information.* MIT Press, 2017, ch. Spreadsheets and Spreadsheet Events in Organizational Life.

[14] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers *et al.*, "The state of the art in end-user software engineering," *ACM Computing Surveys (CSUR),* vol. 43, no. 3, pp. 1–44, 2011.

[15] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices,* vol. 46, no. 1, pp. 317–330, 2011.

[16] K. Dhamdhere, K. S. McCurley, R. Nahmias, M. Sundararajan, and Q. Yan, "Analyza: Exploring data with conversation," in *Proceedings of the 22nd International Conference on Intelligent User Interfaces*, ser. IUI '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 493–504. [Online]. Available: https://doi.org/10.1145/3025171.3025227

[17] M. Ginés. (2022) Better insights from analyze data feature in Excel. Microsoft Corporation. Office Insider Blog. [Online]. Available: https://insider.office.com/en-us/blog/better-insights-from-analyze-data-feature-in-excel

[18] M. X. Liu, A. Sarkar, C. Negreanu, B. Zorn, J. Williams, N. Toronto, and A. Gordon, ""What it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models," in *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems.* ACM, April 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/what-it-wants-me-to-say-bridging-the-abstraction-gap-between-end-user-programmers-and-code-generating-large-language-models/

[19] G. Chalhoub and A. Sarkar, ""It' s freedom to put things where my mind wants"': Understanding and improving the user experience of structuring data in spreadsheets," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491102.3501833

[20] J. D. Weisz, M. Muller, S. Houde, J. Richards, S. I. Ross, F. Martinez, M. Agarwal, and K. Talamadupula, "Perfection not required? human-AI partnerships in code translation," in *26th International Conference on Intelligent User Interfaces.* New York, NY, USA: Association for Computing Machinery, 2021, pp. 402–412. [Online]. Available: https://doi.org/10.1145/3397481.3450656

[21] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491101.3519665

[22] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," 2022. [Online]. Available: https://arxiv.org/abs/2206.15000

[23] A. Ziegler, E. Kalliamvakou, S. Simister, G. Sittampalam, A. Li, A. Rice, D. Rifkin, and E. Aftandilian, "Productivity assessment of neural code completion," 2022.

[24] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in AI-assisted programming," 2022.

[25] A. M. McNutt, C. Wang, R. A. DeLine, and S. M. Drucker, "On the design of AI-powered code assistants for notebooks," 2023.

[26] J. T. Liang, C. Yang, and B. A. Myers, "Understanding the usability of AI programming assistants," 2023.

[27] H. Peleg, R. Gabay, S. Itzhaky, and E. Yahav, "Programming with a read-eval-synth loop," *Proc. ACM Program. Lang.,* vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428227

[28] T. Zhang, L. Lowmanstone, X. Wang, and E. L. Glassman, "Interactive Program Synthesis by Augmented Examples," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 627–648. [Online]. Available: https://doi.org/10.1145/3379337.3415900

[29] M. B. James, Z. Guo, Z. Wang, S. Doshi, H. Peleg, R. Jhala, and N. Polikarpova, "Digging for fold: synthesis-aided API discovery for Haskell," *Proceedings of the ACM on Programming Languages,* vol. 4, no. OOPSLA, pp. 205:1–205:27, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428273

[30] A. Fariha and A. Meliou, "Example-driven query intent discovery: Abductive reasoning using semantic similarity," *Proc. VLDB Endow.,* vol. 12, no. 11, p. 1262–1275, jul 2019. [Online]. Available: https://doi.org/10.14778/3342263.3342266

[31] A. Fariha, M. Brucato, P. J. Haas, and A. Meliou, "Sudocu: Summarizing documents by example," *Proc. VLDB Endow.,* vol. 13, no. 12, p. 2861–2864, aug 2020. [Online]. Available: https://doi.org/10.14778/3415478.3415494

[32] E. K. Rezig, A. Bhandari, A. Fariha, B. Price, A. Vanterpool, V. Gadepally, and M. Stonebraker, "Dice: Data discovery by example," *Proc. VLDB Endow.,* vol. 14, no. 12, p. 2819–2822, jul 2021. [Online]. Available: https://doi.org/10.14778/3476311.3476353

[33] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani, "User interaction models for disambiguation in programming by example," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 291–301. [Online]. Available: https://doi.org/10.1145/2807442.2807459

[34] T. Schmitz, D. Jannach, B. Hofer, P. Koch, K. Schekotihin, and F. Wotawa, "A decomposition-based approach to spreadsheet testing and debugging," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017, pp. 117–121.

[35] S. Lau, S. S. Srinivasa Ragavan, K. Milne, T. Barik, and A. Sarkar, "Tweakit: Supporting end-user programmers who transmogrify code," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3411764.3445265

[36] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert, "Visual specifications of correct spreadsheets," in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 2005, pp. 189–196.

[37] A. Hall, M. Wermelinger, T. Hirst, and S. Phithakkitnukoon, "Structuring spreadsheets with the "lish" data model," *arXiv preprint arXiv:1801.08603*, 2018.

[38] G. Miller and F. Hermans, "Gradual structuring in the spreadsheet paradigm," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2016, pp. 240–241.

[39] M. McCutchen, S. Itzhaky, and D. Jackson, "Object spreadsheets: A new computational model for end-user development of data-centric web applications," in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2016, pp. 112–127.

[40] A. Sarkar, A. D. Gordon, S. P. Jones, and N. Toronto, "Calculation view: multiple-representation editing in spreadsheets," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 85–93.

[41] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 1993, pp. 237–247.

[42] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. Gulwani, "Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. CHI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–12. [Online]. Available: https://doi.org/10.1145/3313831.3376442

[43] A. Pal, "Top 50 Bestselling Novels 2009-2020 of Amazon," Feb. 2021. [Online]. Available: https://www.kaggle.com/datasets/palanjali007/amazons-top-50-bestselling-novels-20092020

[44] A. Ramachandran, "Top IMDB Rated TV Shows," Jul. 2022. [Online]. Available: https://www.kaggle.com/datasets/adityaramachandran27/top-imdb-rated-tv-shows

[45] K. Ferdowsi, J. Williams, I. Drosos, A. Gordon, C. Negreanu, N. Polikarpova, A. Sarkar, and B. Zorn, "ColDeco: An end user spreadsheet inspection tool for AI-generated code," Microsoft Research, Tech. Rep., 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/coldeco-an-end-user-spreadsheet-inspection-tool-for-ai-generated-code/

[46] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[47] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies, ""Yours is better!": participant response bias in HCI," in *CHI*. ACM, 2012, pp. 1321–1330.

[48] M. Tohidi, W. Buxton, R. Baecker, and A. Sellen, "Getting the right design and the design right," in *CHI*. ACM, 2006, pp. 1243–1252.

[49] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.