# Programming by Example Made Easy

JIARONG WU, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, China
LILI WEI, Department of Electrical and Computer Engineering, McGill University, Canada
YANYAN JIANG*, State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, China
SHING-CHI CHEUNG*, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, China
LUYAO REN, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, EECS, Peking University, China
CHANG XU, State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, China

Programming by example (PBE) is an emerging programming paradigm that automatically synthesizes programs specified by user-provided input-output examples. Despite the convenience for end-users, implementing PBE tools often requires strong expertise in programming language and synthesis algorithms. Such a level of knowledge is uncommon among software developers. It greatly limits the broad adoption of PBE by the industry. To facilitate the adoption of PBE techniques, we propose a PBE framework called Bee, which leverages an "entity-action" model based on relational tables to ease PBE development for a wide but restrained range of domains. Implementing PBE tools with Bee only requires adapting domain-specific data entities and user actions to tables, with no need to design a domain-specific language or an efficient synthesis algorithm. The synthesis algorithm of Bee exploits bidirectional searching and constraint-solving techniques to address the challenge of value computation nested in table transformation. We evaluated Bee's effectiveness on 64 PBE tasks from three different domains and usability with a human study of 12 participants. Evaluation results show that Bee is easier to learn and use than the state-of-the-art PBE framework, and the bidirectional algorithm achieves comparable performance to domain-specifically optimized synthesizers.

Additional Key Words and Phrases: program synthesis, programming by example

*Corresponding authors

Authors' addresses: Jiarong Wu, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China, jwubf@cse.ust.hk; Lili Wei, Department of Electrical and Computer Engineering, McGill University, Montreal, Canada, lili.wei@mcgill.ca; Yanyan Jiang, State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China, jyy@nju.edu.cn; Shing-Chi Cheung, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China, scc@cse.ust.hk; Luyao Ren, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, EECS, Peking University, Beijing, China, rly@pku.edu.cn; Chang Xu, State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China, changxu@nju.edu.cn.


1049-331X/2023/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

**ACM Reference Format:**
Jiarong Wu, Lili Wei, Yanyan Jiang, Shing-Chi Cheung, Luyao Ren, and Chang Xu. 2023. Programming by Example Made Easy. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (August 2023), 35 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

**Background and Motivation.** Given a set of program inputs and their expected corresponding outputs as *examples*, programming by example (PBE) is a program synthesis paradigm that can automatically deduce a program conforming to the provided examples [19, 20]. PBE has been widely demonstrated to be successful in automating domain-specific repetitive end-user tasks [4, 18, 27, 32, 33, 41, 44, 54]. Despite such reported successes, PBE is seldom a supported feature in mainstream software products. Thus far, each PBE implementation requires a concise domain-specific language for the target application and an algorithm optimized for the language. The skills involved are out of reach by many developers, hindering the wide adoption of PBE.

Let us illustrate the capability of PBE and the skills involved in supporting such a capability using a GIF editing task. Suppose that Cathy wants to create an animated GIF in TikTok style as shown in Figure 1. Existing video editing software allows Cathy to place copies of an image (`tiktok.jpg`) on a timeline and apply graphical effects to the image copies. However, when the effects follow a non-trivial pattern, in which the green-blue channel shifts in opposite directions for odd/even frames with increasing amplitude, Cathy has to manually apply the effects *separately for each image copy*, which is labor-intensive and error-prone. Existing industrial video editing software provides no easy way to create such effects.

The editing task can be automated using PBE. With appropriate customization, existing PBE techniques are capable of synthesizing programs to automate effects creation for each frame. Below is an example code snippet in C# style.

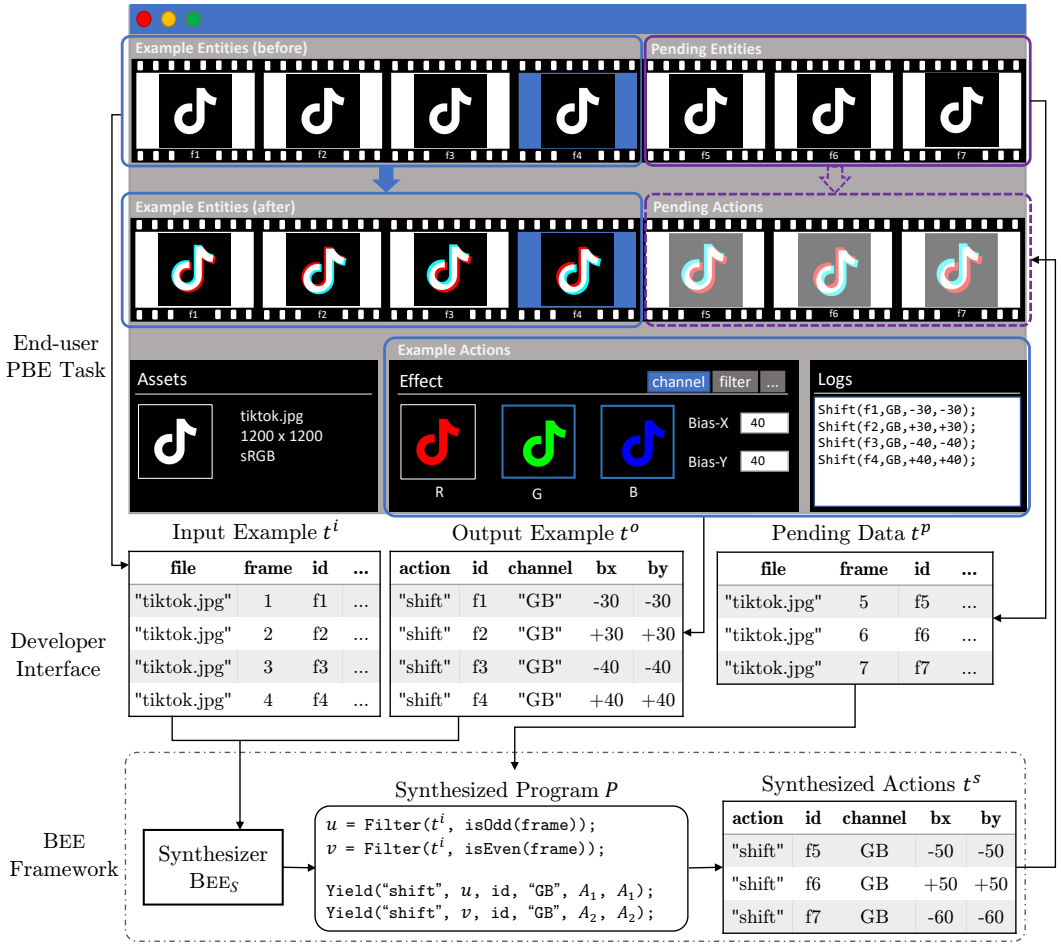
```
int b = 20 + (frame_id + 1) / 2 * 10;  // calculating the biases in pixels
return (frame_id % 2) switch {
    0 => shift("GB", b, b),    // for even frames, shift down and right (positive biases)
    1 => shift("GB", -b, -b)   // for odd frames, shift up and left (negative biases)
}
```

In general, software developers have three options with which to incorporate PBE support (like the example above) into their tools.

(1) Implement the PBE feature from scratch, which is a typical approach in the research literature [18, 33, 54, 59]. This requires designing a domain-specific language (DSL) and an algorithm to synthesize programs satisfying the user-given examples [1].
(2) Adopt a PBE framework (e.g., Prose [26, 42] or Trinity [31]) by designing a DSL and customizing the framework's built-in synthesis algorithm.
(3) Reuse existing PBE tools whose DSLs offer the desired expressiveness. For example, the code snippet above requires a PBE tool supporting conditional integral arithmetics.

However, these options require being acquainted with DSL and algorithm design. Such a requirement is not commonly amenable to software developers. Existing research [42] indicates that implementing a PBE feature is time-consuming, even for PBE experts.[1] Moreover, existing PBE implementations are usually research artifacts specifically built for a specific domain. Adapting them beyond these specific domains is non-trivial.

[1]This is confirmed by our human study results (Section 5.2), where participants, who are experienced developers but without PBE expertise, found the state-of-the-art PBE framework, Prose, difficult to learn and use.




| file | frame | id | ... |
|---|---|---|---|
| "tiktok.jpg" | 1 | f1 | ... |
| "tiktok.jpg" | 2 | f2 | ... |
| "tiktok.jpg" | 3 | f3 | ... |
| "tiktok.jpg" | 4 | f4 | ... |

| action | id | channel | bx | by |
|---|---|---|---|---|
| "shift" | f1 | "GB" | -30 | -30 |
| "shift" | f2 | "GB" | +30 | +30 |
| "shift" | f3 | "GB" | -40 | -40 |
| "shift" | f4 | "GB" | +40 | +40 |

| file | frame | id | ... |
|---|---|---|---|
| "tiktok.jpg" | 5 | f5 | ... |
| "tiktok.jpg" | 6 | f6 | ... |
| "tiktok.jpg" | 7 | f7 | ... |

| action | id | channel | bx | by |
|---|---|---|---|---|
| "shift" | f5 | GB | -50 | -50 |
| "shift" | f6 | GB | +50 | +50 |
| "shift" | f7 | GB | -60 | -60 |

Fig. 1. A PBE task of GIF editing in a hypothetical video editing application, in which a designer creates a TikTok-style GIF animation [37] by providing a few example frame editing actions. With the table representations of example frames/actions modeled by the application developer, Bee generalizes the examples and synthesizes a program $P$ (semantics available in Figure 2) to automate the conversion of remaining frames. The GUI is fabricated, but the user actions are adapted from ImageMagick [35] commands.

We propose to bridge the gap with a novel PBE framework, Bee, which enables software developers to quickly implement efficient PBE tools via an easy-to-use interface without expertise in DSL and algorithm design. Bee can be applied to PBE tasks in a variety of domains and scenarios.

**Programming by Example Made Easy.** As discussed above, a major obstacle to PBE implementation is its requirement to design a concise DSL and a synthesis algorithm optimized for the DSL. The DSL is so designed to concisely express domain-specific tasks and allow for a tractable synthesis procedure.

Bee relieves the effort of DSL design with a predefined SQL-like meta DSL for a class of PBE tasks that *perform repetitive user actions over a large number of similar data entities*. This class is bounded but is also inclusive and meets the mainstream application scenario of PBE. Software developers

using Bee do not need to modify/customize this meta DSL. To ease presentation, we refer to these developers using PBE frameworks to implement the PBE features as *PBE developers*. The adoption of a SQL-like meta DSL for Bee is inspired by the observation that scripts for automating repetitive actions over similar entities usually comprise two parts:

(1) Representation of data entities and user actions (e.g., frames and channel shifts in Figure 1). *Relational tables* (or simply tables) containing cells of primitive types (e.g., strings and integers) are suitable for uniformly representing data entities, as evidenced by the success of the entity-relation model (relational databases and ORM [40]) in a wide variety of application domains. Actions performed over data entities can also be represented in the form of tables, where the first column in a table specifies an action type and the remaining columns specify action arguments.
(2) Constructs for program logic description. Most of the logic behind applying repetitive actions over a set of entities with the same data structure can be mechanically expressed as below. (a) Transform the data entities (tables) into views (tables), each of which holds the data entities that are subject to the same kind of user actions, e.g., two views separately holding odd and even frames for the task in Figure 1. Such transformations can be achieved by SQL-like operators (e.g., `WHERE` and `JOIN`). (b) The desired actions (recall that actions are represented as tables) are projected from the views like `SELECT` in SQL. In addition to column-to-column projection, Bee allows the generation of a column (action argument) by applying a computation operator (e.g., linear arithmetic) on certain columns, e.g., bias to shift in Figure 1 is computed on column `frame`. Specifically, Bee limits the computation in a predefined set of commonly used string and integer operators for synthesis tractability.

With Bee shipping the meta DSL for program logic, PBE developers can integrate a PBE feature (like "intelligent task automation") into their application by modeling domain-specific data entities and user actions as relational tables. Take the PBE task in Figure 1 for example, PBE developers can: (1) model the example input frames as tables with self-designed schemas that include the key properties needed in the task (e.g., `frame`)[2] and (2) model the example output user actions as a table, e.g., shifting the green and blue channels of a frame `f1` 30 pixels up and left can be modeled as a row ⟨shift, `f1`, GB, −30, −30⟩.

Given the input-output examples as tables (Bee provides ORM-style APIs [40] for easy Python/C# integration), Bee takes care of the program synthesis by producing a *consistent* program that executes on the example input table and yields the example output table. The data entities pending to be processed would be represented as tables with the same schemas of example inputs. Then, the tables of pending-data can be fed to the synthesized program to generate the desired user actions. Specifically, the generated user actions are represented as a table with the same schema of example output. The software can follow each row in the generated user-action table to perform the corresponding action. For example, the software can follow a row of ⟨shift, `f5`, GB, −50, −50⟩ to create a frame by shifting the green-blue channels of frame `f5` 50 pixels up and left. Alternatively, the software may visualize the generated actions' effects (e.g., the frames to create) and wait for the user's confirmation to enable an interactive PBE.

**Contributions.** This paper presents a novel meta DSL $\text{Bee}_L$ that supports table-wise operations and common value computations. The former includes common SQL operations such as filter, aggregation, join, and union. The latter includes string manipulations and integral arithmetics. To boost efficiency, the synthesis in Bee is driven by a bidirectional algorithm. The forward direction follows the syntax-guided search to explore table-wise operations. The backward direction decomposes the output table into sub-tables (sub-synthesis-problems), solves each separately, and

[2]Further guidelines for schema design are available in Bee's tutorial [37].

merges the solutions later. The interplay between two search directions synthesizes the value computations by solving constraints deduced from matching forward generated tables and backward propagated sub-tables.

We evaluated the effectiveness of Bee by comparing it with synthesizers implemented atop Prose, a state-of-the-art PBE framework, on 64 benchmarks from three application domains: file management, spreadsheet transformation, and XML transformation. The domain-specifically optimized synthesizers powered by Prose solved 47 out of 64 benchmarks, while Bee solved 53 out of 64. The results indicate that the bidirectional synthesis algorithm of Bee is effective for practical scenarios. Bee achieves a performance comparable to that of the domain-specifically optimized synthesizers.

We also evaluated the usability of Bee via a human study with 12 participants without a program synthesis background. They were asked to complete PBE tasks with both Bee and Prose. The feedback shows that Bee is significantly more accessible than Prose in developing PBE tools.

In summary, this paper makes four major contributions.

(1) We propose a customizable PBE framework named Bee that allows programmers to implement PBE tools for their software without expertise in program synthesis. Many existing PBE applications [4, 21, 27, 32, 44, 54, 59] can be viewed in this data-action manner.
(2) We adopt relational tables and SQL-like language $\text{Bee}_L$ to model PBE tasks and design a bidirectional synthesis algorithm that can effectively guide the search over the large search space specified by $\text{Bee}_L$.
(3) We implement a prototype of Bee and apply it to three data processing domains, namely, file management, spreadsheet transformation, and XML transformation.
(4) We evaluate Bee in terms of usability and performance. The evaluation results show that programmers find it effective to implement PBE tools with Bee, and Bee's algorithm is capable of handling real-world PBE tasks.

## 2 OVERVIEW

This section takes the GIF editing task in Figure 1 as an example to illustrate the workflow of (1) how PBE developers can build applications on Bee, and (2) how Bee synthesizes target programs. In Section 2.1, we demonstrate how developers can easily leverage the interface to specify the figures and user actions as tables with several lines of code. In Section 2.2, we introduce the programs synthesized by Bee to automate PBE tasks and briefly illustrate our bidirectional approach to synthesizing programs.

### 2.1 Developer Interface

Bee relieves developers from designing DSLs by using a predefined meta DSL to process entities and actions in various domains. To apply Bee to PBE tasks in a specific domain (e.g., GIF editing in Figure 1), developers need only to model data entities and user actions in the domain as relational tables. Relational tables in Bee are "flat" in that only primitive types (e.g., integers and strings) are allowed. The design decision follows Daniel Jackson's Alloy [23], which points out that flat tables are sufficiently general for software abstractions and tractable for search-based software automation.

**Data Entities.** To use Bee, developers should provide a data entity *model* that (1) identifies what data (and data fields) are related to the repetitive tasks aiming to be automated, and (2) represents the identified data (and data fields) as relational tables. Data entities can be identified from the subjects in the repetitive tasks, i.e., what the users usually need to operate frequently. For example, when editing GIFs, the frame is where users apply changes and thus a natural choice for the data

entity. Developers also need to decide which data fields of the identified data entities are needed in the repetitive tasks. For example, a frame's order in the frame sequence is needed because some PBE tasks (e.g., our motivating task) need to perform user actions dependent on the order.

Modeling the data entities for PBE tasks in tables is like modeling a relational database. A row and a column in the table represent one data entity and one identified data field, respectively. For instance, the four rows in the Input Example table in Figure 1 represent four frame entities, and the columns (e.g., order in the frame sequence) represent various fields of the frames. Bee allows each data entity to be indexed by a unique *identifier* (e.g., the `id` column in Figure 1).

Implementation of a table schema can be accomplished through Bee's ORM-style [40] APIs. Listing 1 gives an example of implementing the schema for the Input Example table in Figure 1. In a nutshell, a table can be implemented by a class annotated with `[Entity]` (annotations in C#). Table columns are realized by methods annotated with `[Field]`. Specially, methods annotated with `[IdField]` return `[Entity]` objects, where each `[Entity]` object would be automatically converted to a unique value in identifier columns (e.g., the `id` column in Figure 1).

```
[Entity]
public class GraphicEntity {
    private File _file;
    private Frame _frame;
    GraphicEntity(File file, Frame frame) {
        _file = file;
        _frame = frame;
    }

    [Field]
    public string File() {
        return _file.Filename;
    }

    [Field]
    public int Frame() {
        return _frame.order;
    }

    [IdField]
    public GraphicEntity Id() {
        return this;
    }

    ...
}
```

Listing 1. A C# code snippet for specifying the table schema of Input Example in Figure 1.

```
[Entity]
public class GraphicEntity {
    ...

    [Action]
    public static void Shift(GraphicEntity ge, string rgb, int biasX, int biasY) {
        foreach (var channel in ge._frame.GetChannels(rgb))
            channel.translate(biasX, biasY);
    }
}
```

Listing 2. A C# code snippet for specifying the shift action, i.e., the table schema of Output Example in Figure 1.

**User Actions.** While data entities describe a system's state, user actions over entities can describe state changes. Like data entities, user actions can be modeled as relational tables. Without loss of generality, we assume that (1) developers hold a set of predefined user actions that can be taken to change the system's state, such as renaming/moving files or making various effects as in Figure 1;

(2) each action can be modeled by a command `action(arg1, arg2, ...)`.[3] Bee generates the table representation of user actions by taking `action`, `arg1`, ... as columns and each action instance as a row. Therefore, developers only need to define user actions in the form of commands (action name and arguments).

User actions are usually implemented as functions/methods over entities. Thus, Bee allows specifying actions with annotations on methods, as the C# example for the shift action in Listing 2.

**PBE Tasks.** In the scenario of automating repetitive user actions over data entities, the process of producing the desired user actions from the inputted data entities usually follows a task-specific logic, i.e., the target program in PBE systems.

Such programs can usually be generalized from a small set of example inputs (data entities) and example outputs (user actions). Each PBE task is a problem of generating programs consistent with the given input-output examples. In Bee, the workflow of solving a PBE task is as follows:

(1) Users initiate a PBE task (e.g., by pressing a "start PBE" button) and select example data entities (e.g., the first several frames in Figure 1).
(2) The software application passes the corresponding `[Entity]` objects to Bee, and Bee converts them into tables.
(3) Users perform example actions (e.g., setting graphic effects in Figure 1).
(4) Each user action triggers an invocation of an `[Action]` method. Bee records the arguments and converts them into tables.
(5) Bee synthesizes programs consistent with the example inputs and outputs.

In summary, using Bee mainly requires the specification of data entities and user actions without knowledge of the domain-specific language and program synthesis algorithm. Such a PBE workflow can be integrated into software applications via lightweight engineering efforts. Further details of the Bee developer interface are available in our tutorial [37], including three PBE tools as instances.

### 2.2 Bee Programs and Synthesis Procedure

**Bee Programs.** In Bee, the process of generating user actions from data entities corresponds to a program performing a sequence of "intermediate actions", each creating a new intermediate table. The final intermediate table contains the user actions. For example, the logic for the task in Figure 1 can be decomposed mechanically into two processes: (1) Extract odd and even frames into two separate tables that compute the biases to shift differently; (2) Generate a user action with the computed biases for each frame. Therefore, Bee programs are designed to consist of two phases accordingly.

(1) The *transformation* phase applies a sequence of table-to-table transformations (e.g., Filter, Join) to generate intermediate tables that would participate in the next phase.
(2) The *mapping* phase generates a table of user actions by projecting the intermediate tables onto the target columns (computed from expressions) using Yield statements.

The transformation phase can be written as a series of SQL-like operators (e.g., Filter resembles a `SELECT` with a `WHERE` clause) that take in a table together with a predicate and output an intermediate table. Figure 2 shows the synthesized program $P$ that generalizes the given input/output examples. The first two statements are Filter statements. The first (resp. second) statement takes in the table $t^i$ (example frames) along with a predicate judging if the frame is odd (resp., even) and outputs a table $u$ (resp., $v$) containing odd-frames (resp., even-frames).

[3]The number of arguments depends on the action, e.g., the directory creating action `mkdir(path)` has one argument while the file moving action `mv(old_path, new_path)` has two.

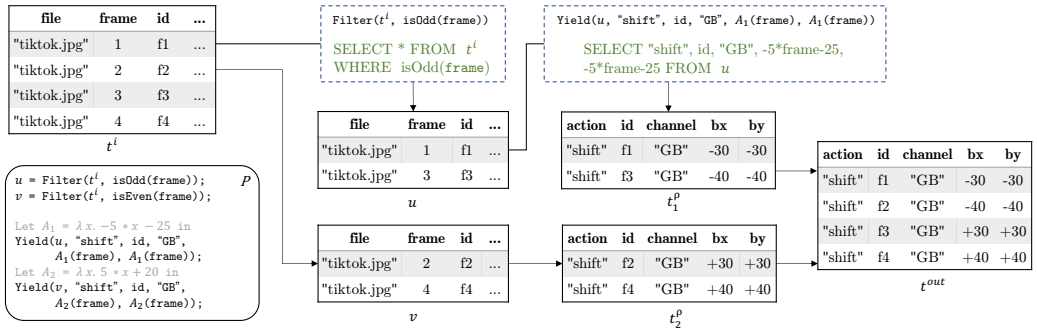



Fig. 2. The program $P$ for the task in Figure 1 and example execution procedure with informal semantics in SQL. We use "Let" expressions here to introduce $A_1$ and $A_2$ to simplify the presentation (the formal grammar of Bee does not include "Let").

The mapping phase can be written as a sequence of Yield statements. Each Yield statement projects a table onto some given "columns" like the SELECT clause in SQL. In addition, we allow arithmetic or string expressions in the projection to support computations such as the bias shifts. Take the first Yield statement in Figure 2 for example. It takes in table $u$ (odd-frames) and outputs table $t_1^\rho$ by executing Yield in the following ways:

(1) Constant expressions (e.g., "shift" and "GB" in Yield) result in columns containing the constant values.
(2) Expressions containing a column name (e.g., `id` in Yield) result in the projected column.
(3) Expressions with computations (e.g., $A_1(\texttt{frame})$ in Yield) result in columns that contain the computed values for each row in the input table. Take $A_1(\texttt{frame}) = -5 * \texttt{frame} - 25$ for example, the resulting column contains $-30$ computed from the first row in $u$ ($\texttt{frame} = 1$) and $-40$ computed from the second row ($\texttt{frame} = 3$).

Moreover, tables generated from Yield statements (e.g., $t_1^\rho$ and $t_2^\rho$) would be unioned into a single table (e.g., $t^{out}$) as final output (the desired user actions).

**Bee Synthesis Algorithm.** To obtain a Bee$_L$ program like the one in Figure 2, we need to synthesize both the transformation part (e.g., the two Filter statements) and the mapping part (e.g., the two Yield statements). A simple algorithm to synthesize example-consistent Bee$_L$ programs is to exhaustively enumerate all syntactically valid programs (Section 3.2) and filter out those that are inconsistent with the given examples.

However, such a simple syntax-guided search strategy is unscalable. There could be hundreds of thousands of programs (and resulting tables) in the transformation phase to enumerate, even if only considering two to three transformation statements. The expressions with constant parameters (e.g., $-5$ and $-25$ in $A_1$) and the "union" semantics of *Yield* statements make efficient searching even more challenging. This paper presents a bidirectional search algorithm in which the backward direction generates partial programs with placeholders whose exact values (e.g., parameters in computation expressions) are determined by a matching phase.

Specifically, statements in the transformation phase are synthesized in the forward-searching by enumerating possible table-to-table transformation statements (e.g., Filter, Join) according to the grammar. Figure 3 shows the synthesis process for the program in Figure 2, where the two Filter statements generating $u$ and $v$ are enumerated in the forward-searching.

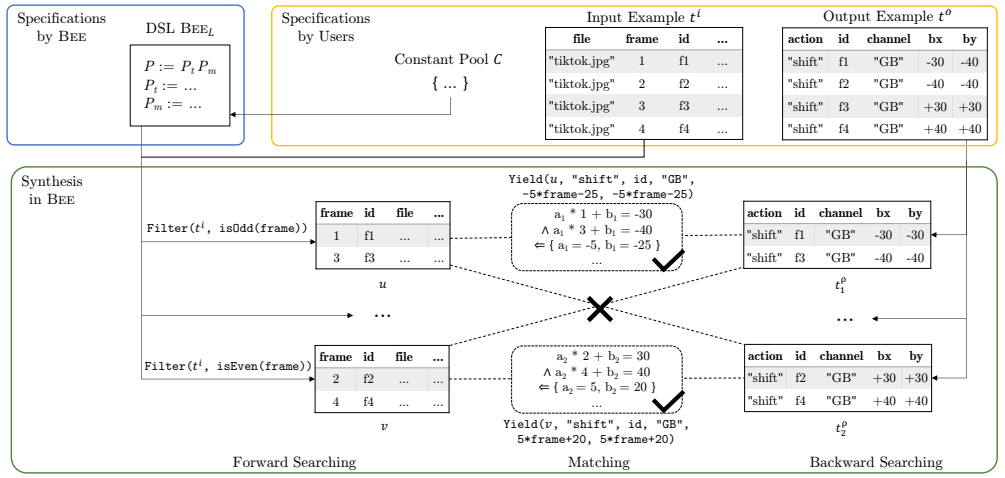



Fig. 3. The bidirectional synthesis procedure for the program in Figure 2. The specifications to the synthesis task include the part provided by Bee (DSL) and the parts provided by users through the software application (input/output example tables and a constant pool). The synthesis procedure in Bee includes forward searching, backward searching, and matching between the two directions. The constant pool $C$ in this example is empty.

Yield statements in the mapping phase are synthesized in the backward-searching. The backward searching first enumerates all possible sub-tables of the example output table, each corresponding to the output of a Yield statement. For example, tables $t_1^\rho$ and $t_2^\rho$ are enumerated in Figure 3. Considering that the number of sub-tables is exponential to the number of rows in the output table, we need to rank the sub-tables to avoid explosion when the number of rows is large (detailed in Section 4.3).

The expressions in the Yield statements are synthesized by matching forward searched intermediate tables against backward searched tables and solving constraints. Specifically, we first range over a forward table, a backward table, and their row-to-row mapping. Then, for each column in the backward table, we enumerate a parameterizable expression from a predefined set that includes common arithmetics and string manipulations. Finally, we check (e.g., with an SMT solver) if there exist parameters for the expression that exactly maps some columns in the forward table to the backward table.

For example, consider that we are matching table $u$ and table $t_1^\rho$ in Figure 3 and we know the row-to-row mapping by comparing the `id` columns. To synthesize column `bx` that needs arithmetic computation, we first enumerate the parameterizable expression $\lambda x.a * x + b$, then check that the `frame` column of $u$ can be mapped to column `bx` with $a = -5, b = -25$.

To summarize, with bidirectional searching and constraint solving, we avoid enumerating parameters in expressions, which is infeasible concerning synthesis efficiency. We present the complete algorithm in Section 4.

## 3 THE BEE SPECIFICATION

With the developer-provided modeling of data entities and user actions, as well as the workflow introduced in Section 2.1, solving a PBE task in Bee is like invoking a function Bee$_S(\langle T^i, t^o \rangle, T^p, C)$, where Bee$_S$ is the program synthesizer in Bee, $T^i$ denotes the tables converted from example data

entities, $t^o$ denotes the table converted from example user actions, $T^p$ denotes the table converted from data entities pending processing, and $C$ is a constant pool.

In this section, we first formulate the above synthesis problem in Section 3.1, then introduce our meta DSL $\text{BEE}_L$ that defines the synthesizable programs in BEE (Section 3.2 to 3.3).

### 3.1 Synthesizer Interface Specification

**Tables.** Two-dimensional table representation of domain-specific data/action is the foundation of BEE. The inputs $T^i, t^o, T^p$ to $\text{BEE}_S$ are all based on tables. Formally, an $n$-ary table $t$ is an *unordered* set of typed tuples. Each table $t$ is associated with a name $\eta$ and a table *schema*

$$\mathrm{T} = \langle col_1 : \tau_1, \ldots, col_n : \tau_n \rangle,$$

where $col_i$ and $\tau_i$ denote the $i$-th column's name and type, respectively. Each row $row \in t$ is a tuple $\langle v_1, \ldots, v_n \rangle$, where each $v_i$ is of type $\tau_i \in \{\texttt{Int}, \texttt{String}, \texttt{Id}\}$. `Id`, like the primary key in SQL, uniquely distinguishes entities in a table. The following operations are defined over tables:

- (*fetch*) $t[row, col]$ or $row[col]$ ($t$ is omitted when the context is clear) denotes the value of the cell at *row* and *col* in table $t$.
- (*project*) $t[col_1, \ldots, col_n]$ denotes a new table obtained by removing columns other than $col_1, \ldots, col_n$ in $t$ and deduplicating repetitive rows.
- (*append*) $t + col_{new}$ denotes appending $col_{new}$ (a list of values) to $t$. Suppose $t$ is an $n$-ary table with $m$ rows $row_1, \ldots, row_m$, and $col_{new}$ is a list of values $v_1, \ldots, v_m$. $t + col_{new}$ yields a new $(n+1)$-ary table with $m$ rows, where the $i$-th row is an $(n+1)$-ary tuple appending $v_i$ to $row_i$.
- (*union*) Suppose $t_1$ and $t_2$ have the same table schema T, $t_1 \cup t_2$ denotes a new table with its schema being T and its rows being the union of rows in $t_1$ and $t_2$.

**Inputs to $\text{BEE}_S$.** The inputs to $\text{BEE}_S(\langle T^i, t^o \rangle, T^p, C)$ comprise the followings:

$T^i$ (*input examples*) is a list of input examples (tables) that model the application domain's data entities. In $T^i = \langle t_1, \ldots, t_n \rangle$, each table $t_x$ is associated with a table name $\eta_x$ and a table schema $\mathrm{T}_x$. Data entities can be modeled by table schemas, e.g., graphic frames can be modeled with columns `file`, `frame`, `id`, etc., as in Figure 1. Entity relations can also be modeled by table schemas, e.g., the parental relation in XML documents can be modeled by $\mathrm{T} = \langle \text{parent} : \texttt{Id},\ \text{child} : \texttt{Id} \rangle$, where each pair denotes a direct inclusion relation between two XML elements.

$t^o$ (*output examples*) models the application domain's example actions performed on $T^i$. $t^o$ has a fixed schema

$$\mathrm{T}^o = \langle \text{action} : \texttt{String},\ arg_1 : \tau_1,\ \ldots \rangle.$$

Specifically, the first column denotes the kind of action to be performed. The remaining columns denote the arguments of the action. For example, if the action is "rename", the first argument should be the file path at present, and the second argument should be the new file path. For concise presentation, this paper assumes the $t^o$ in each invocation of $\text{BEE}_S$ has only one kind of action. In case multiple kinds of user actions were performed (e.g., file removal and renaming), $\text{BEE}_S$ can be invoked independently for each kind of action.

$T^p$ (*pending data*) is a list of tables that are the additional (non-example) inputs that need to be processed by the synthesized program. In $T^p = \langle t_1, \ldots, t_n \rangle$, each $t_x$ has the name $\eta_x$ and schema $\mathrm{T}_x$. Note that $T^p$ should have the same structure as that of the input example $T^i = \langle t'_1, \ldots, t'_n \rangle$, i.e., for any $x$, $\eta_x = \eta'_x$ (name of $t'_x$) and $\mathrm{T}_x = \mathrm{T}'_x$ (schema of $t'_x$). This assures that the program synthesized from $\langle T^i, t^o \rangle$ can be applied to $T^p$.

$C$ (*constant pool*) is a set of constant values (can be empty) as ingredients of predicates synthesized by $\text{BEE}_S$. Our motivating task does not require such constants. Take another task as an example.

| | | | |
|---|---|---|---|
| Program | $P$ | $:=$ | $P_t\ P_m$ |
| Transform Program | $P_t$ | $:=$ | $s_t; P_t \mid \epsilon$ |
| Transform Statement | $s_t$ | $:=$ | $t$ = Filter $(t, \phi)$ |
| | | $\mid$ | $t$ = Join $(t_1, t_2, col_1, col_2)$ |
| | | $\mid$ | $t$ = GroupJoin $(t, col_{index}, (\alpha, col)\ldots)$ |
| | | $\mid$ | $t$ = Order $(t, col, c_{start}, c_{inv})$ |
| | | $\mid$ | $t$ = Order $(t, col, c_{start}, c_{inv}, col_{index})$ |
| Aggregation | $\alpha$ | $:=$ | max $\mid$ min $\mid$ sum $\mid$ avg $\mid$ cnt |
| Predicate | $\phi$ | $:=$ | $(\phi_1 \wedge \phi_2) \mid (\phi_1 \vee \phi_2) \mid \neg\phi$ |
| | | $\mid$ | $ps\ (col_1, col_2) \mid ps\ (col, c) \mid ps\ (col)$ |
| Mapping Program | $P_m$ | $:=$ | $s_m; P_m \mid \epsilon$ |
| Mapping Statement | $s_m$ | $:=$ | Yield $(t, \rho\ldots)$ |
| Projection | $\rho$ | $:=$ | $col \mid c \mid$ Mutate $(f, col\ldots)$ |

Fig. 4. The context-free grammar of $\text{Bee}_L$. $\epsilon$ denotes an empty literal, $f$ denotes a feature (introduced later), $t$ denotes a table variable, $col$ denotes a column name, $c$ denotes a constant value, $ps$ denotes a predicate symbol.

If we want to delete `pdf` files, "pdf" should be included in $C$. The availability of a constant pool facilitates the synthesis procedure, typically reducing wrong search branches when generating "where" clauses. As observed from online forums and suggested by previous SQL synthesis work [53], the constants can be commonly identified from the textural task description. Hence, the software application may ask the end-users to provide such "keywords" in prompt or automatically extract from essential fields in the application domain (e.g., file extension in file management).

**Synthesis Problem.** The output of $\text{Bee}_S(\langle T^i, t^o\rangle, T^p, C)$ is a program $P$ written in a language called $\text{Bee}_L$ such that

$$[\![P(T^i)]\!] = t^o,$$

i.e., $P$ is the generalization of the input-output example $\langle T^i, t^o\rangle$. Applying $P$ to the pending data $T^p$ yields the target user actions $t^s = [\![P(T^p)]\!]$.

### 3.2 $\text{Bee}_L$ Syntax and Semantics

$\text{Bee}_L$ is the meta DSL defining the exact scope of Bee, i.e., the set of programs that can be synthesized to solve PBE tasks.

**Syntax.** The syntax of $\text{Bee}_L$ is defined in Figure 4. $\text{Bee}_L$ is designed based on adding value computation to SQL and aims to reach a balance between practical expressiveness and synthesis tractability. A $\text{Bee}_L$ program $P$ consists of two parts: a transformation program $P_t$ for creating useful intermediate tables followed by a mapping program $P_m$ for projecting intermediate tables to actions. Constants in predicate $\phi$ are from the constant pool $C$, while other constants can be inferred automatically.

**Semantics of Transformation Program.** $P_t$ is a sequence of transformation statements. Each transformation statement $s_t$ generates a new table variable by applying a transformation on existing tables. The following transformations are defined in $\text{Bee}_L$:

*Filter* resembles the "where" clause `SELECT * FROM` $t$ `WHERE` $\phi$ in SQL for applying predicates on rows. Each predicate symbol $ps$ in $\phi$ receives one to two columns/constants and produces a boolean value, e.g., `IntEquals`$(col_1, col_2)$, `IsSubstring`$(col, c)$, `IsOdd`$(col)$. Constants $c$ are picked from the end-user-provided constant pool $C$ (recall that $C$ is input to $\text{Bee}_S$).

t

| x | y |
|---|---|
| 10 | 22 |
| 20 | 33 |
| 30 | 33 |

(a) Origin

Order(t,x,0,false)

| x | y | ord |
|---|---|---|
| 10 | 22 | 0 |
| 20 | 33 | 1 |
| 30 | 33 | 2 |

(b) Order

Order(t,x,0,false,y)

| x | y | ord |
|---|---|---|
| 10 | 22 | 0 |
| 20 | 33 | 0 |
| 30 | 33 | 1 |

(c) Order with index

Mutate(sum(100),x,y)

| x | y | sum |
|---|---|---|
| 10 | 22 | 132 |
| 20 | 33 | 153 |
| 30 | 33 | 163 |

(d) Mutate

Fig. 5. Examples of operators Order and Mutate.

*Join* resembles the "join" clause in SQL for connecting two tables: SELECT * FROM $t_1$ JOIN $t_2$ ON $t_1.col_1 = t_2.col_2$. Unlike SQL, Bee$_L$ only allows joining two columns of `Id` type. We made such a tradeoff because: (1) Eliminating joining on columns of arbitrary type (e.g., strings) significantly reduces the search space. (2) We observe that most join operations needed in practical PBE tasks belong to two categories, one is joining on `Id` columns, and the other is covered by the GroupJoin operator introduced below.

*GroupJoin* aims to resemble the "group" clause in SQL for aggregating values, e.g., SELECT $\alpha_1$ $col_1, \ldots$ FROM $t$ GROUP BY $col_{index}$. In practice, the semantics of GroupJoin is adjusted because we notice a common type of PBE task that needs to join the aggregated values and the values in the original table. A typical example is to extract the filename (original data) of the largest file under each folder (aggregation). To save a join step after aggregation, instead of only producing the aggregated table, GroupJoin produces a table by joining the original table and aggregated table on the index column. That is, it resembles SELECT * FROM $t$ JOIN $g$ = (SELECT $\alpha_1$ $col_1, \ldots$ FROM $t$ GROUP BY $col_{index}$) ON $t.col_{index} = g.col_{index}$ in SQL. Each newly generated column by aggregation is assigned a new name distinct from existing columns.

*Order* resembles the "order" clause SELECT * FROM $t$ ORDER BY $col$ in SQL for sorting rows. However, tables in Bee$_L$ are unordered sets and cannot be sorted.n Alternatively, Bee$_L$ appends a new column to indicate each row's rank sorted by Order.

An example of using Order is shown in Figure 5b, where column `ord` is ordered on column `x`. If an optional column argument $col_{index}$ is provided, the values in $col$ only compare with rows of the same value at $col_{index}$. An example of using such an index column is shown in Figure 5c. Different from Figure 5b, the new value at row 2 is 0 because only row 3 shares the same value with row 2 at column `y`. The new value at row 3 becomes 1 for the same reason.

**Semantics of Mapping Program.** The mapping program $P_m$ consists of a sequence of Yield statements and finally produces the table $t^{out}$ of actions. Suppose the $k$-th Yield statement returns table $t_k$, $t^{out}$ is the union of all Yield results, i.e.,

$$t^{out} = \bigcup t_k.$$

*Yield* takes in a table $t$ and multiple projection expressions $\rho$. Each $\rho_i$ is evaluated as a column $col_i$ before executing Yield, and the evaluation of Yield projects $t$ on columns $col_i$, i.e.,

$$[\![\text{Yield}(t, \rho_1, \ldots, \rho_n)]\!] = t[col_1, \ldots, col_n].$$

Each projection expression $\rho$ computes a column to be projected in Yield. Three types of projections are allowed:

- Column name $col$ is evaluated to be the column $col$ in $t$.
- Constant $c$ is evaluated to be a column filled with constant $c$. Typically, the first $\rho$ expression in a Yield statement must be a constant string denoting the action name, e.g., "shift" in Figure 2.
- Mutate expression is evaluated to be a column, as explained below.

Mutate is for the case where the PBE task may involve domain-specific computations, e.g., arithmetic computations are required for the task in Figure 1. However, these computations are beyond the expressiveness of the commonly used SQL operators (e.g., where, join). To strike a balance between expressiveness (e.g., allowing such computations everywhere, even in $P_t$) and synthesis tractability, we limit such computation in Mutate at the mapping phase.

Specifically, Mutate takes in a *feature* $f$, i.e., a function of type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau_o$, a list of selected columns $col_1 : \tau_1, \ldots, col_n : \tau_n$, and an implicit table $t$ in the Yield statement. The mutation is conducted for each *row* of $t$, yielding a new $\tau_o$-typed column $col_{new}$, denoted by $f(col_1, \ldots, col_n)$, where the value in each row *row* is computed by

$$row[col_{new}] = f(row[col_1], \ldots, row[col_n]).$$

For example, in Figure 5d, the feature $f$ is `sum`(100) that sums two values plus a bias of 100, and the column `sum` is obtained by applying $f$ on columns `x` and `y`.

To make Bee$_L$ applicable to a broad spectrum of PBE tasks, we introduce *higher-order features*, i.e., functions that take parameters as input and output a feature. Bee$_L$ incorporates the following higher-order features.

$$\begin{aligned}
\texttt{substring}(regex) &= \lambda x.\ \text{extract}(x, regex) : \texttt{String}\rightarrow\texttt{String} \\
\texttt{concat}(M) &= \lambda x_1 \ldots \lambda x_n.\ M(x_1, \ldots, x_n) : \texttt{String}^n\rightarrow\texttt{String} \\
\texttt{sum}(b) &= \lambda x.\lambda y.\ x + y + b : \texttt{Int} \times \texttt{Int}\rightarrow\texttt{Int} \\
\texttt{linear}(a, b) &= \lambda x.\ ax + b : \texttt{Int}\rightarrow\texttt{Int} \\
\texttt{div}(b, d) &= \lambda x.\ (x + b) \text{ div } d : \texttt{Int}\rightarrow\texttt{Int} \\
\texttt{mod}(b_1, b_2, d) &= \lambda x.\ (x + b_1) \bmod d + b_2 : \texttt{Int}\rightarrow\texttt{Int}
\end{aligned} \tag{1}$$

These higher-order features are useful in abstracting common PBE tasks and are well-supported by modern constraint solvers [12] or program synthesizers [18]. For example, `div` and `mod` can be used to plot a sequence of $3 \times 2$ blocks to a sequence of $6 \times 1$ blocks. [4] `concat` can simulate a lot of string-like transformations. For example, the movement of files can often be viewed as path string manipulation.

The `substring` and `concat` features are for string manipulation. An extract($x$, *regex*) expression extracts a substring of $x$ specified by a regular expression *regex*, where the scope of *regex* is borrowed from FlashFill [18]. A string manipulation program $M$ is characterized by $m$ regular expressions, and $M(x_1, \ldots, x_n)$ concatenates (denoted by $\oplus$) $m$ substrings:

$$\begin{aligned}
M(x_1, \ldots, x_n) \ &= \ \text{extract}(c_1, regex_1) \oplus \ldots \oplus \text{extract}(c_m, regex_m), \\
&\text{where } c_i \in \{x_1, \ldots, x_n\}.
\end{aligned} \tag{2}$$

In our workflow, these higher-order features are designed to be built in Bee instead of provided by PBE developers. First, solving the feature parameters requires sound background knowledge of SMT and PBE. The requirement violates our goal to make Bee easy to use. Second, Bee's library of higher-order features can be extended by PBE experts, while the users only need to select those they need. In principle, a high-order feature can be added as long as there exists an efficient backbone (i.e., constraint solver or program synthesizer).

### 3.3 Execution and Validity of Bee$_L$ Programs

Bee$_S(\langle T^i, t^o \rangle, T^p, C)$ produces a program $P$ such that $[\![P(T^i)]\!] = t^o$. Bee$_S$ only considers *valid* programs. A Bee$_L$ program is considered valid if no type violation occurs, and all used variables are defined before use during execution.

[4] The `mod` feature only supports a fixed range of common dividends (2 to 10) since an unlimited range goes beyond a constraint solver's capability.

Specifically, the validity check is done statement by statement during execution. The execution of $P_t$ uses a set $\Sigma$ to maintain the tables that have been defined. Suppose that $P(\langle t_1, \ldots, t_n \rangle)$ is executed, $\Sigma$ is initialized as $\Sigma = \{t_1, \ldots, t_n\}$. Each execution of a transformation statement $t = op(\ldots)$ adds the generated table $t$ to $\Sigma$. A transformation statement $t = op(\ldots)$ is valid if the followings hold.

(1) The table variables and column variables used in $s_t$ are defined in $\Sigma$.
(2) The name of the generated table $t$ does not duplicate with the table names in $\Sigma$.
(3) The operations in $op(\ldots)$ are type-checked, e.g., Join requires the two columns to have the same type.

A Yield statement Yield$(t, \rho_1, \ldots, \rho_n)$ is valid if the followings hold.

(1) $t$ and the column variables used in $\rho_1, \ldots \rho_n$ are defined in $\Sigma$.
(2) $\rho_1$ is a constant string that corresponds to a defined action $a$.
(3) The operations in $\rho_1, \ldots \rho_n$ are type-checked.
(4) The types of $\rho_1, \ldots \rho_n$ are consistent with the action arguments of $a$.

## 4 THE BEE SYNTHESIZER

This section illustrates our approach to synthesizing a $\text{Bee}_L$ program $P$ satisfying the given input-output example $T^i$ and $t^o$, i.e., the synthesis problem defined in Section 3.2. Before diving into the algorithm, we first introduce the following challenges under our problem setting if we were to adopt the syntax-guided enumeration-based synthesis algorithm commonly used in SQL query synthesis [52, 53, 60].

### 4.1 Motivation

$P$ consists of $P_t$, a sequence of table operations to create intermediate tables, and $P_m$, a sequence of projections. Therefore, the straightforward baseline algorithm could be:

(1) Synthesizing $P_t$ by maintaining a work list of tables $T$ (initially, $T = T^i$). Each search iteration enumerates all possible Filter/Join/GroupJoin/Order operations on tables in $T$ (Join operates on $(t, t') \in T \times T$) and adds the result tables back to $T$.
(2) To synthesize $P_m$, we can synthesize each Yield statement by deciding whether each $t \in T$ can be projected onto a subset of $t^o$, i.e., whether $\exists \rho_1, \ldots, \rho_n$. such that $[\![\text{Yield}(t, \rho_1 \ldots \rho_n)]\!] \subseteq t^o$. Then, $P_m$ can be obtained by deciding whether there exists a set of Yield statements such that the union of their projected tables is $t^o$.

This baseline algorithm is inefficient because we need to synthesize features' parameters in Equation (1), e.g., regular expression *regex* in `substring`. Brute-force enumeration of parameters (integers and regular expressions) yields a huge search space. Alternatively, a practical approach is to deduce the parameters (e.g., via a solver) given the values this feature is supposed to output. However, this requires enumerating all possible subsets of $t^o$ because the subset's values would decide the deduction results of the parameters. For example, when synthesizing the feature in Figure 2, among the 15 non-empty subsets, only {-30, -40} and {+30, 430} can deduce the correct linear arithmetic expression $A_1$ and $A_2$, respectively. In the worst case, we need to explore $O(2^n)$ subsets of $t^o$, where $n$ is the number of rows in $t^o$ ($n$ can be up to 30 in our evaluation benchmarks). Note that such worst-case frequently happens because the table $t$ we try to find projection ranges over all the intermediate tables in $T$. Many of them do not have projections but still require trying with all the subsets of $t^o$.

In response to this challenge, we observe that we can alternate the search order to *hypothesize* a sub-table of $t^o$ (a subset of rows, denoted by $h \subseteq t^o$) in advance. With effective prioritization of sub-table search (detailed in Section 4.3), the synthesis of a large $t^o$ is divided into solving smaller sub-problems, which greatly scales the problem size we can solve. As reflected in the evaluation,

**Algorithm 1:** Top-level synthesis algorithm.

```
1  Function Synthesize(T^i, t^o)
2  begin
3      s_t ← {⟨ϵ, t⟩ | t ∈ T^i};
4      s_m ← ∅;
5      for d ← 1 to ∞ do
6          s_t ← s_t ∪ Expand(s_t, d);
7          g ← HypothesisGenerator(T^i, t^o);
8          while HasNext(g) do
9              h ← Next(g);
10             s_m ← Match(h, s_t);
11             if s_m ≠ ⊥ then
12                 s_m ← s_m ∪ {⟨s_m, h⟩};
13                 g ← UpdateRank(g, h);
14                 P_m ← AssembleMapping(s_m, t^o);
15                 if P_m ≠ ⊥ then
16                     return AssembleProgram(s_t, P_m);
```

this approach enabled Bee to synthesize programs for the tasks with 20 user actions, while the baseline algorithm timed out for all tasks with more than ten user actions.

### 4.2 Bidirectional Search

The above observation derives a bidirectional search algorithm (Algorithm 1) in our synthesizer:

(1) In the forward searching, we enumerate statements in $P_t$ by syntax, and each one produces an intermediate table.
(2) In the backward searching, we prioritize sub-tables in $t^o$ and try to synthesize a Yield statement for each of them.
(3) The higher-order feature parameters are determined during the table matching from two directions.

**Forward Searching.** The forward searching adopts the syntax-guided enumeration like the baseline algorithm in Section 4.1. Specifically, we maintain a set $\overline{s_t} = \{\langle s_t, t\rangle\}$ where each element contains a transform statement $s_t$ and a result table $t$. $\overline{s_t}$ is initialized with empty statements and tables from $T^i$ (line 3).

The forward searching enumerates statements in order of *depth*. We assign depth to each statement $s_t$ (and its corresponding result table $t$) as follows. Empty statement $\epsilon$ and tables from $T^i$ have depth 0. The depth of a non-empty statement $s_t$ with operator $op$ and operand tables $\bar{t}$ is $max_{t\in\bar{t}}\text{depth}(t) + \text{depth}(op)$. For operator Filter, depth($op$) is the number of predicate symbols in the predicate, and for other operators, depth is 1.

Specifically, the depth budget $d$ is increased in each iteration (line 5), and the Expand($\overline{s_t}, depth$) procedure in line 6 enumerates all tables of depth $d$ by enumerating operators and tables in $\overline{s_t}$.

**Backward Searching.** The backward searching synthesizes the mapping program $P_m$. We maintain a set $\overline{s_m} = \{\langle s_m, h\rangle\}$ for each synthesized mapping statement $s_m$ and the sub-table $h$ it outputs. Specifically, we use a structure *HypothesisGenerator* (detailed in Section 4.3) to decompose $t^o$ into a

prioritized and bounded queue of sub-tables (*hypotheses*) [5], where each element $h \subseteq t^o$ (lines 7–9 of Algorithm 1). Then, each iteration chooses a hypothesis $h$ and matches it against the intermediate tables in $\overline{s_t}$ (line 10). If a successful mapping is found and a mapping statement $s_m$ is returned, we add the synthesized $s_m$ and hypothesis $h$ to $\overline{s_m}$ (lines 10–12) and update the ranking in the hypothesis generator accordingly (line 13).

**Table Matching.** The forward and backward searching meet in the middle via table matching by $\textsc{Match}(h, \overline{s_t})$ (line 10), which verifies whether a hypothesis $h$ can be produced by projecting some table from $\overline{s_t}$.

Specifically, $\textsc{Match}(h, \overline{s_t})$ iterates each table $t$ in $\overline{s_t}$, and checks whether there exist projection expressions $\rho \ldots$ such that $[\![\text{Yield}(t, \rho \ldots)]\!] = h$. To ease the illustration, we introduce a table $t_\square$ abstracting all columns that can be projected from $t$. Specifically, $t_\square$ is obtained by appending new columns to $t$, where each new column corresponds to a constant projection or mutate projection.

$$t_\square = t + \sum_c col_c + \sum_{f_h} \sum_{\overline{col}} f_h(\overline{a})(\overline{col})$$

Here, $\sum$ denotes appending a sequence of columns. The constant $c$ ranges over the values $c$ in $h$ such that there exist a column in $h$ having only one value $c$. $col_c$ denotes a column filled with constant $c$. $f_h$ ranges over the higher-order features in Equation (1) and $\overline{a}$ denotes the *free variables* to parameterize $f_h$, e.g., *regex* for `substring`. $\overline{col}$ ranges over the column combinations in $t$ that are type consistent with the feature $f = f_h(\overline{a})$. Recall that $f(\overline{col})$ denotes the column obtained from Mutate $(f, \overline{col})$.

For each higher-order feature $f_h$, we introduce free variables $\overline{a}$ to denote the parameterized feature $f_h(\overline{a})$ and also the values obtained by evaluating $f_h(\overline{a})$ on columns $\overline{col}$. For example, consider a column with values $\langle 2, 3\rangle$, and the higher-order feature is `linear` that accepts parameters $(a, b)$ and outputs a feature $\lambda x.ax + b$. If we introduce free variables $a_1$ and $b_1$, the parameterized feature is denoted by $\lambda x.a_1 * x + b_1$, and the resulting column is denoted by $\langle 2a_1 + b_1, 3a_1 + b_1\rangle$.

With this representation of $t_\square$, we can model the matching between $h$ and $t_\square$ as a constraint-solving problem. To ease the illustration, we assume the rows and columns in $t_\square$ and $h$ are indexed. Suppose $t_\square$ is an $m \times n$ table, $h$ is an $m' \times n'$ table ($m \geq m' \wedge n \geq n'$), our target is to find a surjection $r(i) : [1 \ldots m] \rightarrow [1 \ldots m']$ from rows of $t_\square$ to rows of $h$, and a mapping $c(j) : [1 \ldots n'] \rightarrow [1 \ldots n]$ from columns of $h$ to columns of $t_\square$, such that

$$\exists\ r,\ c,\ \theta. \bigwedge_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n'}} t_\square[row_i, col_{c(j)}] = h[row_{r(i)}, col_j], \tag{3}$$

where $\theta$ is the set of free variables.

To solve the constraint, we first enumerate the mappings $r$ and $c$, then solve the values in $\theta$. Specifically, for integral constraints, we solve the integer parameters with an off-the-shelf constraint solver (e.g., Z3 [12] in our implementation). For string constraints, we solve the string manipulation program by adopting the algorithm in FlashFill [18].

**Assembling Program.** Once a hypothesis is checked feasible (line 11), $\textsc{AssembleMapping}(\overline{s_m}, t^o)$ checks whether the found hypotheses (sub-tables) form a union equal to $t^o$ and outputs the corresponding mapping program $P_m$ (line 14). If $P_m$ is found, then $\textsc{AssembleProgram}(\overline{s_t}, P_m)$ generates $P_t$ from the tables $P_m$ relies on (select statements from $\overline{s_t}$) and returns $\langle P_t, P_m\rangle$ as the final result (lines 15–16).

[5] In our evaluation, the bound is 20, i.e., we try synthesizing with at most 20 hypotheses for each depth.

Theorem 1. ***(Soundness)*** *Given input-output examples $\langle T^i, t^o \rangle$, and $P$ is generated from the algorithm Synthesize($\langle T^i, t^o \rangle$), then $P$ is consistent with $\langle T^i, t^o \rangle$, i.e., $[\![P(T^i)]\!] = t^o$.*

Theorem 2. ***(Completeness)*** *Suppose the bound of hypotheses is unlimited, given input-output examples $\langle T^i, t^o \rangle$ such that $\exists P. [\![P(T^i)]\!] = t^o$, Synthesize($\langle T^i, t^o \rangle$) must return a program. In practice, with the limited bound, Bee$_S$ is not complete, i.e., it may not return a program even if a consistent program exists.*

### 4.3 Optimizations

**Ranking Heuristics.** The number of hypotheses to search in backward searching is exponential to the number of rows in $t^o$, i.e., the number of elements in the power set of $t^o$. In our evaluating PBE tasks, the number of rows (user actions) could be up to 30, which makes the number of subsets too massive for brute-force enumeration. Therefore, we adopt heuristics-based ranking to prioritize the hypotheses to search.

To better illustrate the heuristics, we introduce one more PBE task in Figure 6 that aims to categorize Pass/Fail scores. In the following, we illustrate our heuristics with Figure 6.

The first heuristic is that we assume the hypotheses generated by different Yield statements have no intersection. This assumption usually would not miss the correct programs because, in practice, user actions produced by different Yield statements are for processing different data entities. For example, the two Yield statements in our motivating task are for processing odd and even frames separately. Also, the task in Figure 6 processes each spreadsheet row separately.

On the other hand, if the overlap between hypotheses is allowed, the synthesized program is likely to be overfitting. A typical case of over-fitting is that the program comprises many Yield statements, each covering a small subset of $t^o$. When a hypothesis has fewer rows, it is more likely to find a projection that satisfies Equation (3), even though it is not the user's expectation. A naive case is that when there is only one row in a hypothesis, the corresponding Yield statement can be simply filled with constant projections. A program composed of all such Yield statements is likely to be over-fitting and what we try to circumvent.

Our second heuristic is that if $t$ is one of the target sub-table in $t^o$, then the values in each column of $t$ should be from the same projection. For example, among the four user actions in Figure 1, −30 and −40 are computed from $A_1$, while +30 and +40 are computed from $A_2$. Based on this intuition, the hypothesis generator ranks the sub-tables of $t^o$ with the consistent score defined in Figure 7.

For each column in a sub-table $t$, we check if it can be a constant column ($S_{const}$) or an order column ($S_{ord}$) or a concatenation column ($S_{concat}$). By instinct, these conditions check whether the values in the column are from the same projection. For example, the `content` columns of $t_1^\rho$ and $t_2^\rho$

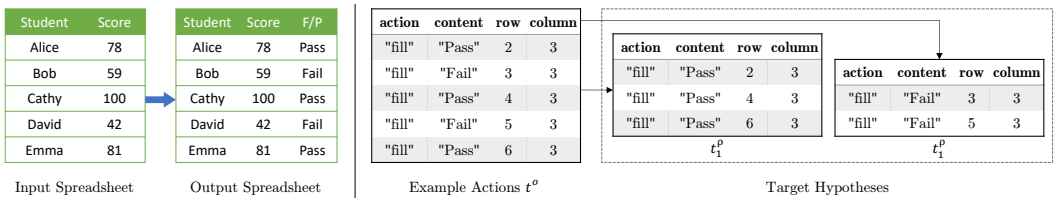



Fig. 6. Example PBE task for illustrating the heuristics. Given a spreadsheet of students and scores, the task needs to classify scores >= 60 as "Pass" and < 60 as "Fail" and append the status to the third column. The left-hand side shows five examples. $t^o$ denotes the example actions. $t_1^\rho$ and $t_2^\rho$ denote the target hypotheses that should be ranked high by the heuristics.

$$S(t) = \sum_{col \in t} [S_{const}(col) + S_{ord}(col) + S_{concat}(col)] * rows(t)$$

$$S_{const}(col) = \begin{cases} 1, & \text{if all values in } col \text{ are identical} \\ 0, & \text{otherwise} \end{cases}$$

$$S_{ord}(col) = \begin{cases} 1, & \text{if the values in } col \text{ are consecutive integers} \\ 0, & \text{otherwise} \end{cases}$$

$$S_{concat}(col) = \begin{cases} 1, & \text{if the strings in } col \text{ can be projected by a } \texttt{concat} \text{ feature on some rows in } T^i \\ 0, & \text{otherwise} \end{cases}$$

Fig. 7. Equations measuring the consistency score $S(t)$ of table $t$. $rows(t)$ denotes the number of rows in $t$.

are captured by $S_{ord}$. $S(t)$ is proportional to the number of rows in $t$ because more rows tend to reduce the probability of over-fitness, as we discussed above. Nonetheless, over-fitness may also happen, e.g., $t^o$ as a whole is scored high because the `row` column happens to be consecutive.

We implemented the above heuristics in a *hypothesis generator* (line 7 in Algorithm 1). A hypothesis generator takes in the input example $T^i$ and the output table $t^o$, ranks the sub-tables of $t^o$, and iterates them in order of ranking scores. In addition to the consistency score, we optimize the ranking result using the following measures:

- If a table $t$ is a subset of a table $t'$ previously iterated and $t'$ has been matched, rank down $t$. This measure prevents the subsets of a top-ranked table from dominating the top positions because if $t'$ has high scores in $S_{const}, S_{ord}$, and $S_{concat}$, its subsets also tend to have high scores. For example, when $t_1^\rho$ has been matched, the two-row subsets of $t_1^\rho$ may also be ranked high but more attention should be paid to tables with other rows, e.g., $t_2^\rho$.
- If a table $t$ is matched successfully with a table generated forward, the generator would rank the complement sub-tables higher (line 13 in Algorithm 1). This measure is effective if some target sub-tables fail to get high scores. For example, $t_2^\rho$ only has two rows and is ranked lower than some wrong sub-tables, e.g., the sub-tables with `row` number "2-3-4", "3-4-5", and "4-5-6". But when $t_1^\rho$ is matched successfully, we could prioritize the matching of $t_2^\rho$, which is the complement of $t_1^\rho$.

**Implementation.** We briefly introduce other optimizations in our implementation here. First, we noticed that the tables generated in the forward searching have many duplicates because different predicates used in the filter construct may produce the same tables. In our implementation, we merged predicates and tables into equivalent classes and only considered one in the search process. We also noticed many same columns could be produced from the GroupJoin and Order construct, and we merged the equivalent columns.

Second, the solving of constraints usually contains a lot of common sub-constraints. We cached the previously computed results to avoid re-invoking the same costly constraint-solving processes.

Third, as some features preserve the order after transformation (e.g., linear, divide, and sum), we can prune the search by matching and solving the constraints instead of enumerating all the possible permutations.

Fourth, when generating columns from Order, we do not explicitly enumerate all parameters, we fix $c_{start} = 0$ and $c_{reverse} = false$ to generate columns, then leverage the feature-solving procedure to synthesize the actual parameters.

## 5 EVALUATION

In this section, we evaluate the performance and usability of Bee with the following research questions:

- **RQ1: (Effectiveness of Bee)** How effective is Bee in solving PBE tasks in multiple domains? How is it compared to implementing known DSLs and algorithms on the state-of-the-art PBE frameworks (an alternative for developers to realize PBE for their applications)?
- **RQ2: (Effectiveness of backward searching)** How does the backward searching component contribute to the effectiveness of Bee?
- **RQ3: (Usability)** Compared with the existing PBE framework, does Bee make PBE more approachable for software developers?

To evaluate the effectiveness of Bee, we collected PBE tasks in three different domains as benchmarks and evaluated the success rate of different techniques (i.e., the proportion of tasks that can be synthesized within a timeout). To answer RQ1, we compared the PBE tools implemented on top of Bee with those implemented on top of a state-of-the-art PBE framework, Prose [26, 42]. To answer RQ2, we compared two versions of Bee with and without using the heuristic hypothesis generator. To answer RQ3, we conducted a human study where participants were asked to implement two PBE tools based on Bee and Prose and then report their assessment of the frameworks' usability in a survey.

### 5.1 RQ1 & RQ2: Effectiveness of Bee

*5.1.1 Experiment Setup.* In this section, we explain our experiment setup to answer RQ1 and RQ2.

**Data Collection.** To evaluate the effectiveness of Bee, we collected 64 benchmarks in the domains of file management (22), spreadsheet transformation (22), and XML transformation (20). These three domains were chosen because (1) they represent different forms of data: set, tabular, and hierarchical; (2) PBE tasks in these domains can be obtained from the literature or online forums. Each benchmark is a PBE task that contains a task description and input/output examples. The benchmarks are available on our project website [37].

We collected the benchmarks from existing studies and online forums. All 22 file management benchmarks are extracted from an earlier study [59]. However, we have no access to the original evaluation dataset, so we recovered a dataset from their task descriptions. Two tasks were excluded due to their ambiguous descriptions. As for spreadsheet transformation, seven benchmarks were collected from examples in earlier studies [5, 21]. Similar to file management benchmarks, we recovered the benchmarks from the task descriptions in the papers. We further augmented the benchmarks with another 15 PBE tasks collected from an online forum, Excel Forum [34]. All the 20 XML transformation benchmarks were collected from an online forum OxygenXML [36]. We collected these benchmarks from the latest posts on the forums when we conducted the experiments. Posts were selected when they contained a clear description and specific input/output examples. For each benchmark, we manually inspect the task and specify the values in constant pools.

**Baselines.** To answer RQ1 and RQ2, we compared Bee with two baselines, respectively:

Prose: To answer RQ1, we compared Bee with Prose. Prose is a state-of-the-art PBE framework developed and maintained by Microsoft. The framework serves as backends for different PBE applications [18, 41, 45]. It allows developers to develop PBE tools by providing DSLs and functions to optimize the synthesis algorithms of Prose with domain-specific knowledge. Prose is a general-purpose framework that supports arbitrary DSL and algorithms, which differs from Bee. The comparison aims to assess the relative merits of Bee against a simulated scenario of using a SOTA framework (i.e., Prose) to implement known DSLs and algorithms (detailed below).

Table 1. Success Rates and Over-fitting Rates of Different PBE Techniques for Benchmarks in Three Domains

| Benchmarks | Success Rate | | | Over-fitting Rate | | |
|---|---|---|---|---|---|---|
| | Bee | Prose | Bee-f | Bee | Prose | Bee-f |
| File management | 20/22 (90.9%) | 20/22 (90.9%) | 21/22 (95.5%) | 1/20(5.0%) | 4/20(20.0%) | 1/21(4.8%) |
| Spreadsheet transformation | 16/22 (72.7%) | 15/22 (68.2%) | 7/22 (31.8%) | 3/16(25.0%) | 7/15(46.7%) | 2/7(28.6%) |
| XML transformation | 17/20 (85.0%) | 12/20 (60.0%) | 18/20 (90.0%) | 3/17(17.6%) | 0/12(0.0%) | 3/18(16.7%) |
| Total | 53/64 (82.8%) | 47/64 (73.4%) | 46/64 (71.9%) | 7/53(13.2%) | 11/47(23.4%) | 6/46(13.0%) |

Bee-f: To answer RQ3, we introduced Bee-f, a variant of Bee without the backward searching component, i.e., the baseline algorithm introduced in Section 4.1. All the other optimizations are preserved.

**PBE Tool Implementations.** Both Bee and Prose are frameworks allowing a PBE tool to be constructed with additional customization by developers. The following customization is adopted in our evaluation.

*Customization on Bee.* As mentioned, the 64 benchmarks collected span across three application domains. We, therefore, prepare a PBE tool for each domain by customizing Bee with the concerned data schema and user actions in tables. The customization is made using the C# API interface provided by Bee.

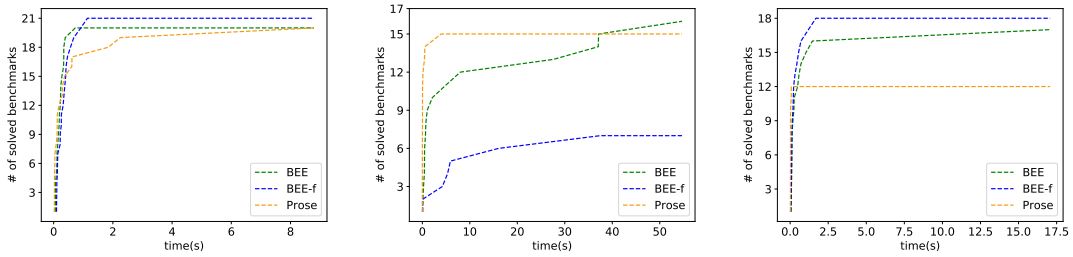
*Customization on Prose.* We customize Prose to implement three state-of-the-art DSL-based domain-specific PBE tools as baselines. For each tool, we provide a grammar file specifying the DSL, a set of files expressing the semantics of the DSL elements, and callback functions. Specifically, the DSLs and algorithms for file management and XML transformation were extracted from Hades [59]. The DSL and algorithm for spreadsheet transformation were extracted from a study on synthesizing spreadsheet layout transformation [21]. While other spreadsheet transformation synthesizers [7, 16] exist, they are not completely DSL-based and therefore are not considered.

**Evaluation Metrics.** We evaluated the performance of a synthesizer by measuring: (a) the success rate: the proportion of benchmarks that can be solved within a timeout (120s), i.e., programs matching the input/output examples were successfully generated, (b) the time taken to solve a benchmark, and (c) the number of over-fitting solutions. A solution to a benchmark is over-fitting if the synthesized program does not capture the intent of the benchmark. We checked whether a synthesized program is over-fitting by manually comparing the semantics of the program against the description in each benchmark.

All the experiments were conducted on a Windows machine with a 4GHz AMD 5800X CPU and 32GB memory, simulating the ordinary computing power of end-users. The timeout of 120s is sufficient concerning the response time limit (10s) for keeping the user's attention [39] and the time limit (60s) for the user to complete simple tasks [22].

*5.1.2 Experiment Results.* Table 1 shows the success rate of Bee and the baseline techniques in the three domains. Within the timeout (120 seconds), the PBE tools powered by Bee were able to solve 53 out of the total 64 (82.8%) benchmarks, outperforming that of Prose and Bee-f (73.4% and 71.9%, respectively). Moreover, all the benchmarks successfully solved by Bee were finished within one minute, and 48 benchmarks (75.0%) were finished within 10 seconds (Figure 8). Among the successfully solved benchmarks, the longest time taken in file management, spreadsheet transformation, and XML transformation benchmarks is 0.7s, 53.7s, and 15.5s, respectively.

We manually inspected all the synthesized programs and found that seven of the synthesized programs over-fitted the provided examples. Bee over-fitted the benchmarks because they did




(a) File management. (b) Spreadsheet transformation. (c) XML transformation.

Fig. 8. Experiment results on benchmarks under different algorithm settings. The $x$ axis indicates the time (seconds) taken to solve the benchmarks. The $y$ axis indicates the accumulated number of benchmarks that can be solved. The green lines denote Bee's tools, the blue lines denote Bee-f's tools, and the orange lines denote Prose's tools.

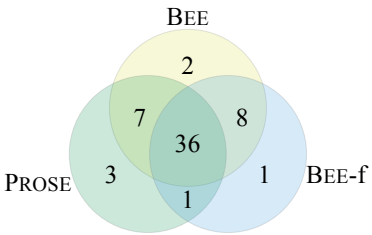



Fig. 9. Venn diagram of benchmarks solved by each technique. For example, 36 benchmarks were solved by both the three techniques.

not provide sufficient input/output examples, e.g., three of them only contained one input/output example. We further provided examples for these seven benchmarks, and six over-fitting programs were corrected.

We studied the remaining 11 failed benchmarks to inspect the reasons for the failures. Five of them cannot be well expressed by our current language, and two of them timed out due to their intrinsic complexity. Typically, the target programs of these benchmarks involve feature computation mixed with complex structural transformations such as join. The remaining four cases cannot be captured by the optimization heuristics we proposed. Figure 9 shows the Venn Diagram plotting the relationship between benchmarks that can be solved by each technique. As shown in the figure, among the 11 benchmarks failed by Bee, five of them can be solved by Prose or Bee-f. Two benchmarks can be solved by Bee-f but not Bee. Our backward searching component failed to decompose appropriate sub-tables for these benchmarks because their output examples were simple and indistinguishable in projection. This enlightens us to improve the strategy further to generate hypotheses in backward searching by considering more information from the input. As for Prose, two of the three uniquely solved benchmarks contain operations beyond the expressiveness of Bee, and the remaining one was too complex to be solved by Bee's synthesis algorithm. In the future, we plan to further improve the expressiveness of Bee by incorporating more constructs and further optimizing our synthesis algorithm. In the following, we compare Bee with the two baselines in detail and answer RQ1 and RQ2.

**Comparison with Prose.** Bee outperformed Prose with regard to both the success rate and the over-fitting rate. As shown in Table 1, overall, Bee achieved a higher success rate (82.8% v.s. 73.4%) and a lower over-fitting rate (13.2% v.s. 23.4%). Prose had success rates similar to Bee in the domains of file management and spreadsheet transformation but had more over-fitting cases in both of these domains (4 v.s. 1 and 7 v.s. 3). Prose had higher over-fitting rates in these domains because the domain-specific algorithms described in their papers did not incorporate ranking strategies [21, 59]. Thus, over-fitting solutions were first synthesized and outputted in these cases. In the domain of XML transformation, Prose had no over-fitting cases but had a success rate much lower than Bee. This is because many tasks in the domain of XML transformation are out of the scope of the domain-specific DSL in Prose adapted from Hades [59]. Eight of our XML transformation tasks violate the assumption adopted by the DSL and cannot be expressed. In comparison, Bee successfully synthesized five of these tasks. This shows that the expressiveness of Bee is satisfying. Many real-world XML transformation tasks out of the scope of existing domain-specific DSLs can still be supported by Bee.

Figure 8 plots the number of solved benchmarks as time passes. The green line represents Bee, and the orange line represents Prose. The performance of Bee is similar to that of Prose. Prose grows faster than Bee only in the spreadsheet transformation domain. This is because the synthesizer built atop Prose considers a more limited search space optimized for spreadsheet transformation. Bee cannot leverage such domain-specific optimization since we aim to hide the sophisticated design of DSL and synthesis algorithms by adopting a more inclusive language. However, the evaluation results show that Bee achieved a good performance comparable to Prose even without domain-specific optimizations. This shows that the bidirectional synthesis algorithm we proposed helped Bee to synthesize the target programs effectively.

**Comparison with Bee-f.** As shown in Table 1, Bee achieved a higher overall success rate and a similar over-fitting rate compared with Bee-f. Specifically, Bee significantly outperformed Bee-f in the domain of spreadsheet transformation (72.7% v.s. 31.8%). In the domains of file management and XML transformation, Bee had s slightly worse performance than Bee-f: Bee-f solved one more benchmark in both domains. Bee failed to solve these two benchmarks because the output examples in these benchmarks were small-size ($\leq$ six rows in $t^o$) and indistinguishable. The backward searching component of Bee failed to decompose the output examples into sub-tables correctly and misled the forward searching. Instead, the brute-force forward searching adopted by Bee-f remained manageable due to the relatively small number of output rows. However, in the domain of spreadsheet transformation, the benchmarks are more complicated, with usually more than ten rows in $t^o$. Bee-f timed out for most tasks because the forward searching cannot scale without the guidance of the backward searching. These results show that our bi-directional searching strategy can improve the effectiveness of the synthesizer, especially for complicated PBE tasks. To mitigate the issue induced by failures of the backward searching, we can adopt an adaptive strategy that falls back to using only forward searching for simple tasks with a limited number of output examples.

To summarize, PBE tools implemented on top of Bee achieved better performance than those implemented with the state-of-art PBE framework and integrated domain-specific optimizations. Our results also show that the bidirectional search strategy adopted by Bee can improve the effectiveness of the synthesis algorithm, especially for complicated tasks.

### 5.2 RQ3: Usability

In RQ3, we aim to evaluate whether Bee can ease the process of implementing PBE tools for software developers. To achieve this, we conducted a human study with 12 participants and compared the usability of Bee and Prose.

**Participants.** We recruited 12 participants who have four to ten years of coding experience. They have different levels of familiarity with context-free grammar (CFG, a premise of DSL design in Prose). Three participants are unfamiliar, six have a basic understanding, and three have used CFG in their projects. Regarding their SQL familiarity, one is unfamiliar, four have a basic understanding, and seven have used SQL in their projects. This confirmed our assumption that SQL is a common skill among programmers. All the participants have no prior experience in PBE. They are suitable human study subjects because the target developers of Bee are those with mature programming skills but limited PBE-specific knowledge. All participants received the same compensation after the human study.

**Methodology.** In our human study, we asked our participants to implement PBE tools with Bee and Prose. It is impractical for the participants to implement a complete PBE tool in laboratory settings since it can take hours. Confounding factors like participants' level of concentration may arise during the human study. To address this problem, we first implemented several PBE tools and removed parts of them to form programming tasks. The participants only need to finish the programming tasks instead of implementing complete PBE tools. We prepared PBE tasks as test cases to examine whether a participant's implementation is correct or not. An implementation is considered correct if it can synthesize programs matching our intended PBE tasks (i.e., passing the PBE test cases).

Each participant used both Bee and Prose and rated the difficulties in learning and using both frameworks. A participant used the two frameworks to implement PBE tools for two domains (file management and spreadsheet transformation), respectively. These two domains were chosen so that the experience in using one framework will not affect the experiment results of using the other. However, since the difficulty in implementing PBE tools for the two domains can be different, we divided the participants into two groups: Group 1 used Bee to implement file management PBE tools ($F_B$) and Prose to implement spreadsheet transformation ($S_P$). Group 2 used Prose to implement file management PBE tools ($F_P$) and Bee to implement spreadsheet transformation ($S_B$). We compared the human study results of the two groups to assess the impact of the difficulties of the two domains on participants' performance. We did not divide the participants into two groups and let each group use and rate only one PBE framework because the criteria to evaluate usability can vary across different participants, and their ratings may not be comparable.

Participants carried out the tasks in different timeslots with no overlap. Each participant was required to read tutorials of Bee and Prose (with framework names anonymized) before participating in the study to gain a basic understanding of the two frameworks. The on-site human study started with a discussion with the participant to solve her/his puzzles on the tutorials, followed by a quiz to ensure the participant has a correct and sufficient understanding before doing the programming tasks. For each programming task, we introduced the design of the PBE tool, which portions were missed, and the PBE tasks (test cases) it needed to solve. A participant either finished the PBE tool and passed all the test cases within a timeout of 30 minutes or was considered to fail in the task. After completing two programming tasks, the participant took a survey to report her/his feelings about Bee and Prose.

The human study was conducted in a university and approved by the university's Human Participants Research Panel. No personally identifiable information was recorded.

**Results.** We report the time spent by participants to finish programming tasks in Table 2. The participants mostly spent less time on the programming tasks with Bee than that of Prose in both groups. Although the removed portions in both frameworks have similar sizes and complexities (usually a variable, predicate, or a single line of code), understanding the context in Prose projects may consume more time. Only $P_4$ and $P_8$ spent more time on Bee than Prose. However, $P_4$ finished

Table 2. Time Spent on Programming Tasks

| Group 1 | $P_1$ | $P_3$ | $P_5$ | $P_7$ | $P_9$ | $P_{11}$ | Avg |
|---|---|---|---|---|---|---|---|
| $F_B$ | 8 | 3 | 10 | 20 | 18 | 15 | 12.33 |
| $S_P$ | 24 | 8 | 13 | 27 | 22 | 24 | 19.67 |
| Group 2 | $P_2$ | $P_4$ | $P_6$ | $P_8$ | $P_{10}$ | $P_{12}$ | Avg |
| $S_B$ | 25 | >30 | 7 | 9 | 25 | 13 | 18.17 |
| $F_P$ | 30 | 30 | 12 | 5 | >30 | 13 | 20.00 |

All times are in minutes. Each $P_i$ denotes a participant. Avg shows the average minutes and is calculated by taking numbers > 30 as 30.

Table 3. Ranks of Difficulties to Learn/Use Frameworks

| Learn | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ | $P_8$ | $P_9$ | $P_{10}$ | $P_{11}$ | $P_{12}$ | Avg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bee | 2 | 2 | 2 | 4 | 1 | 2 | 2 | 1 | 3 | 3 | 2 | 2 | 2.17 |
| Prose | 4 | 5 | 3 | 5 | 4 | 4 | 2 | 4 | 3 | 3 | 4 | 4 | 3.75 |
| Use | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ | $P_8$ | $P_9$ | $P_{10}$ | $P_{11}$ | $P_{12}$ | Avg |
| Bee | 1 | 2 | 2 | 3 | 1 | 2 | 2 | 2 | 2 | 3 | 2 | 2 | 2.00 |
| Prose | 4 | 3 | 3 | 4 | 3 | 4 | 3 | 3 | 3 | 3 | 4 | 3 | 3.34 |

Ranks are: 1 (very easy), 2 (easy), 3 (median), 4 (difficult) or 5 (very difficult). Each $P_i$ denotes a participant. Avg is the average rating.

the programming task of Prose ($F_P$) in 30 minutes, but the implementation hardcoded a few conditions and over-fitted our test cases. As for $P_8$, the participant had a good understanding of both frameworks and was able to complete the programming tasks fast, except that the participant asked about a concept specific to $S_B$ (instead of Bee) during the human study. The discussion took four to five minutes.

In the survey, participants were asked to rate the difficulties of learning/using Prose/Bee (Table 3). Nine out of 12 participants ranked Bee easier to learn than Prose, and 11 ranked Bee easier to use than Prose. To investigate whether there is a statistical difference between the difficulty measurements of the two frameworks (Bee v.s. Prose), we adopt the Wilcoxon signed-rank test [58]. This non-parametric paired difference test is suitable for paired samples with a small size. The test result on the samples of difficulties to learn (resp., use) Bee and Prose shows statistical significance ($p < 0.01$) with the exact $p = 0.007$ (resp., $p = 0.002$). We received four similar comments like *"Bee is easier to get started with while Prose better suits PBE experts"*. These results show that Bee is easier to learn and use than Prose for software developers without a PBE background.

We also surveyed which part of Prose the participants deemed the most difficult. Most participants found the components related to the design of DSL and synthesis algorithms the most difficult: Seven found witness functions (functions to customize synthesis algorithms in Prose) the most difficult to understand, and three identified DSL-related components the most difficult. Another participant indicated that Prose is difficult to use because it requires much PBE background knowledge. Our communications with the participants also show supporting evidence of these difficulties. Before starting the programming tasks, we had a briefing to clarify their puzzles on the tutorials of the two frameworks. All the participants had spent more time in the briefing on questions for Prose, and half of them (six) expressed that they could not understand the purpose of witness functions.

In contrast, no participants reported that the database-style interface of Bee caused their confusion. These observations align with our motivation: (1) the major obstacle for software developers without a PBE background to implement PBE tools is the design of DSLs and synthesis algorithms, and (2) existing PBE frameworks like Prose are still difficult to use for such software developers. To conclude, although Bee still requires domain-specific customization, the database-style interface lowers the barrier to PBE tool implementation compared with the traditional interface that requires DSL and synthesis algorithm.

## 6 DISCUSSION

### 6.1 Threats to Validity

**Performance.** The performance of Prose in our experiments can be affected by the design of DSLs and the implementation of synthesis algorithms in each domain. To address the threat, we borrowed the DSLs and synthesis algorithms from representative synthesizers of each domain [21, 59]. Since the artifacts are not publicly available, we re-implemented the DSLs and synthesis algorithms according to the descriptions in the papers. We also made the tools as compatible with our benchmarks as possible, e.g., by adding language elements as long as they do not require changing the synthesis algorithms.

For Bee, its performance can vary depending on the design of the table schemas. In our evaluation, the table schemas were designed to suit the general tasks in each domain while not over-fitting the tasks in the benchmarks (e.g., we never hard coded any properties that are shortcuts to the desired ones in specific tasks). Specifically, one of the authors designed the table schemas for the three evaluating domains, and the other authors validate the design.

Another threat to performance comparison is that specifications (e.g., languages, input-outputs) to synthesizers on Bee and Prose are different, the difficulties of synthesizing programs and being not over-fitted are also different. Some tasks can be solved in the Prose specification more easily than in the Bee specification, or sometimes the other way around, as reflected in Figure 9. However, the purpose of RQ1 is not to strictly compare two algorithms with the same input specification, but to assess whether Bee as a whole can perform comparatively well against its alternative, i.e., implementing known DSLs and algorithms on Prose. To this end, we have tried our best to make the synthesizers on Prose compatible with our benchmarks without changing the synthesis algorithms, which simulate the alternative scenarios. We have also made our benchmarks and implementations open-sourced [37] for reproduction.

**Usability.** In our usability study, the participants only completed parts of the synthesizer implementation as programming tasks. For example, both the DSLs of Prose and table schemas of Bee were provided to the users in our study. We did not ask the participants to implement a complete PBE tool because it is likely to take more time than affordable for in-lab studies. As such, participants can underestimate the difficulties of implementing synthesizers based on Prose and Bee. However, we asked the participants to learn the whole process of implementing PBE synthesizers based on both frameworks in our tutorials. This can help them understand and assess the overall difficulties in PBE synthesizer implementations.

We asked each participant to complete two PBE tools based on each framework respectively in one experiment. There is a risk that participants may learn some knowledge of PBE implementation from their first task, influencing the second task. To mitigate the influence, we divided the participants into two groups and altered the order of the frameworks used in our formal experiment. We did not find substantial differences in the assessments of the two groups of participants. This is because the approaches to implementing PBE tools on top of Prose and Bee are divergent. Besides, the two

tasks they need to solve fall into two different domains. As a result, the knowledge learned from one framework had a minor impact on the other one.

### 6.2 Expressiveness

With the design goal to maximize the ease of developer interface, Bee adopts the scheme of a predefined meta DSL with customizable entities/actions. Consequently, Bee is not as generally applicable as other frameworks that allow DSL customization (e.g., Prose). The expressiveness of the meta DSL (i.e., Bee$_L$) is limited purposefully to make the synthesis procedure tractable. This also makes Bee inapplicable to certain tasks. Although formally describing a language's expressiveness is difficult [30], we could discuss the expressiveness of Bee$_L$ by comparing it with its basis, a commonly used SQL subset (i.e., select, where, join, group, order) that has been studied [30] extensively.

A fundamental limitation in expressiveness inherited by Bee$_L$ is recursive query, which queries results from itself as a recursive function calls itself. For example, given a set of edges $E$ in a graph, query the reachability closure $R$, i.e., node pairs $(a, b)$ such that there is a path from $a$ to $b$. The recursive query can be $(a, b) \in R$ if and only if $(a, b) \in E$ or there exists $c$ such that $(a, c) \in E$ and $(c, b) \in R$ . Despite recursive queries that require an unbounded number of nested queries, some tasks requiring a large (but bounded) number of queries are also difficult for Bee to synthesize in practice, although possible in theory. For example, given a code repository's abstract syntax trees (ASTs), we could represent the hierarchical structure by introducing a `parent` field to each AST node. However, to extract AST subtrees that match a specific pattern, the target Bee$_L$ program may need many joining steps to recover a subtree and compare it with the target pattern. Synthesizing such programs is likely to fail due to the overwhelming search space.

The major difference brought by Bee is the adoption of features in the Yield statements (i.e., the mapping program $P_m$). This largely strengthens the expressiveness of Bee$_L$ compared with the base SQL subset, particularly when handling PBE tasks, which generally require computations such as string manipulations. Before $P_m$, the tables that could be expressed/queried in the transformation program $P_t$ are nearly identical to those queried by the base SQL subset. Then, in $P_m$, we can perform *row-wise* computations to query columns (or called features) that cannot be expressed in the base SQL subset. Such computations would not change the original table structure, i.e., no rows would be added/removed, but only new columns would be added. The expressiveness of these row-wise computations is defined by the set of features, specifically, integral arithmetics and string manipulations in the current version. In a nutshell, the expressiveness of Bee$_L$ could be viewed as the common SQL queries followed by row-wise feature computations.

Bee$_L$ is restrained but can also automate a wide range of repetitive tasks. Many existing PBE tools [4, 21, 27, 32, 44, 54, 59] share de facto similar expressive power with different predicates/features. Besides, the interface of relational tables for capturing entities/actions is general and compatible with any language that can generate actions from entities, which allows us to improve the expressiveness of Bee$_L$ in the future without affecting extant tools built on Bee.

### 6.3 Future Work

To enhance the expressiveness of Bee and let it support more PBE tasks, the underlying search space at the synthesis stage could be even larger. While the current synthesis algorithm works well for simple cases, it may not scale to complex ones, e.g., those involving four or more nested filter/join/group operators. We plan to further improve the algorithm with techniques such as abstraction [15, 55], search prioritization [9], and data-driven library learning [10, 14] to improve the performance of Bee. Another direction is to leverage interactive synthesis to alleviate the

synthesis pressure with additional end-user inputs, e.g., hints on the entity fields related to the task. The number of interactions could be minimized by adapting existing techniques [24, 46, 47].

Besides, the inclusive data-action model in Bee has the potential to support cross-domain PBE tasks (e.g., finding intersected items between a webpage and a spreadsheet). In the future, we will explore leveraging Bee to synthesize PBE tasks that involve data and actions in multiple domains.

## 7 RELATED WORK

**Domain-specific PBE Techniques.** In the recent decade, PBE has gained increasing popularity and has found applications in various domains such as spreadsheet [18, 21, 49], information extraction [4, 27, 44], data visualization [54], version control [41], text/code edition [17, 32], etc [11, 29]. Typically, to make the search space of the programs to synthesize tractable, each application focuses on a DSL (defined by the researchers) capturing the specific needs of each domain. Domain-specific techniques often adopt synthesis algorithms tailored for the DSLs to optimize the synthesis performance. Take the classic string manipulation DSL [18] for example. To synthesize $f$ for $f(\text{“foo”}, \text{“bar”}) = \text{“FooBar”}$, the synthesis algorithm searches for solutions to sub-problems, e.g., $f_1(\text{“foo”}, \text{“bar”}) = \text{“Foo”}$, $f_2(\text{“foo”}, \text{“bar”}) = \text{“Bar”}$, etc. The sub-solutions are then composed into a complete solution, e.g., $f = \lambda x.\texttt{concat}(f_1(x), f_2(x))$. This divide-and-conquer approach is tailored for the commonly used `concat` operator in the string manipulating DSL and may not apply to other DSLs. Likewise, specialized DSLs and synthesis techniques are generally hard to be transferred from an existing domain to another one.

**Customizable PBE Frameworks.** Several program synthesis frameworks [31, 42, 51] have been proposed to facilitate the development of program synthesis tools. They provide facilities (e.g., APIs) to help developers define a DSL (syntax and semantics) and ship a default synthesis algorithm. These frameworks are designed for experts with sufficient background knowledge on program synthesis. For example, to use Rosette [51], the user needs to understand what operators have support in SMT background theories [6] and design a compliant DSL because the synthesis algorithm in Rosette is based on SMT solving. To use Prose [42], the user needs to understand its synthesis algorithm and design a compliant DSL where each production is a function call, e.g., $A \rightarrow f(B, C)$. Usually, the synthesis algorithm also requires the user to provide callback functions, i.e., functions invoked by the synthesis engine to decompose constraints on $f(B, C)$ into constraints on $B$ and $C$. Tuning the DSL or synthesis algorithm is arduous for non-PBE-experts. In comparison, Bee provides a developer-friendly interface to make PBE more approachable for PBE novices. Our evaluation also shows that Bee is easier to learn and use than existing PBE frameworks.

**Table-based Program Synthesis.** There are many synthesis techniques based on table-like data structures, including SQL query synthesis by examples [52, 53, 60, 61] and by natural language [13, 38], table schema refactoring [57], database access model synthesis [46, 47], Datalog-like program synthesis [43, 48, 50], and other transformations on tables beyond SQL [7, 15, 16, 54]. These techniques cannot synthesize complex expressions (e.g., with constants) as we discussed in Section 4. There are several recent works that aim to synthesize such complex expressions [8, 56, 61]. They all require extra specifications inputted from users, e.g., sketches of expressions or demonstration of the computation procedure instead of only the final value. Such specifications can only be provided by spreadsheet/database users who have a good understanding of the desired programs. We cannot introduce such specifications to Bee because the majority of applications (e.g., file manager, E-mail client) that Bee may support are facing end-users without a programming background. We design a bidirectional algorithm for a simplified setting where the expression only exists in the last step. This design does not require additional inputs from end users and achieved good performance in practice, as demonstrated in our evaluation.

**Synthesis Technique.** The synthesis algorithm of Bee leverages bidirectional search and divide-and-conquer. Bidirectional search is an effective approach to exploring the program space (e.g., programs defined in a DSL) with respect to the given specification (e.g., examples). Generally speaking, given a specification in the form of examples ($\xi_{in} = \{i_1, \ldots i_n\}, \xi_{out} = \{o_1, \ldots, o_n\}$), to find $P$ such that $P(i_j) = o_j$ for each $j$, bidirectional search (1) explores the programs and states that derive from $\xi_{in}$ (forward searching), (2) explores the programs and specifications that derive from $\xi_{out}$ (backward searching), and (3) composes $P$ using specifications explored from the two directions. Forward searching usually exploits syntax-guided enumeration of the DSL to produce concrete programs and running states, while the form of specification in backward searching varies from application to application. For example, Duet [28] targets SyGuS [1] programs and recursively deduces the output specification (initially $\xi_{out}$) of functions $f(x_1, \ldots)$ to produce specifications for sub-expressions $(x_1, \ldots)$. Viser [54] targets table transformation programs and leverages table-inclusion constraints (tables explored forward need to subsume the tables explored backward) to prune search space. Different from them, Bee searches for concrete sub-tables of the target table in the backward searching to support synthesizing the complex value computations that are required in many Bee tasks. The backward searching process (i.e., searching sub-tables) in Bee follows a divide-and-conquer pattern, which was also leveraged by the STUN (synthesis through unification) framework [2, 3] to solve the problem of synthesizing nested `if-then-else` expressions. STUN synthesizes programs in a bottom-up manner, which first syntactically enumerates expressions for each atomic branch until the expressions cover all input-output examples and finally unifies the expressions using decision-tree learning. In contrast, Bee decomposes sub-tables in a top-down manner and matches with tables generated forward to synthesize the Yield statement. As pointed out by a recent work [25], STUN is likely to synthesize a program by unifying many branches, each covering a small number of examples. If the constraints of the desired program are given by examples, such programs are likely to be over-fitting, as we analyzed in Section 4.3. In contrast, the ranking strategies of Bee prefer larger sub-tables and thus bias not to synthesize such over-fitting programs.

## 8 CONCLUSION

This paper introduces Bee, a PBE framework making PBE approachable for software developers without PBE expertise. We formulated Bee's interface, where relational tables serve as a unified representation to model PBE tasks, and developers can easily build PBE tools by customizing the table schemas specific to their needs. We introduced a DSL $\text{Bee}_L$ that combines table transformation and non-trivial value computation to express PBE tasks. We designed a bidirectional synthesis algorithm that effectively synthesizes $\text{Bee}_L$ programs. Our evaluation in three different domains showed that Bee could achieve good performance comparable to that of domain-specific synthesizers. Moreover, our human study revealed that, compared to a state-of-the-art PBE framework, Bee is easier to use and learn for non-PBE-expert developers. In the future, we will further improve Bee by enhancing the expressiveness of the language and the performance of the synthesis algorithm.

## ACKNOWLEDGMENTS

We thank the editors and the anonymous reviewers for their constructive comments and suggestions. We also thank the participants in the human study and undergraduate students who helped explore the applications of Bee.

This project is supported by the National Science Foundation of China (Nos. 61932021 and 62272218), the Leading-edge Technology Program of the Jiangsu Natural Science Foundation under Grant (No. BK20202001), the Hong Kong Research Grant Council/General Research Fund (Grant No. 16207120), the Hong Kong Research Grant Council/Research Impact Fund (Grant No. R503418),

and the Natural Sciences and Engineering Research Council of Canada Discovery Grant (Grant Nos. RGPIN-2022-03744 and DGECR-2022-00378).

This project is also supported by the Fundamental Research Funds for the Central Universities of China (020214912220). The authors would like to express their gratitude for the support from the Xiaomi Foundation and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

## REFERENCES

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[2] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 319–336.

[3] Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification (CAV 2015)*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

[4] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: Web Automation by Demonstration. *SIGPLAN Not.* 51, 10 (Oct. 2016), 748–764. https://doi.org/10.1145/3022671.2984020

[5] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. *SIGPLAN Not.* 50, 6 (June 2015), 218–228. https://doi.org/10.1145/2813885.2737952

[6] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13. 14.

[7] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 168 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360594

[8] BavishiRohan, LemieuxCaroline, SenKoushik, and StoicaIon. 2021. Gauss: program synthesis by reasoning over graphs. *Proceedings of the ACM on Programming Languages* (Oct. 2021). https://doi.org/10.1145/3485511 Publisher: ACM PUB27 New York, NY, USA.

[9] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: probabilistic model for code. In *International Conference on Machine Learning*. PMLR, 2933–2942.

[10] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2023. Top-Down Synthesis for Library Learning. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 41:1182–41:1213. https://doi.org/10.1145/3571234

[11] Salman Cheema, Sarah Buchanan, Sumit Gulwani, and Joseph J LaViola Jr. 2014. A practical framework for constructing structured drawings. In *Proceedings of the 19th international conference on Intelligent User Interfaces*. 311–316.

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[13] Li Dong and Mirella Lapata. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, Iryna Gurevych and Yusuke Miyao (Eds.). Association for Computational Linguistics, 731–742. https://doi.org/10.18653/v1/P18-1068

[14] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3453483.3454080

[15] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. *SIGPLAN Not.* 53, 4 (June 2018), 420–435. https://doi.org/10.1145/3296979.3192382

[16] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. *SIGPLAN Not.* 52, 6 (June 2017), 422–436. https://doi.org/10.1145/3140587.3062351

[17] Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, and Ashish Tiwari. 2020. Feedback-driven semi-supervised synthesis of program transformations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 219:1–219:30. https://doi.org/10.1145/3428287

[18] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.
[19] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010
[20] Daniel Conrad Halbert. 1984. *Programming by example.* Ph. D. Dissertation. University of California, Berkeley.
[21] William R. Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. *PLDI* 47 (2011), 317. https://doi.org/10.1145/2345156.1993536
[22] Steve Henty. 2015. UI Response Times. https://medium.com/@slhenty/ui-response-times-acec744f3157 Accessed: 2023.
[23] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press.
[24] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 1143–1158. https://doi.org/10.1145/3385412.3386025
[25] JiRuyi, XiaJingtao, XiongYingfei, and HuZhenjiang. 2021. Generalizable synthesis through unification. *Proceedings of the ACM on Programming Languages* (Oct. 2021). https://doi.org/10.1145/3485544 Publisher: ACM PUB27 New York, NY, USA.
[26] Ashwin Kalyan, Abhishek Mohta, Alex Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *6th International Conference on Learning Representations (ICLR)* (6th international conference on learning representations (iclr) ed.). https://www.microsoft.com/en-us/research/publication/neural-guided-deductive-search-real-time-program-synthesis-examples/
[27] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. *SIGPLAN Not.* 49, 6 (June 2014), 542–553. https://doi.org/10.1145/2666356.2594333
[28] Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 54:1–54:28. https://doi.org/10.1145/3434335
[29] Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive parser synthesis by example. *ACM SIGPLAN Notices* 50, 6 (2015), 565–574.
[30] Leonid Libkin. 2003. Expressive power of SQL. *Theoretical Computer Science* 296, 3 (March 2003), 379–404. https://doi.org/10.1016/S0304-3975(02)00736-3
[31] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An Extensible Synthesis Framework for Data Science. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1914–1917. https://doi.org/10.14778/3352063.3352098
[32] Anders Miltner, Sumit Gulwani, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 143:1–143:29. https://doi.org/10.1145/3360569
[33] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012
[34] n.d. 2023. Excel Forum. https://www.excelforum.com/excel-general Accessed: 2023.
[35] n.d. 2023. ImageMagick. https://imagemagick.org/script/command-line-tools.php Accessed: 2023.
[36] n.d. 2023. OxygenXML. https://www.oxygenxml.com/forum Accessed: 2023.
[37] n.d. 2023. Bee Homepage. https://github.com/Sissel-Wu/Bee Accessed: 2023.
[38] Ansong Ni, Pengcheng Yin, and Graham Neubig. 2020. Merging Weak and Active Supervision for Semantic Parsing.. In *AAAI.* 8536–8543.
[39] Jakob Nielsen. 2014. Response Times: The 3 Important Limits. https://www.nngroup.com/articles/response-times-3-important-limits/ Accessed: 2023.
[40] Elizabeth J. O'Neil. 2008. Object/Relational Mapping 2008: Hibernate and the Entity Data Model (Edm). In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08).* Association for Computing Machinery, New York, NY, USA, 1351–1356. https://doi.org/10.1145/1376616.1376773
[41] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike Kaufman. 2021. Can Program Synthesis be Used to Learn Merge Conflict Resolutions? An Empirical Analysis *(ICSE 2021).* IEEE Computer Society, 785–796. https://doi.org/10.1109/ICSE43902.2021.00077 ISSN: 1558-1225.
[42] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. *OOPSLA* 50, 10 (2015), 107–126. https://doi.org/10.1145/2858965.2814310
[43] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2019. Provenance-guided synthesis of Datalog programs. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 62:1–62:27. https://doi.org/10.1145/3371130

[44] Mohammad Raza and Sumit Gulwani. 2017. Automated Data Extraction Using Predictive Program Synthesis.. In *AAAI*. 882–890.

[45] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*. IEEE Press, Buenos Aires, Argentina, 404–415. https://doi.org/10.1109/ICSE.2017.44

[46] Jiasi Shen and Martin C. Rinard. 2019. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3314221.3314591

[47] Jiasi Shen and Martin C. Rinard. 2021. Active Learning for Inference and Regeneration of Applications that Access Databases. *ACM Transactions on Programming Languages and Systems* 42, 4 (Jan. 2021), 18:1–18:119. https://doi.org/10.1145/3430952

[48] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-guided synthesis of Datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 515–527. https://doi.org/10.1145/3236024.3236034

[49] Rishabh Singh and Sumit Gulwani. 2016. Transforming Spreadsheet Data Types Using Examples. *SIGPLAN Not.* 51, 1 (Jan. 2016), 343–356. https://doi.org/10.1145/2914770.2837668

[50] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-guided synthesis of relational queries. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1110–1125. https://doi.org/10.1145/3453483.3454098

[51] Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

[52] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by Output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 535–548. https://doi.org/10.1145/1559845.1559902

[53] Chenglong Wang, Alvin Cheung, and Ras Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. *PLDI* (2017), 452–466.

[54] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by example. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

[55] Xinyu Wang, Greg Anderson, Isil Dillig, and K. L. McMillan. 2018. Learning Abstractions for Program Synthesis. In *Computer Aided Verification (CAV 2018)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 407–426. https://doi.org/10.1007/978-3-319-96145-3_22

[56] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 62:1–62:26. https://doi.org/10.1145/3133886

[57] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing Database Programs for Schema Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3314221.3314588

[58] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968

[59] Navid Yaghmazadeh and Christian Klinger. 2016. Synthesizing Transformations on Hierarchically Structured Data. *PLDI* (2016), 508–521.

[60] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings* (2013), 224–234. https://doi.org/10.1109/ASE.2013.6693082

[61] Xiangyu Zhou, Rastislav Bodik, Alvin Cheung, and Chenglong Wang. 2022. Synthesizing analytical SQL queries from computation demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 168–182. https://doi.org/10.1145/3519939.3523712

## A RUNNING EXAMPLE SPECIFICATION

In this section, we detail the specifications of the running example in Figure 1 in case readers find some parts of the illustration unclear due to the simplification. To clarify, the GUI and the

entity/action schemas in this example are hypothesized for presenting the idea of Bee and not implemented.

**Entities.** In this task, each frame is an entity. A frame could be modeled by many fields such as a unique identifier *id*, the order in the frame sequence $frame$, the source file of the figure $file$, the width/length of the figure, etc. To make the illustration simple, we only consider the following schema with three fields.

$$\mathrm{T} = \langle id : \texttt{Id},\ frame : \texttt{Int},\ file : \texttt{String} \rangle$$

The schema can be specified with C# annotation APIs, as shown in Listing 1.

**Actions.** In this task, we only consider a shift action with the following schema.

$$\mathrm{T}^o = \langle action : \texttt{String},\ id : \texttt{Id},\ channel : \texttt{String},\ bx : \texttt{Int},\ by : \texttt{Int} \rangle$$

Specifically, the *action* must be "shift" for Bee to recognize. Executing an action would move the *channel*s (specified by strings such as "GB" and "RGB") of a frame (specified by *id*) $bx$ pixels right (or left if $bx$ is negative) and $by$ pixels down (or up if $by$ is negative). The schema can be specified with C# annotation APIs, as shown in Listing 2.

**Input-Output Examples.** The input examples are the first four frames. They can be modeled by T as the following table $t^i$, and $T^i = \{t^i\}$.

| file | frame | id |
|---|---|---|
| "tiktok.jpg" | 1 | f1 |
| "tiktok.jpg" | 2 | f2 |
| "tiktok.jpg" | 3 | f3 |
| "tiktok.jpg" | 4 | f4 |

The output examples are the four shift actions applied on the first four frames. The example output table $t^o$, modeled by $\mathrm{T}^o$, is as follows.

| action | id | channel | bx | by |
|---|---|---|---|---|
| "shift" | f1 | "GB" | -30 | -30 |
| "shift" | f2 | "GB" | +30 | +30 |
| "shift" | f3 | "GB" | -40 | -40 |
| "shift" | f4 | "GB" | +40 | +40 |

**Target Program.** The target $\text{Bee}_L$ program for handling the running example is as follows.

```
u = Filter(ti, isOdd(frame));
v = Filter(ti, isEven(frame));

Yield("shift", u, id, "GB", linear(-5, -25)(frame), linear(-5, -25)(frame));
Yield("shift", v, id, "GB", linear(5, 20)(frame), linear(5, 20)(frame));
```

Before executing the program, a state $\Sigma$ to store table variables is initialized with $\{ti\}$, where $ti$ is bounded to $t^i$. When executing the first statement `u = Filter(ti, isOdd(frame));`, a table $u$ as follows is produced by filtering rows whose frames are odd numbers. $\Sigma$ is updated as $\{ti, u\}$.

| file | frame | id |
|---|---|---|
| "tiktok.jpg" | 1 | f1 |
| "tiktok.jpg" | 3 | f3 |

When executing the second statement `u = Filter(ti, isEven(frame));`, a table $v$ as follows is produced by filtering rows whose frames are even numbers. $\Sigma$ is updated as $\{ti, u, v\}$.

| file | frame | id |
|---|---|---|
| "tiktok.jpg" | 2 | f2 |
| "tiktok.jpg" | 4 | f4 |

Finally, when executing the last two `Yield` statements, the following two tables $t_1^\rho$ and $t_2^\rho$ would be generated as sub-tables of the final program output.

| action | id | channel | bx | by |
|---|---|---|---|---|
| "shift" | f1 | "GB" | -30 | -30 |
| "shift" | f3 | "GB" | -40 | -40 |

| action | id | channel | bx | by |
|---|---|---|---|---|
| "shift" | f2 | "GB" | +30 | +30 |
| "shift" | f4 | "GB" | +40 | +40 |

The final output table is the union of tables generated from `Yield`, e.g., the above two tables.

| action | id | channel | bx | by |
|---|---|---|---|---|
| "shift" | f1 | "GB" | -30 | -30 |
| "shift" | f2 | "GB" | +30 | +30 |
| "shift" | f3 | "GB" | -40 | -40 |
| "shift" | f4 | "GB" | +40 | +40 |

**Synthesis Procedure.** The target program is synthesized in a bidirectional way.

In the forward direction, transform statements are enumerated according to the syntax in Figure 4 with the depth starting from 1. The first two `Filter` statements (and the corresponding tables $u$ and $v$) are enumerated when the depth is 1. During the synthesis, a set $\overline{s_t}$ is used to maintain the enumerated statements/tables, e.g., $\{\langle \epsilon, ti\rangle, \langle \texttt{Filter(...)}, u\rangle, \langle \texttt{Filter(...)}, v\rangle, \ldots\}$.

In the backward direction, the expected output table is decomposed into sub-tables using the hypothesis generator introduced in Section 4. In this running example, the generator ranks and generates top scored tables including the two sub-tables $t_1^\rho$ and $t_2^\rho$. Each sub-table corresponds to a `Yield` statement and is matched against tables in $\overline{s_t}$.

We illustrate the matching process considering two tables $u$ and $v$ in the forward direction and two tables $t_1^\rho$ and $t_2^\rho$ in the backward direction. Before matching $u$ against $t_1^\rho$, we first prepare an abstract table $t_\square$ by adding to $u$ constant columns in $t_1^\rho$ and parameterized features, which is shown as follows. $a$ and $b$ are integer variables to be resolved.

| file | frame | id | c1 | c2 | linear |
|---|---|---|---|---|---|
| "tiktok.jpg" | 1 | f1 | "shift" | "GB" | $a+b$ |
| "tiktok.jpg" | 3 | f3 | "shift" | "GB" | $3a+b$ |

Then, we can enumerate the mappings from rows in $t_\square$ to rows in $t_1^\rho$ and mappings from columns in $t_1^\rho$ to columns in $t_\square$. The correct mappings can be viewed as follows.

| c1 | id | c2 | linear | linear | … |
|---|---|---|---|---|---|
| "shift" | f1 | "GB" | $a+b$ | $a+b$ | … |
| "shift" | f3 | "GB" | $3a+b$ | $3a+b$ | … |

$\rightarrow$

| action | id | channel | bx | by |
|---|---|---|---|---|
| "shift" | f1 | "GB" | -30 | -30 |
| "shift" | f3 | "GB" | -40 | -40 |

✓

With the mapping, we try solving Equation (3) by leveraging off-the-shelf solvers, e.g., SMT solver in this case to solve constraints $a+b=-30 \wedge 3a+b=-40$. Finally, we found the correct mappings between rows/columns and the parameter values $a=-5 \wedge b=-25$. The corresponding `Yield` statement can be generated accordingly.

The trial of matching $u$ against $t_2^\rho$ fails because we cannot find any `Id`-type value in $u$ that can be mapped to f2 or f4 in $t_2^\rho$.

| file | frame | id |
|---|---|---|
| "tiktok.jpg" | 1 | f1 |
| "tiktok.jpg" | 3 | f3 |

$\rightarrow$

| action | id | channel | bx | by |
|---|---|---|---|---|
| "shift" | f2 | "GB" | +30 | +30 |
| "shift" | f4 | "GB" | +40 | +40 |

✗

Similarly, we can find a mapping statement that can map from $v$ to $t_2^\rho$ but no statement from $u$ to $t_2^\rho$. Since we have found successful mappings from $\overline{s_t}$ to $t_1^\rho$ and $t_2^\rho$, and the union of $t_1^\rho$ and $t_2^\rho$ is exactly the expected output, we can assemble the program accordingly.

## B PBE TOOL IMPLEMENTATIONS

We briefly introduce how the PBE tools in the three evaluating domains are implemented on Bee in this section. Each subsection corresponds to one domain. For each domain, we introduce the

table schemas used to model the data entities and the user actions. Readers may find full details on the homepage of Bee [37].

### B.1 File Management

Each file entity is modeled as a row in the table. The fields are listed as follows.

| Field | Type |
|---|---|
| id | `Id` |
| basename | `String` |
| extension | `String` |
| path | `String` |
| size | `Int` |
| modification_time | Time |
| readable | Boolean |
| writable | Boolean |
| executable | Boolean |
| group | `String` |
| year | `Int` |
| month | `Int` |
| day | `Int` |
| year_s | `String` |
| month_s | `String` |
| day_s | `String` |

Different actions have different schemas (arguments). The actions and arguments they take are listed as follows.

| Action | Arguments |
|---|---|
| chmod | id: `Id`, mod: `String` |
| copy | id: `Id`, path: `String` |
| unzip | id: `Id`, path: `String` |
| move | id: `Id`, path: `String` |
| rename | id: `Id`, name: `String` |
| delete | id: `Id` |
| chgrp | id: `Id`, group: `String` |
| chext | id: `Id`, extension: `String` |
| tar | id: `Id`, name: `String` |

### B.2 Spreadsheet

Given a spreadsheet file and a consecutive range of cells, we model the cells with two tables of different schemas. The first table is obtained by modeling each cell as a row with the following fields. The type of row_head, col_head, content depends on the type of spreadsheet cell. The read_ord field is the reading order of cells (left-right and up-down).

| Field | Type |
|---|---|
| id | `Id` |
| row | `Int` |
| col | `Int` |
| row_head | `String` \| `Int` |
| col_head | `String` \| `Int` |
| content | `String` \| `Int` |
| read_ord | `Int` |

The second table is close to the original tabular representation of a spreadsheet range. As shown in the schema below, each row of the cells corresponds to a row in the output table, and each column of the cells corresponds to a column in the output table. The type of value in column col$i$ depends on the type of the $i$-th column of spreadsheet.

| Field | Type |
|---|---|
| row | `Int` |
| col1 | `String` \| `Int` |
| col2 | `String` \| `Int` |
| | ... |

The only user action is to fill a cell (given by row and column) with content. Deleting a cell is equivalent to filling it with empty content.

| Action | Arguments |
|---|---|
| fill | content: `String`, row: `Int`, col: `Int` |

### B.3 XML

Given an XML file and a list of selected elements, we represent them with two SQL tables using the schemas below, one for XML elements and one for XML attributes. Each XML element is modeled as a row in the former table with its id, its tag, its text, and the ids of its parent, its previous sibling, and its next sibling. [6] Each XML attribute is modeled a row in the latter table with its id, the XML element it belongs to, its key, and its value.

| Field | Type |
|---|---|
| id | `Id` |
| tag | `String` |
| text | `String` |
| parent | `Id` |
| previous | `Id` |
| next | `Id` |

| Field | Type |
|---|---|
| id | `Id` |
| element | `Id` |
| key | `String` |
| value | `String` |

The actions used are listed as follows. These actions generally follow APIs in manipulating XML elements and attributes. However, when used in the context of XML editing, end-users may need to provide additional information, e.g., which action does the editing correspond to.

| Action | Arguments |
|---|---|
| delete_element | element: `Id` |
| modify_text | element: `Id`, text: `String` |
| modify_attribute | element: `Id`, value: `String` |
| modify_tag | element: `Id`, tag: `String` |
| add_element | parent: `Id`, tag: `String`, text: `String` |
| add_element_above | element: `Id`, tag: `String`, text: `String` |
| add_attribute | element: `Id`, key: `String`, value: `String` |
| wrap | element: `Id`, tag: `String` |
| move_below | element: `Id`, target: `Id` |
| append_child | element: `Id`, target: `Id` |

[6] If there is no parent/previous/next element, replace with a special null id.