CLX: Towards verifiable PBE data transformation

Zhongjun Jin[1]   Michael Cafarella[1]   H. V. Jagadish[1]   Sean Kandel[2]
Michael Minar[2]   Joseph M. Hellerstein[2,3]
[1]University of Michigan, Ann Arbor   [2]Trifacta Inc.   [3]UC Berkeley
{markjin,michjc,jag}@umich.edu,{skandel,mminar}@trifacta.com,hellerstein@berkeley.edu

ABSTRACT

Effective data analytics on data collected from the real world usually begins with a notoriously expensive pre-processing step of data transformation and wrangling. Programming By Example (PBE) systems have been proposed to automatically infer transformations using simple examples that users provide as hints. However, an important usability issue—**verification**—limits the effective use of such PBE data transformation systems, since the verification process is often effort-consuming and unreliable.

We propose a data transformation paradigm design CLX (pronounced "clicks") with a focus on facilitating verification for end users in a PBE-like data transformation. CLX performs pattern clustering in both input and output data, which allows the user to verify at the pattern level, rather than the data instance level, without having to write any regular expressions, thereby significantly reducing user verification effort. Thereafter, CLX automatically generates transformation programs as regular-expression replace operations that are easy for average users to verify.

We experimentally compared the CLX prototype with both FLASHFILL, a state-of-the-art PBE data transformation tool, and TRIFACTA, an influential system supporting interactive data transformation. The results show improvements over the state of the art tools in saving user verification effort, without loss of efficiency or expressive power. In a user study on data sets of various sizes, when the data size grew by a factor of 30, the user verification time required by the CLX prototype grew by 1.3× whereas that required by FLASHFILL grew by 11.4×. In another user study assessing the users' understanding of the transformation logic — a key ingredient in effective verification — CLX users achieved a success rate about twice that of FLASHFILL users.

1 INTRODUCTION

Data transformation, or data wrangling, is a critical pre-processing step essential to effective data analytics on real-world data and is widely known to be human-intensive as it usually requires professionals to write ad-hoc scripts that are difficult to understand and maintain. A *human-in-the-loop* Programming By Example (PBE) approach has been shown to reduce the burden for the end user: in projects such as FLASHFILL [6], BLINKFILL [24], and FOOFAH [11], the system synthesizes data transformation programs using simple examples the user provides.

Problems — Most of existing research in PBE data transformation tools has focused on the "system" part — improving the efficiency and expressivity of the program synthesis techniques. Although these systems have demonstrated some success in efficiently generating high-quality data transformation programs for real-world data sets, **verification**, as an indispensable interaction procedure in PBE, remains a major bottleneck within existing PBE data transformation system designs. The high labor cost, may deter the user from confidently using these tools.

Any reasonable user who needs to perform data transformation should certainly care about the "correctness" of the inferred transformation logic. In fact, a user will typically go through rounds of "verify-and-specify" cycles when using a PBE system. In each interaction, a user has to verify the correctness of the current inferred transformation logic by validating the transformed data instance by instance until she identifies a data instance mistakenly transformed; then she has to provide a new example for correction. **Given a potentially large and varied input data set, such a verification process is like "finding a needle in a haystack" which can be extremely time-consuming and tedious.**

A naïve way to simplify the cumbersome verification process is to add explanations to the transformed data so that the user does not have to read them in their raw form. For example, if we can somehow know the desired data pattern, we can write a checking function to automatically check if the post-transformed data satisfies the desired pattern, and highlight data entries that are not correctly transformed.

However, a *data explanation* procedure alone can not solve the entire verification issue; the undisclosed transformation logic remains untrustworthy to the end user. Users can at best verify that existing data are converted into the right form, but **the logic is not guaranteed to be correct and may function unexpectedly on new input** (see Section 2 for an example). Without good insight into the transformation logic, PBE system users cannot tell if the inferred transformation logic is correct, or when there are errors in the logic, they may not be able to debug it. **If the user of a traditional PBE system lacks good understanding of the synthesize program's logic, she can only verify it by spending large amounts of time testing the synthesized program on ever-larger datasets**.

Naïvely, previous PBE systems can support *program explanation* by presenting the inferred programs to end users. However, these data transformation systems usually design their own Domain Specific Languages (DSLs), which are usually sophisticated. The steep learning curve makes it unrealistic for most users to quickly understand the actual logic behind the inferred programs. Thus, besides more explainable data, a desirable PBE system should be able to present the transformation logic in a way that most people are already familiar with.

Insight — Regular expressions (regexp) have been known to most programmers of various expertise and regexp replace operations have been commonly applied in data transformations. The influential data transformation system, WRANGLER (later as TRIFACTA), proposes simplified natural-language-like regular expressions which can be understood and used even by non-technical data analysts. This makes regexp replace operations a good choice for an *explainable transformation language*. The challenge then is how to automatically synthesize regexp replace operations as the desired transformation logic in a PBE system.

A regexp replace operation takes in two parameters: an *input pattern* and a *replacement function*. Suppose an input data set is given, and the desired data pattern can be known, the challenge is to determine a suitable input pattern and the replacement function to convert all input data into the desired pattern. Moreover, if the input data set is heterogeneous with many formats, we need to find out an unknown set of such input-pattern-and-replace-function pairs.

Pattern profiling can be used to discover clusters of data patterns within a data set that are are useful to generate regular replace operations. Moreover, it can also serve as a data explanation approach helping the user quickly understand the pre- and post-transformation data which reduces the verification challenge users face in PBE systems.

Proposed Solution — In this project, we propose a new data transformation paradigm, CLX, to address the two specific problems within our claimed verification issue. The CLX paradigm has three components: two algorithmic components—*clustering* and *transformation*—with an intervening component of *labeling*. In this paper, we present an instantiation of the CLX paradigm. We present (1) an efficient pattern clustering algorithm that groups data with similar structures into small clusters, (2) a DSL for data transformation, that can be interpreted as a set of regular expression replace operations, (3) a program synthesis algorithm to infer desirable transformation logic in the proposed DSL.

Through the above means, we are able to greatly ameliorate the usability issue in verification within PBE data transformation systems. Our experimental results show improvements over the state of the art in saving user verification effort, along with increasing users' comprehension of the inferred transformations. Increasing comprehension is highly relevant to reducing the verification effort. In one user study on a large data set, when the data size grew by a factor of 30, the CLX prototype cost 1.3× more verification time whereas FlashFill cost 11.4× more verification time. In a separate user study accessing the users' understanding of the transformation logic, CLX users achieved a success rate about twice that of FlashFill users. Other experiments also suggest that the expressive power of the CLX prototype and its efficiency on small data are comparable to those of FlashFill.

Organization — After motivating our problem with an example in Section 2, we discuss the following contributions:

- We define the data transformation problem and present the PBE-like CLX framework solving this problem. (Section 3)
- We present a data pattern profiling algorithm to hierarchically cluster the raw data based on patterns. (Section 4)
- We present a new DSL for data pattern transformation in the CLX paradigm. (Section 5)
- We develop algorithms synthesizing data transformation programs, which can transform any given input pattern to the desired standard pattern. (Section 6)
- We experimentally evaluate the CLX prototype and other baseline systems through user studies and simulations. (Section 7)

We explore the related work in Section 8 and finish with a discussion of future work in Section 9.

2 MOTIVATING EXAMPLE

Bob is a technical support employee at the customer service department. He wanted to have a set of 10,000 phone numbers in various formats (as in Figure 1) in a unified format of "(xxx)

Figure 1: Phone numbers with diverse formats

Figure 2: Patterns after transformation

Figure 3: Pattern clusters of raw data

```
1  Replace '/^\(({digit}{3})\)({digit}{3})\-({digit}{4})$/'
       in column1 with '($1) $2-$3'
2  Replace '/^({digit}{3})\-({digit}{3})\-({digit}{4})$/' in
       column1 with '($1) $2-$3'
3  ...
```

Figure 4: Suggested data transformation operations

xxx-xxxx". Given the volume and the heterogeneity of the data, neither manually fixing them or hard-coding a transformation script was convenient for Bob. He decided to see if there was an automated solution to this problem.

Bob found that Excel 2013 had a new feature named FlashFill that could transform data patterns. He loaded the data set into Excel and performed FlashFill on them.

Example 2.1. Initially, Bob thought using FlashFill would be straightforward: he would simply need to provide an example of the transformed form of each ill-formatted data entry in the input and copy the exact value of each data entry already in the correct format. However, in practice, it turned out not to be so easy. First, Bob needed to carefully check each phone number entry deciding whether it is ill-formatted or not. After obtaining a new input-output example pair, FlashFill would update the transformation results for the entire input data, and Bob had to carefully examine again if any of the transformation results were incorrect. This was tedious given the large volume of heterogeneous data **(verification at string level is challenging)**. After rounds of repairing and verifying, Bob was finally sure that FlashFill successfully transformed all existing phone numbers in the data set, and he thought the transformation inferred by FlashFill was impeccable. Yet, when he used it to transform another data set, a phone number "+1 724-285-5210" was mistakenly transformed as "(1) 724-285", which suggested that the transformation logic may fail anytime **(unexplainable transformation logic functions unexpectedly)**. Customer phone numbers were critical information for Bob's company and it was important not to damage them during the transformation. With little insight from FlashFill regarding the transformation program generated, Bob was not sure if the transformation was reliable and had to do more testing **(lack of understanding increases verification effort)**.

Bob heard about CLX and decided to give it a try.

Example 2.2. He loaded his data into CLX and it immediately presented a list of distinct string patterns for phone numbers in the input data (Figure 3), which helped Bob quickly tell which part of the data were ill-formatted. After Bob selected the desired pattern, CLX immediately transformed all the data and showed a new list of string patterns as Figure 2. **So far, verifying the transformation result was straightforward.** The inferred program is presented as a set of Replace operations on raw patterns in Figure 3, each with a picture visualizing the transformation effect. Bob was not a regular expressions guru, but

Notation	Description
$\mathcal{S} = \{s_1, s_2, \dots\}$	A set of ad hoc strings $s_1, s_2, \dots$ to be transformed.
$\mathcal{P} = \{p_1, p_2, \dots\}$	A set of string patterns derived from $\mathcal{S}$.
$p_i = \{t_1, t_2, \dots\}$	Pattern made from a sequence of tokens t_i
$\mathcal{T}$	The desired target pattern that all strings in $\mathcal{S}$ needed to be transformed into.
$\mathcal{L} = \{(p_1, f_1), (p_2, f_2), \dots\}$	Program synthesized in CLX transforming data the patterns of $\mathcal{P}$ into $\mathcal{T}$.
$\mathcal{E}$	The expression $\mathcal{E}$ in $\mathcal{L}$, which is a concatenation of Extract and/or ConstStr operations. It is a transformation plan for a source pattern. We also refer to it as an *Atomic Transformation Plan* in the paper.
$Q(\widetilde{t}, p)$	Frequency of token $\widetilde{t}$ in pattern p
$\mathcal{G}$	Potential expressions represented in Directed Acyclic Graph.

Table 1: Frequently used notations

Token Class	Regular Expression	Example	Notation
digit	`[0-9]`	"12"	$\langle D \rangle$
lower	`[a-z]`	"car"	$\langle L \rangle$
upper	`[A-Z]`	"IBM"	$\langle U \rangle$
alpha	`[a-zA-Z]`	"Excel"	$\langle A \rangle$
alpha-numeric	`[a-zA-Z0-9_-]`	"Excel2013"	$\langle AN \rangle$

Table 2: Token classes and their descriptions

these operations seemed simple to understand and verify. Like many users in our User Study (Section 7.3), **Bob had a deeper understanding of the inferred transformation logic with CLX than with** FlashFill, **and hence, he knew well when and how the program may fail, which saved him from the effort of more blind testing.**

3 OVERVIEW

3.1 Patterns and Data Transformation Problem

A data pattern, or string pattern, is a "high-level" description of the attribute value's string. A natural way to describe a pattern could be a regular expression over the characters that constitute the string. In data transformation, we find that groups of contiguous characters are often transformed together as a group. Further, these groups of characters are meaningful in themselves. For example, in a date string "11/02/2017", it is useful to cluster "2017" into a single group, because these four digits are likely to be manipulated together. We call such meaningful groups of characters as *tokens*.

Table 2 presents all *token classes* we currently support in our instantiation of CLX, including their class names, regular expressions, and notation. In addition, we also support tokens of constant values (e.g., ",", ":"). In the rest of the paper, we represent and handle these tokens of constant values differently from the 5 token classes defined in Table 2. For convenience of presentation, we denote such tokens with constant values as ***literal tokens*** and tokens of 5 token classes defined in Table 2 as ***base tokens***.

A pattern is written as a sequence of tokens, each followed by a quantifier indicating the number of occurrences of the preceding token. A quantifier is either a single natural number or "+", indicating that the token appears at least once. In the rest of the paper, to be succinct, a token will be denoted as "$\langle\widetilde{t}\rangle q$" if q is a number (e.g., $\langle D \rangle 3$) or "$\langle\widetilde{t}\rangle +$" otherwise (e.g., $\langle D \rangle +$). If $\widetilde{t}$ is a literal token, it will be surrounded by a single quotation mark, like ':'. When a pattern is shown to the end user, it is presented as a *natural-language-like regular expression* proposed by Wrangler [13] (see regexps in Fig 4).
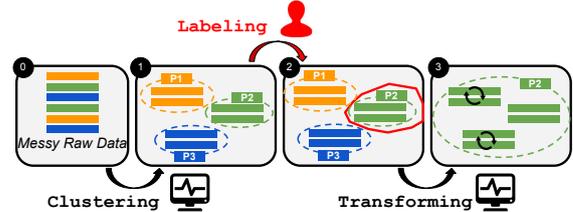

Figure 5: "CLX" Model: Cluster–Label–Transform

With the above definition of data patterns, we hereby formally define the problem we tackle using the CLX framework—data transformation. Data transformation or wrangling is a broad concept. Our focus in this paper is to apply the CLX paradigm to transform a data set of heterogeneous patterns into a desired pattern. A formal definition of the problem is as follows:

DEFINITION 3.1 (DATA (PATTERN) TRANSFORMATION). *Given a set of strings* $\mathcal{S} = \{s_1, \dots, s_n\}$, *generate a program* $\mathcal{L}$ *that transforms each string in* $\mathcal{S}$ *to an equivalent string matching the user-specified desired target pattern* $\mathcal{T}$.

$\mathcal{L} = \{(p_1, f_1), (p_2, f_2), \dots\}$ is the program we synthesize in the transforming phase of CLX. It is represented as a set regexp replace operations, $\text{Replace}(p, f)$[1], that many people are familiar with (e.g., Fig 4).

With above definitions of patterns and data transformations, we present the CLX framework for data transformation.

3.2 CLX Data Transformation Paradigm

We propose a data transformation paradigm called Cluster-Label-Transform (CLX, pronounced "clicks"). Figure 5 visualizes the interaction model in this framework.

Clustering — The clustering component groups the raw input data into clusters based on their data patterns/formats. Compared to raw strings, data patterns is a more abstract representation. The number of patterns is fewer than raw strings, and hence, it can make the user understand the data and verify the transformation more quickly. Patterns discovered during clustering is also useful information for the downstream program synthesis algorithm to determine the number of regexp replace operations, as well as the desirable input patterns and transformation functions.

Labeling — Labeling is to specify the desired data pattern that every data instance is supposed to be transformed into. Presumably, labeling can be achieved by having the user choose among the set of patterns we derive in the clustering process assuming some of the raw data already exist in the desired format. If no input data matches the target pattern, the user could alternatively choose to manually specify the target data form.

Transforming — After the desired data pattern is labeled, the system automatically synthesizes data transformation logic that transforms all undesired data into the desired form and also proactively helps the user understand the transformation logic.

In this paper, we present an instantiation of the CLX paradigm for *data pattern transformation*. Details about the clustering component and the transformation component are discussed in Section 4 and 6. In Section 5, we show the domain-specific-language (DSL) we use to represent the program $\mathcal{L}$ as the outcome of program synthesis, which can be then presented as the regexp replace operations. The paradigm has been designed to allow new

---

[1] p is the regular expression, and f is the replacement string indicating the operation on the string matching the pattern p.

algorithms and DSLs for transformation problems other than data pattern transformation; we will pursue other instantiations in future work.

4 CLUSTERING DATA ON PATTERNS

In CLX, we first cluster data into meaningful groups based on their structure and obtain the pattern information, which helps the user quickly understand the data. To minimize user effort, this clustering process should ideally not require user intervention.

LEARNPADS [4] is an influential project that also targets string pattern discovery. However, LEARNPADS is orthogonal to our effort in that their goal is mainly to find a comprehensive and unified description for the entire data set whereas we seek to partition the data into clusters, each cluster with a single data pattern. Also, the PADS language [3] itself is known to be hard for a non-expert to read [29]. Our interest is to derive simple patterns that are comprehensible. Besides the explainability, efficiency is another important aspect of the clustering algorithm we must consider, because input data can be huge and the real-time clustering must be interactive.

To that end, we propose an automated means to hierarchically cluster data based on data patterns given a set of strings. The data is clustered through a two-phase profiling: (1) tokenization: tokenize the given set of strings of ad hoc data and cluster based on these initial patterns, (2) agglomerative refinement: recursively merge pattern clusters to formulate a **pattern cluster hierarchy** that allows the end user to view/understand the pattern structure information in a simpler and more systematic way, and also helps CLX generate a simple transformation program.

4.1 Initial Clustering Through Tokenization

Tokenization is a common process in string processing when string data needs to be manipulated in chunks larger than single characters. A simple parser can do the job.

Below are the rules we follow in the tokenization phase.

- Non-alphanumeric characters carry important hints about the string structure. Each such character is identified as an individual literal token.
- We always choose the most precise base type to describe a token. For example, a token with string content "cat" can be categorized as "lower", "alphabet" or "alphanumeric" tokens. We choose "lower" as the token type for this token.
- The quantifiers are always natural numbers.

Here is an example of the token description of a string data record discovered in tokenization phase.

Example 4.1. Suppose the string "Bob123@gmail.com" is to be tokenized. The result of tokenization becomes [$\langle U \rangle$, $\langle L \rangle 2$, $\langle D \rangle 3$, '@', $\langle L \rangle 5$, '.', $\langle L \rangle 3$].

After tokenization, each string corresponds to a data pattern composed of tokens. We create the initial set of pattern clusters by clustering the strings sharing the same patterns. Each cluster uses its pattern as a label which will later be used for refinement, transformation, and user understanding.

Find Constant Tokens — Some of the tokens in the discovered patterns have constant values. Discovering such constant values and representing them using the actual values rather than base tokens helps improve the quality of the program synthesized. For example, if most entities in a faculty name list contain "Dr.", it is better to represent a pattern as ['Dr.','\ ', $\langle U \rangle$', '$\langle L \rangle +$'] than ['$\langle U \rangle$', '$\langle L \rangle$', '.', '\ ', '$\langle U \rangle$', '$\langle L \rangle +$']. Similar to [4], we find tokens

---

Algorithm 1: Refine Pattern Representations

Data: Pattern set $\mathcal{P}$, generalization strategy $\tilde{g}$
Result: Set of more generic patterns $\mathcal{P}_{final}$

1   $\mathcal{P}_{final}, \mathcal{P}_{raw} \leftarrow \emptyset$;
2   $C_{raw} \leftarrow \{\}$;
3   **for** $p_i \in \mathcal{P}$ **do**
4     $p_{parent} \leftarrow getParent(p_i, \tilde{g})$;
5   add p_{parent} to $\mathcal{P}_{raw}$;
6     $C_{raw}[p_{parent}] = C_{raw}[p_{parent}] + 1$;
7   **for** $p_{parent} \in \mathcal{P}_{raw}$ *ranked by* C_{raw} *from high to low* **do**
8     $p_{parent}.child \leftarrow \{p_j | \forall p_j \in \mathcal{P}, p_j.isChild(p_{parent})\}$;
9   add p_{parent} to $\mathcal{P}_{final}$;
10   remove $p_{parent}.child$ from $\mathcal{P}$;
11 **Return** $\mathcal{P}_{final}$;

---

with constant values using the statistics over tokenized strings in the data set.

4.2 Agglomerative Pattern Cluster Refinement

In the initial clustering step, we distinguish different patterns by token classes, token positions, and quantifiers, the actual number of pattern clusters discovered in the ad hoc data in tokenization phase could be huge. User comprehension is inversely related to the number of patterns. It is not very helpful to present too many very specific pattern clusters all at once to the user. Plus, it can be unacceptably expensive to develop data pattern transformation programs separately for each pattern.

To mitigate the problem, we build *pattern cluster hierarchy*, i.e., a hierarchical pattern cluster representation with the leaf nodes being the patterns discovered through tokenization, and every internal node being a *parent pattern*. With this hierarchical pattern description, the user can understand the pattern information at a high level without being overwhelmed by many details, and the system can generate simpler programs. Plus, we do not lose any pattern discovered previously.

From bottom-up, we recursively cluster the patterns at each level to obtain *parent patterns*, i.e., more generic patterns, formulating the new layer in the hierarchy. To build a new layer, Algorithm 1 takes in different generalization strategy $\tilde{g}$ and the child pattern set $\mathcal{P}$ from the last layer. Line 3-5 clusters the current set of pattern clusters to get parent pattern clusters using the generalization strategy $\tilde{g}$. The generated set of parent patterns may be identical to others or might have overlapping expressive power. Keeping all these parent patterns in the same layer of the cluster hierarchy is unnecessary and increases the complexity of the hierarchy generated. Therefore, we only keep a small subset of the parent patterns initially discovered and make sure they together can cover any child pattern in $\mathcal{P}$. To do so, we use a counter C_{raw} counting the frequencies of the obtained parent patterns (line 6). Then, we iteratively add the parent pattern that covers the most patterns in $\mathcal{P}$ into the set of more generic patterns to be returned (line 7-10). The returned set covers all patterns in $\mathcal{P}$ (line 11). Overall, the complexity is $O(n \log n)$, where n is the number of patterns in $\mathcal{P}$, and hence, the algorithm itself can quickly converge.

In this paper, we perform three rounds of refinement to construct the new layer in the hierarchy, each with a particular generalization strategy:
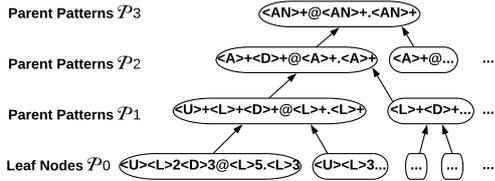
(1) natural number quantifier to '+'

Figure 6: Hierarchical clusters of data patterns

(2) $\langle L \rangle$, $\langle U \rangle$ tokens to $\langle A \rangle$

(3) $\langle A \rangle$, $\langle N \rangle$, '-', '_' tokens to $\langle AN \rangle$

Example 4.2. Given the pattern we obtained in Example 4.1, we successively apply Algorithm 1 with Strategy 1, 2 and 3 to generalize parent patterns $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ and construct the pattern cluster hierarchy as in Figure 6.

4.3 Limitations

The pattern hierarchy constructed can succinctly profile the pattern information for many data. However, the technique itself may be weak in two situations. First, as the scope of this paper is limited to addressing the syntactic transformation problem (Section 5), the pattern discovery process we propose only considers syntactic features, but no semantic features. This may introduce the issue of "misclustering". For example, a date of format "MM/DD/YYYY" and a date of format "DD/MM/YYYY" may be grouped into the same cluster of "$\langle N \rangle 2 / \langle N \rangle 2 / \langle N \rangle 4$", and hence, transforming from the former format into the latter format is impossible in our case. Addressing this problem requires the support for semantic information discovery and transformation, which will be in our future work. Another possible weakness of "fail to cluster" is also mainly affected by the semantics issue: we may fail to cluster semantically-same but very messy data. E.g., we may not cluster the local-part (everything before '@') of a very weird email address "Mike'John.Smith@gmail.com" (token $\langle AN \rangle$ cannot capture ''' or '.'). Yet, this issue can be easily resolved by adding additional regexp-based token classes (e.g., emails). Adding more token classes is beyond the interest of our work.

5 DATA PATTERN TRANSFORMATION PROGRAM

As motivated in Section 1 and Section 3, our proposed data transformation framework is to synthesize a set of regexp replace operations that people are familiar with as the desired transformation logic. However, representing the logic as regexp strings will make the program synthesis difficult. Instead, to simplify the program synthesis, we propose a new language, UNIFI, as a representation of the transformation logic internal to CLX. The grammar of UNIFI is shown in Figure 7. We then discuss how to explain a inferred UNIFI program as regexp replace operations.

The top-level of any UNIFI program is a Switch statement that conditionally maps strings to a transformation. Match checks whether a string s is an *exact match* of a certain pattern p we discover previously. Once a string matches this pattern, it will be processed by an *atomic transformation plan* (expression $\mathcal{E}$ in UNIFI) defined below.

DEFINITION 5.1 (ATOMIC TRANSFORMATION PLAN). *Atomic transformation plan is a sequence of parameterized string operators that converts a given source pattern into the target pattern.*

The available string operators include ConstStr and Extract. ConstStr($\widetilde{s}$) denotes a constant string $\widetilde{s}$. Extract($\widetilde{t_i}, \widetilde{t_j}$) extracts

$$\text{Program } \mathcal{L} := \text{Switch}((b_1, \mathcal{E}_1), \ldots, (b_n, \mathcal{E}_n))$$
$$\text{Predicate } b := \text{Match}(s, p)$$
$$\text{Expression } \mathcal{E} := \text{Concat}(f_1, \ldots, f_n)$$
$$\text{String Expression } f := \text{ConstStr}(\widetilde{s}) \mid \text{Extract}(\widetilde{t_i}, \widetilde{t_j})$$
$$\text{Token Expression } t_i := (\widetilde{t}, r, q, i)$$

Figure 7: UNIFI Language Definition

from the ith token to the jth token in a pattern. In the rest of the paper, we express an Extract operation as Extract(i,j), or Extract(i) if $i = j$. A token t is represented as $(\widetilde{t}, r, q, i)$: $\widetilde{t}$ is the token class in Table 2; r represents the corresponding regular expression of this token; q is the quantifier of the token expression; i denotes the index (one-based) of this token in the source pattern.

As with FLASHFILL [6] and BLINKFILL [24], we only focus on syntactic transformation, where strings are manipulated as a sequence of characters and no external knowledge is accessible, in this instantiation design. *Semantic transformation* (e.g., converting "March" to "03") is a subject for future work. Further–again like BLINKFILL [24]–our proposed data pattern transformation language UNIFI does not support loops. Without the support for loops, UNIFI may not be able to describe transformations on an unknown number of occurrences of a given pattern structure.

We use the following two examples used by FLASHFILL [6] and BLINKFILL [24] to briefly demonstrate the expressive power of UNIFI, and the more detailed expressive power of UNIFI would be examined in the experiments in Section 7.4. For simplicity, Match(s, p) is shortened as Match(p) as the input string s is fixed for a given task.

Example 5.1. This problem is modified from test case "**Example 3**" in BLINKFILL. The goal is to transform all messy values in the medical billing codes into the correct form "[CPT-XXXX]" as in Table 3.

Raw data	Transformed data
CPT-00350	[CPT-00350]
[CPT-00340]	[CPT-00340]
[CPT-11536]	[CPT-11536]
CPT115	[CPT-115]

Table 3: Normalizing messy medical billing codes

The UNIFI program for this standardization task is

```
Switch(((Match("\[<U>+\-<D>+"),
    (Concat(Extract(1,4),ConstStr(']])))),
  (Match("<U>+\-<D>+"),
    (Concat(ConstStr('['),Extract(1,3),
      ConstStr(']')))),
  (Match("<U>+<D>+"),
    (Concat(ConstStr('['),Extract(1),
    ConstStr('-'),Extract(2),ConstStr(']')))))
```

Example 5.2. This problem is borrowed from "**Example 9**" in FLASHFILL. The goal is to transform all names into a unified format as in Table 4.

Raw data	Transformed data
Dr. Eran Yahav	Yahav, E.
Fisher, K.	Fisher, K.
Bill Gates, Sr.	Gates, B.
Oege de Moor	Moor, O.

Table 4: Normalizing messy employee names

A UNIFI program for this task is

```
Switch(((Match("<U><L>+\.\ <U><L>+\ <U><L>+"),
    Concat(Extract(8,9),ConstStr(','),
```

```
          ConstStr(' '),Extract(5))),
 (Match("<U><L>+\ <U><L>+\,\ <U><L>+\."),
  Concat(Extract(4,5),ConstStr(','),
        ConstStr(' '),Extract(1))),
 (Match("<U><L>+\ <U>+\ <U><L>+"),
  Concat(Extract(6,7),ConstStr(','),
        ConstStr(' '),Extract(1))))
```

Program Explanation — Given a UniFi program L, we want to present it as a set of regexp replace operations, Replace, parameterized by *natural-language-like regexps* used by Wrangler [13] (e.g., Figure 4), which are straightforward to even non-expert users. Each component of $(b, \mathcal{E})$, within the Switch statement of L, will be explained as a Replace operation. The replacement string f in the Replace operation is created from p and the transformation plan $\mathcal{E}$ for the condition b. In f, a ConstStr($\tilde{s}$) operation will remain as $\tilde{s}$, whereas a Extract($\tilde{t_i}, \tilde{t_j}$) operation will be interpreted as \$$\tilde{t_i} \ldots \$\tilde{t_j}$. The pattern p in the predicate $b = $ Match(s, p) in UniFi naturally becomes the regular expression p in Replace with each tokens to be extracted surrounded by a pair of parentheses indicating that it can be extracted. Note that if multiple consecutive tokens are extracted in p, we merge them as one component to be extracted in p and change the f accordingly for convenience of presentation. Figure 4 is an example of the transformation logic finally shown to the user.

In fact, these Replace operations can be further explained using visualization techniques. For example, we could add a *Preview Table* (e.g., Figure 8) to visualize the transformation effect in our prototype in a sample of the input data. The user study in Section 7.3 demonstrates that our effort of outputting an explainable transformation program helps the user understand the transformation logic generated by the system.

6 PROGRAM SYNTHESIS

We now discuss how to find the desired transformation logic as a UniFi program using the pattern cluster hierarchy obtained. Algorithm 2 shows our synthesis framework.

Given a pattern hierarchy, we do not need to create an *atomic transformation plan* (Definition 5.1) for every pattern cluster in the hierarchy. We traverse the pattern cluster hierarchy top-down to find valid *candidate source patterns* (line 6, see Section 6.1). Once a source candidate is identified, we discover all *token matches* between this source pattern in Q_{solved} and the target pattern (line 7, see Section 6.2). With the generated token match information, we synthesize the data pattern normalization program including an atomic transformation plan for every source pattern (line 11, see Section 6.3).

6.1 Identify Source Candidates

Before synthesizing a transformation for a source pattern, we want to quickly check whether it can be a *candidate source pattern* (or source candidate), i.e., it is possible to find a transformation from this pattern into the target pattern, through validate. **If we can immediately disqualify some patterns, we do not need to go through more expensive data transformation synthesis process for them.** There are a few reasons why some pattern in the hierarchy may not be qualified as a candidate source pattern:

(1) The input data set may be ad hoc and a pattern in this data set can be a description of noise values. For example, a data set of phone numbers may contain "N/A" as a data record because the customer refused to reveal this information. In this case, it is meaningless to generate transformations.

Algorithm 2: Synthesize UniFi Program

Data: Pattern cluster hierarchy root $\mathcal{P}_R$, target pattern $\mathcal{T}$
Result: Synthesized program $\mathcal{L}$

1 $Q_{unsolved}, Q_{solved} \leftarrow [\,]$;
2 $\mathcal{L} \leftarrow \emptyset$;
3 push $\mathcal{P}_R$ to $Q_{unsolved}$;
4 **while** $Q_{unsolved} \neq \emptyset$ **do**
5 $\quad$ $p \leftarrow$ pop $Q_{unsolved}$;
6 $\quad$ **if** validate$(p, \mathcal{T}) = \top$ **then**
7 $\quad\quad$ $\mathcal{G} \leftarrow$ findTokenAlignment$(p, \mathcal{T})$;
8 $\quad\quad$ push $\{p, \mathcal{G}\}$ to Q_{solved};
9 $\quad$ **else**
10 $\quad\quad$ push p.children to $Q_{unsolved}$;
11 $\mathcal{L} \leftarrow$ createProgs(Q_{solved});
12 **Return** $\mathcal{L}$

---

(2) We may be fundamentally not able to support some transformations (e.g., semantic transformations are not supported as in our case). Hence, we should filter out certain patterns which we think semantic transformation is unavoidable, because it is impossible to transform them into the desired pattern without the help from the user.

(3) Some patterns are too general; it can be hard to determine how to transform these patterns into the target pattern. We can ignore them and create transformation plans for their children. For instance, if a pattern is "$\langle AN \rangle +, \langle AN \rangle +$", it is hard to tell if or how it could be transformed into the desired pattern of "$\langle U \rangle \langle L \rangle + : \langle D \rangle +$". By comparison, its child pattern "$\langle U \rangle \langle L \rangle +, \langle D \rangle +$" seems to be a better fit as the candidate source.

Any input data matching no candidate source pattern is left unchanged and flagged for additional review, which could involve replacing values with NULL or default values or manually overriding values.

Since the goal here is simply to quickly prune those patterns that are not good source patterns, the checking process should be able to find unqualified source patterns with *high precision* but not necessarily *high recall*. Here, we use a simple heuristic of *frequency count* that can effectively reject unqualified source patterns with high confidence: examining if there are sufficient base tokens of each class in the source pattern matching the base tokens in the target tokens. The intuition is that any source pattern with fewer base tokens than the target is unlikely to be transformable into the target pattern without external knowledge; base tokens usually carry semantic meanings and hence are likely to be hard to invent *de novo*.

To apply frequency count on the source pattern p_1 and the target pattern p_2, validate (denoted as $\mathcal{V}$) compares the *token frequency* for every class of base tokens in p_1 and p_2. The token frequency Q of a token class $\langle \tilde{t} \rangle$ in p is defined as

$$Q(\langle \tilde{t} \rangle, p) = \sum_{i=1}^{n} \{t_i.q | t.name = \langle \tilde{t} \rangle\}, p = \{t_1, \ldots, t_n\} \quad (1)$$

If a quantifier is not a natural number but "+", we treat it as 1 in computing Q.

Suppose $\mathfrak{T}$ is the set of all token classes (in our case, $\mathfrak{T} = [\langle D \rangle, \langle L \rangle, \langle U \rangle, \langle A \rangle, \langle AN \rangle]$), $\mathcal{V}$ is then defined as

$$\mathcal{V}(p_1, p_2) = \begin{cases} \text{true} & \text{if } Q(\langle \tilde{t} \rangle, p_1) \geq Q(\langle \tilde{t} \rangle, p_2), \forall \langle \tilde{t} \rangle \in \mathfrak{T} \\ \text{false} & \text{otherwise} \end{cases} \quad (2)$$
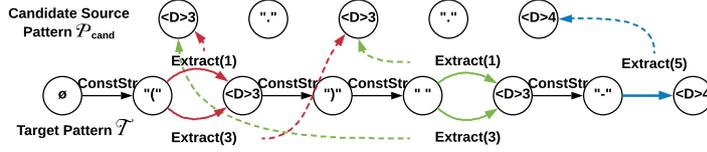
Figure 8: Preview Tab

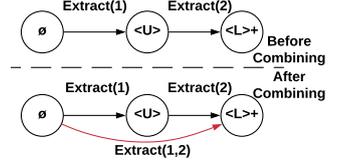Figure 9: Token alignment for the target pattern $\mathcal{T}$

Figure 10: Combine Extracts

Algorithm 3: Token Alignment Algorithm

Data: Target pattern $\mathcal{T} = \{t_1, \ldots, t_m\}$, candidate pattern $\mathcal{P}_{cand} = \{t'_1, \ldots, t'_n\}$, where t_i and t'_i denote base tokens

Result: Directed acyclic graph $\mathcal{G}$

1 $\widetilde{\eta} \leftarrow \{0, \ldots, n\}; \eta^s \leftarrow 0; \eta^t \leftarrow n; \xi \leftarrow \{\}$;
2 **for** $t_i \in \mathcal{T}$ **do**
3   **for** $t'_j \in \mathcal{P}_{cand}$ **do**
4   **if** SyntacticallySimilar$(t_i, t'_j) = \top$ **then**
5           $e \leftarrow$ Extract(t'_j);
6   add e to $\xi_{(i-1,i)}$;
7   **if** $t_i.type = $ 'literal' **then**
8        $e \leftarrow$ ConstStr$(t_i.name)$;
9   add e to $\xi_{(i-1,i)}$;
10 **for** $i \in \{1, \ldots, n-1\}$ **do**
11    $\xi_{in} \leftarrow \{\forall e_p \in \xi_{(i-1,i)}, e_p \text{ is an Extract operation}\}$;
12    $\xi_{out} \leftarrow \{\forall e_q \in \xi_{(i,i+1)}, e_q \text{ is an Extract operation}\}$;
13   **for** $e_p \in \xi_{in}$ **do**
14   **for** $e_q \in \xi_{out}$ **do**
15   **if** $e_p.srcIdx + 1 = e_q.srcIdx$ **then**
16             $e \leftarrow$ Extract$(e_p.t_i, e_q.t_j)$;
17   add e to $\xi_{(i-1,i+1)}$;
18 $\mathcal{G} \leftarrow Dag(\widetilde{\eta}, \eta^s, \eta^t, \xi)$;
19 **Return** $\mathcal{G}$

Example 6.1. Suppose the target pattern $\mathcal{T}$ in Example 5.1 is ['[', $\langle U \rangle$+, '-', $\langle D \rangle$+, ']'], we know

$$Q(\langle D \rangle, \mathcal{T}) = Q(\langle U \rangle, \mathcal{T}) = 1$$

A pattern ['[', $\langle U \rangle 3$, '-', $\langle D \rangle 5$] derived from data record "[CPT-00350" will be identified as a source candidate by validate, because

$$Q(\langle D \rangle, p) = 5 > Q(\langle D \rangle, \mathcal{T}) \wedge$$
$$Q(\langle U \rangle, p) = 3 > Q(\langle U \rangle, \mathcal{T})$$

Another pattern ['[', $\langle U \rangle 3$, '-'] derived from data record "[CPT-" will be rejected because

$$Q(\langle D \rangle, p) = 0 < Q(\langle D \rangle, \mathcal{T})$$

6.2 Token Alignment

Once a source pattern is identified as a source candidate in Section 6.1, we need to synthesize an atomic transformation plan between this source pattern and the target pattern, which explains how to obtain the target pattern using the source pattern. To do this, we need to find the token matches for each token in the target pattern: discover all possible operations that yield a token. This process is called *token alignment*.

For each token in the target pattern, there might be multiple different token matches. Inspired by [6], we store the results of the token alignment in Directed Acyclic Graph (DAG) represented as

a $DAG(\widetilde{\eta}, \eta^s, \eta^t, \xi)$. $\widetilde{\eta}$ denotes all the nodes in DAG with η^s as the source node and η^t as the target node. Each node corresponds to a position in the pattern. ξ are the edges between the nodes in $\widetilde{\eta}$ storing the source information, which yield the token(s) between the starting node and the ending node of the edge. Our proposed solution to token alignment in a DAG is presented in Algorithm 3.

Align Individual Tokens to Sources — To discover sources, given the target pattern $\mathcal{T}$ and the candidate source pattern $\mathcal{P}_{cand}$, we iterate through each token t_i in $\mathcal{T}$ and compare t_i with all the tokens in $\mathcal{P}_{cand}$.

For any source token t'_j in $\mathcal{P}_{cand}$ that is *syntactically similar* (defined in Definition 6.1) to the target token t_i in $\mathcal{T}$, we create a token match between t'_j and t_i with an Extract operation on an edge from t_{i-1} to t_i (line 2-9).

DEFINITION 6.1 (SYNTACTICALLY SIMILAR). *Two tokens t_i and t_j are syntactically similar if: 1) they have the same class, 2) their quantifiers are identical natural numbers or one of them is '+' and the other is a natural number.*

When t_i is a literal token, it is either a symbolic character or a constant value. To build such a token, we can simply use a ConstStr operation (line 7-9), instead of extracting it from the source pattern. This does not violate our previous assumption of not introducing any external knowledge during the transformation.

Example 6.2. Let the candidate source pattern be [$\langle D \rangle 3$, '.', $\langle D \rangle 3$, '.', $\langle D \rangle 4$] and the target pattern be ['(', $\langle D \rangle 3$, ')', ' ', $\langle D \rangle 3$, '-', $\langle D \rangle 4$]. Token alignment result for the source pattern $\mathcal{P}_{cand}$ and the target pattern $\mathcal{T}$, generated by Algorithm 3 is shown in Figure 9. In Figure 9, a dashed line is a token match, indicating the token(s) in the source pattern that can formulate a token in the target pattern. A solid line embeds the actual operation in UniFi rendering this token match.

Combine Sequential Extracts — The Extract operator in our proposed language UniFi is designed to extract one or more tokens sequentially from the source pattern. Line 4-9 only discovers sources composed of an Extract operation generating an individual token. *Sequential extracts* (Extract operations extracting multiple consecutive tokens from the source) are not discovered, and this token alignment solution is not complete. We need to find the *sequential extracts*.

Fortunately, discovering sequential extracts is not independent of the previous token alignment process; sequential extracts are combinations of individual extracts. With the alignment results ξ generated previously, we iterate each state and combine every pair of Extracts on an incoming edge and an outgoing edge that extract two consecutive tokens in the source pattern (line 10-17). The Extracts are then added back to ξ. Figure 10 visualizes combining two sequential Extracts. The first half of the figure (titled "Before Combining") shows a transformation plan that generates a target pattern pattern $\langle U \rangle \langle D \rangle$+ with two operations—Extract(1) and Extract(2). The second half of the figure (titled "After Combining") showcases merging the incoming edge and the outgoing edge (representing the previous two operations)

and formulate a new operation (red arrow), Extract(1,2), as a combined operation of the two.

A benefit of discovering sequential extracts is it helps yield a "simple" program, as described in Section 6.3.

Correctness — Algorithm 3 is *sound* and *complete*, which is proved in Appendix A in the technical report [12].

6.3 Program Synthesis using Token Alignment Result

As we represent all token matches for a source pattern as a DAG (Algorithm 3), finding a transformation plan is to find a path from the initial state 0 to the final state l, where l is the length of the target pattern $\mathcal{T}$.

The Breadth First Traversal algorithm can find all possible atomic transformation plans for this DAG. However, not all of these plans are equally likely to be correct and desired by the end user. The hope is to prioritize the correct plan. The Occam's razor principle suggests that the simplest explanation is usually correct. Here, we apply **Minimum Description Length** (MDL) [23], a formalization of Occam's razor principle, to gauge the *simplicity* of each possible program.

Suppose $\mathcal{M}$ is the set of models. In this case, it is the set of atomic transformation plans found given the source pattern $\mathcal{P}_{cand}$ and the target pattern $\mathcal{T}$. $\mathcal{E} = f_1 f_2 \ldots f_n \in \mathcal{M}$ is an atomic transformation plan, where f is a string expression. Inspired by [21], we define *Description length* (DL) as follows:

$$L(\mathcal{E}, \mathcal{T}) = L(\mathcal{E}) + L(\mathcal{T}|\mathcal{E}) \tag{3}$$

$L(\mathcal{E})$ is the *model description length*, which is the length required to encode the model, and in this case, $\mathcal{E}$. Hence,

$$L(\mathcal{E}) = |\mathcal{E}| \log m \tag{4}$$

where m is the number of distinct types of operations.

$L(\mathcal{T}|\mathcal{E})$ is the *data description length*, which is the sum of the length required to encode $\mathcal{T}$ using the atomic transformation plan $\mathcal{E}$. Thus,

$$L(\mathcal{T}|\mathcal{E}) = \sum_{f_i \in \mathcal{E}} \log L(f_i) \tag{5}$$

where $L(f_i)$ the length to encode the parameters for a single expression. For a Extract(i) or Extract(i,j) operation, $L(f) = \log |\mathcal{P}_{cand}|^2$ (recall Extract(i) is short for Extract(i,i)). For a ConstStr($\bar{s}$), $L(f) = \log c^{|\bar{s}|}$, where c is the size of printable character set ($c = 95$).

With the concept of description length described, we define the minimum description length as

$$L_{min}(\mathcal{T}, \mathcal{M}) = \min_{\mathcal{E} \in \mathcal{M}} \left[L(\mathcal{E}) + L(\mathcal{T}|\mathcal{E}) \right] \tag{6}$$

In the end, we present the atomic transformation plan $\mathcal{E}$ with the minimum description length as the default transformation plan for the source pattern. Also, we list the other k transformation plans with lowest description lengths.

Example 6.3. Suppose the source pattern is "$\langle D \rangle 2/\langle D \rangle 2/\langle D \rangle 4$", the target pattern $\mathcal{T}$ is "$\langle D \rangle 2/\langle D \rangle 2$". The description length of a transformation plan $\mathcal{E}_1 = \text{Concat(Extract(1,3))}$ is $L(\mathcal{E}_1, \mathcal{T}) = 1 \log 1 + 2 \log 3$. In comparison, the description length of another transformation plan $\mathcal{E}_2 = \text{Concat(Extract(1), ConstStr('/'), Extract(3))}$ is $L(\mathcal{E}_2, \mathcal{T}) = 3 \log 2 + \log 3^2 + \log 95 + \log 3^2 > L(\mathcal{E}_1, \mathcal{T})$. Hence, we prefer $\mathcal{E}_1$, a clearly simpler and better plan than $\mathcal{E}_2$.

6.4 Limitations and Program Repair

The target pattern $\mathcal{T}$ as the sole user input so far is more ambiguous compared to input-output example pairs used in most other PBE systems. Also, we currently do not support "semantic transformation". We may face the issue of "semantic ambiguity"—mismatching syntactically similar tokens with different semantic meanings. For example, if the goals is to transform a date of pattern "DD/MM/YYYY" into the pattern "MM-DD-YYYY" (our clustering algorithm works in this case). Our token alignment algorithm may create a match from "DD" in the first pattern to "MM" in the second pattern because they have the same pattern of $\langle D \rangle 2$. The atomic transformation plan we *initially* select for each source pattern can be a transformation that mistakenly converts "DD/MM/YYYY" into "DD-MM-YYYY". Although our algorithm described in Section 6.3 often makes good guesses about the right matches, the system still infers an imperfect transformation about 50% of the time (Appendix E in the technical report [12]).

Fortunately, as our token alignment algorithm is complete and the program synthesis algorithm can discover all possible transformations and rank them in a smart way, the user can quickly find the correct transformation through *program repair*: replace the initial atomic transformation plan with another atomic transformation plans among the ones Section 6.3 suggests for a given source pattern.

To make the repair even simpler for the user, we deduplicate equivalent atomic transformation plans defined below before the repair phase.

DEFINITION 6.2 (EQUIVALENT PLANS). *Two Transformation Plans are* equivalent *if, given the same source pattern, they always yield the same transformation result for any matching string.*

For instance, suppose the source pattern is [$\langle D \rangle 2$, '/', $\langle D \rangle 2$]. Two transformation plans $\mathcal{E}_1 = [\text{Extract(3), Const('/'), Extract(1)}]$ and $\mathcal{E}_2 = [\text{Extract(3), Extract(2), Extract(1)}]$ will yield exactly the same output because the first and third operations are identical and the second operation will always generate a '/' in both plans. If two plans are *equivalent*, presenting both rather than one of them will only increase the user effort. Hence, we only pick the simplest plan in the same equivalence class and prune the rest. The methodology detecting the equivalent plans is elaborated in Appendix B in the technical report [12].

Overall, the repair process does not significantly increase the user effort. In those cases where the initial program is imperfect, 75% of the time the user made just a single repair (Appendix E in the technical report [12]).

7 EXPERIMENTS

We make three broad sets of experimental claims. First, we show that as the input data becomes larger and messier, CLX tends to be less work to use than FLASHFILL because verification is less challenging (Section 7.2). Second, we show that CLX programs are easier for users to understand than FLASHFILL programs (Section 7.3). Third, we show that CLX's expressive power is similar to that of baseline systems, as is the required effort for non-verification portions of the PBE process (Section 7.4).

7.1 Experimental Setup

We implemented a prototype of CLX and compared it against the state-of-the-art PBE system FLASHFILL. For ease of explanation, in this section, we refer this prototype as "CLX". Additionally, to make the experimental study more complete, we
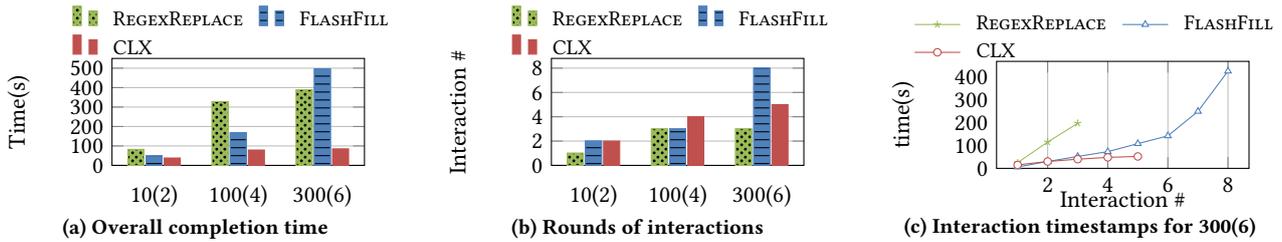
(a) Overall completion time   (b) Rounds of interactions   (c) Interaction timestamps for 300(6)

Figure 11: Scalability of the system usability as data volume and heterogeneity increases (shorter bars are better)
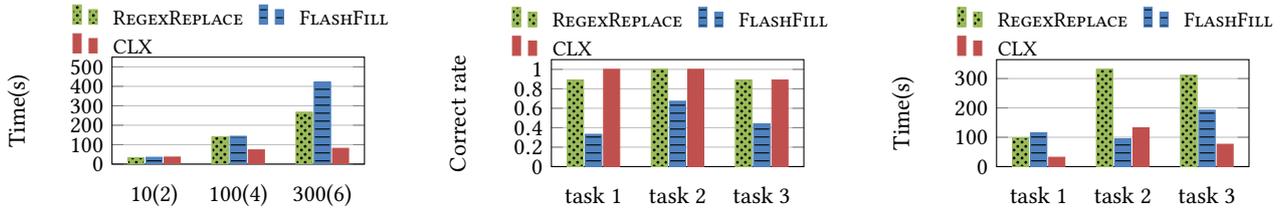

Figure 12: Verification time (shorter bars are better)

Figure 13: User comprehension test (taller bars are better)

Figure 14: Completion time (shorter bars are better)

had a third baseline approach, a non-PBE feature offered by TrifactaWrangler[2] allowing the user to perform string transformation through manually creating Replace operations with simple natural-language-like regexps (referred as RegexReplace). All experiments were performed on a 4-core Intel Core i7 2.8G CPU with 16GB RAM. Other related PBE systems, Foofah [11] and TDE [9], target different workloads and also share the same verification problem we claim for PBE systems, and hence, are not considered as baselines.

7.2 User Study on Verification Effort

In this section, we conduct a user study on a real-world data set to show that (1) verification is a laborious and time-consuming step for users when using the classic PBE data transformation tool (e.g., FlashFill) particularly on a large messy data set, (2) asking end users to hand-write regexp-based data transformation programs is challenging and inefficient, and (3) the CLX model we propose effectively saves the user effort in verification during data transformation and hence its interaction time does not grow fast as the size and the heterogeneity of the data increase.

Test Data Set — Finding public data sets with messy formats suitable for our experiments is very challenging. The first experiment uses a column of 331 messy phone numbers from the "Times Square Food & Beverage Locations" data set [19].

Overview — The task was to transform all phone numbers into the form "⟨D⟩3-⟨D⟩3-⟨D⟩4". We created three test cases by randomly sampling the data set with the following data sizes and heterogeneity: "10(2)" has 10 data records and 2 patterns; "100(4)" has 100 data records and 4 patterns; "300(6)" has 300 data records and 6 patterns.

We invited 9 students in Computer Science with a basic understanding of regular expressions and not involved in our project. Before the study, we educated all participants on how to use the system. Then, each participant was asked to work on one test case on a system and we recorded their performance.

We looked into the user performances on three systems from various perspectives: *overall completion time*, *number of interactions*, and *verification time*. The *overall completion time* gave us a

quick idea of how much the cost of user effort was affected when the input data was increasingly large and heterogeneous in this data transformation task. The other two metrics allowed us to check the user effort in verification. While measuring completion time is straightforward, the other two metrics need to be clarified.

Number of interactions. For FlashFill, the number of interactions is essentially the number of examples the user provides. For CLX we define the number of interactions as the number of times the user verifies (and repairs, if necessary) the inferred atomic transformation plans. We also add one for the initial labeling interaction. For RegexReplace, the number of interactions is the number of Replace operations the user creates.

Verification Time. All three systems follow different interaction paradigms. However, we can divide the interaction process of into two parts, *verification* and *specification*: the user is either busy inputting (typing keyboards, selecting, etc.) or paused to verify the correctness of the transformed data or synthesized/hand-written regular expressions.

Measuring verification time is meaningful because we hypothesize that PBE data transformation systems become harder to use when data is large and messy not because the user has to provide a lot more input, but it becomes harder to verify the transformed data at the instance level.

Results — As shown in Figure 11a, "100(4)" cost 1.1× more time than "10(2)" on CLX, and "300(6)" cost 1.2× more time than "10(2)" on CLX. As for FlashFill, "100(4)" cost 2.4× more time than "10(2)", and "300(6)" cost 9.1× more time than "10(2)". Thus, in this user study, the user effort required by CLX grew slower than that of FlashFill. Also, RegexReplace cost significantly more user effort than CLX but its cost grew not as quickly as FlashFill. This shows good evidence that (1) manually writing data transformation script is cumbersome, (2) the user interaction time grows very fast in FlashFill when data size and heterogeneity increase, and (3) the user interaction time in CLX also grows, but not as fast.

Now, we dive deeper into understanding the causes for observation (2) and (3). Figure 11b shows the number of interactions in all test cases on all systems. We see that all three systems required a similar number of interactions in the first two test cases. Although FlashFill required 3 more interactions than

Task ID	Size	AvgLen	MaxLen	DataType
Task1	10	11.8	14	Human name
Task2	10	20.3	38	Address
Task3	100	16.6	18	Phone number

Table 5: Explainability test cases details

CLX in case "300(6)", this could hardly be the main reason why FLASHFILL cost almost 5x more time than CLX.

We take a close look at the three systems' interactions in the case of "300(6)" and plot the timestamps of each interaction in Figure 11c. The result shows that, in FLASHFILL, as the user was getting close to achieving a perfect transformation, it took the user an increasingly longer amount of time to make an interaction with the system, whereas the interaction time intervals were relatively stable in CLX and REGEXREPLACE. Obviously, the user spent a longer time in each interaction NOT because an example became harder to type in (phone numbers have relatively similar lengths). We observed that, without any help from FLASHFILL, the user had to eyeball the entire data set to identify the data records that were still not correctly transformed, and it became harder and harder to do so simply because there were fewer of them. Figure 12 presents the average verification time on all systems in each test case. "100(4)" cost 1.0× more verification time than "10(2)" on CLX, and "300(6)" cost 1.3× more verification time than "10(2)" on CLX. As for FLASHFILL, "100(4)" cost 3.4× more verification time than "10(2)", and "300(6)" cost 11.4× more verification time than "10(2)". The fact that the verification time on FLASHFILL also grew significantly as the data became larger and messier supports our analysis and claim.

To summarize, this user study presents evidence that FLASH-FILL becomes much harder to use as the data becomes larger and messier mainly because verification is more challenging. In contrast, CLX users generally are not affected by this issue.

7.3 User Study on Explainability

Through a new user study with the same 9 participants on three tasks, we demonstrate that (1) FLASHFILL users lack understanding about the inferred transformation logic, and hence, have inadequate insights on how the logic will work, and show that (2) the simple program generated by CLX improves the user's understanding of the inferred transformation logic.

Additionally, we also compared the overall completion time of three systems.

Test Set — Since it was impractical to give a user too many data pattern transformation tasks to solve, we had to limit this user study to just a few tasks. To make a fair user study, we chose tasks with various data types that cost relatively same user effort on all three systems. From the benchmark test set we will introduce in Section 7.4, we randomly chose 3 test cases that each is supposed to require same user effort on both CLX and FLASHFILL: Example 11 from FlashFill (task 1), Example 3 from PredProg (task 2) and "phone-10-long" from SyGus (task 3). Statistics (number of rows, average/max/min string length of the raw data) about the three data sets are shown in Table 5.

Overview — We designed 3 multiple choice questions for every task examining how well the user understood the transformation regardless of the system he/she interacted with. All the questions were formulated as "Given the input string as x, what is the expected output". All questions are shown in Appendix C in the technical report [12].

Sources	# tests	AvgSize	AvgLen	MaxLen	DataType
SyGus [26]	27	63.3	11.8	63	car model ids, human name, phone number, university name and address
FlashFill [6]	10	10.3	15.8	57	log entry, phone number, human name, date, name and position, file directory, url, product name
BlinkFill [24]	4	10.8	14.9	37	city name and country, human name, product id, address
PredProg [25]	3	10.0	12.7	38	human name, address
Prose [22]	3	39.3	10.2	44	country and number, email, human name and affiliation
Overall	47	43.6	13.0	63	

Table 6: Benchmark test cases details

During the user study, we asked every participant to participate all three tasks, each on a different system (completion time was measured). Upon completion, each participant was asked to answer all questions based on the transformation results or the synthetic programs generated by the system.

Explainability Results — The correct rates for all 3 tasks using all systems are presented in Figure 13. The result shows that the participants were able to answer these questions almost perfectly using CLX, but struggled to get even half correct using FLASHFILL. REGEXREPLACE also achieved a success rate similar to CLX, but required higher user effort and expertise.

The result suggests that FLASHFILL users have insufficient understanding about the inferred transformation logic and CLX improves the users' understanding in all tasks, which provides evidence that verification in CLX can be easier.

Overall Completion Time — The average completion time for each task using all three systems is presented in Figure 14. Compared to FLASHFILL, the participants using CLX spent 30% less time on average: ∼ 70% less time on task 1 and ∼ 60% less time on task 3, but ∼ 40% more time on task 2. Task 1 and task 3 have similar heterogeneity but task 3 (100 records) is bigger than task 1 (10 records). The participants using FLASHFILL typically spent much more time on understanding the data formats at the beginning and verifying the transformation result in solving task 3. This provides more evidence that CLX saves the verification effort. Task 2 is small (10 data records) but heterogeneous. Both FLASHFILL and CLX made imperfect transformation logic synthesis, and the participants had to make several corrections or repairs. We believe CLX lost in this case simply because the data set is too small, and as a result, CLX was not able to exploit its advantage in saving user effort on large-scale data set. The study also gives evidence that CLX is sometimes effective in saving user verification effort in small-scale data transformation tasks.

7.4 Expressivity and Efficiency Tests

In a simulation test using a large benchmark test set, we demonstrate that (1) the expressive power of CLX is comparable to the other two baseline systems FLASHFILL and REGEXREPLACE, and (2) CLX is also pretty efficient in costing user interaction effort.

Test Set — We created a benchmark of 47 data pattern transformation test cases using a mixture of public string transformation test sets and example tasks from related research publications (will be released upon the acceptance of the paper). The information about the number of test cases from each source, average raw input data size (number of rows), average/max data instance length, and data types of these test cases are shown in Table 6.

Baselines	CLX Wins	Tie	CLX Loses
vs. FLASHFILL	17 (36%)	17 (36%)	13 (28%)
vs. REGEXREPLACE	33 (70%)	12 (26%)	2 (4%)

Table 7: User effort simulation comparison.

A detailed description of the benchmark test set is shown in Appendix D in the technical report [12].

Overview — We evaluated CLX against 47 benchmark tests. As conducting an actual user study on all 47 benchmarks is not feasible, we simulated a user following the "lazy approach" used by Gulwani et al. [8]: a simulated user selected a target pattern or multiple target patterns and then repaired the atomic transformation plan for each source pattern if the system proposed answer was imperfect.

Also, we tested the other two systems against the same benchmark test suite. As with CLX, we simulated a user on FLASHFILL; this user provided the first positive example on the first data record in a non-standard pattern, and then iteratively provided positive examples for the data record on which the synthetic string transformation program failed. On REGEXREPLACE, the simulated user specified a Replace operation with two regular expressions indicating the matching string pattern and the transformed pattern, and iteratively specified new parameterized Replace operations for the next ill-formatted data record until all data were in the correct format.

Evaluation Metrics — In experiments, we measured how much user effort all three systems required. Because systems follow different interaction models, a direct comparison of the user effort is impossible. We quantify the user effort by *Step*, which is defined differently as follows

- For CLX, the total Steps is the sum of the number of correct patterns the user chooses (Selection) and the number of repairs for the source patterns whose default atomic transformation plans are incorrect (Repair). In the end, we also check if the system has synthesized a "perfect" program: a program that successfully transforms all data.
- For FLASHFILL, the total Steps is the sum of the number of input examples to provide and the number of data records that the system fails to transform.
- For REGEXREPLACE, each specified Replace operation is counted as 2 Steps as the user needs to type two regular expressions for each Replace, which is about twice the effort of giving an example in FLASHFILL.

In each test, for any system, if not all data records were correctly transformed, we added the number of data records that the system fails to transform correctly to its total *Step* value as a punishment. In this way, we had a coarse estimation of the user effort in all three systems on the 47 benchmarks.

Expressivity Results — CLX could synthesize right transformations for 42/47 ($\sim$ 90%) test cases, whereas FLASHFILL reached 45/47 ($\sim$ 96%). This suggests that the expressive power of CLX is comparable to that of FLASHFILL.

There were five test cases where CLX failed to yield a perfect transformation. Only one of the failures was due to the expressiveness of the language itself, the others could be fixed if there were more representative examples in the raw data. "Example 13" in FlashFill requires the inference of advanced conditionals (Contains keyword "picture") that UNIFI cannot currently express, but adding support for these conditionals in UNIFI is straightforward. The failures in the remaining four test cases were mainly caused by the lack of the target pattern examples

in the data set. For example, one of the test cases we failed is a name transformation task, where there is a last name "McMillan" to extract. However, all data in the target pattern contained last names comprising one uppercase letter followed by multiple lowercase letters and hence our system did not realize "McMillan" needed to be extracted. We think if the input data is large and representative enough, we should be able to successfully capture all desired data patterns.

REGEXREPLACE allows the user to specify any regular expression replace operations, hence it was able to correctly transform all the input data existed in the test set, because the user could directly write operations replacing the exact string of an individual data record into its desired form. However, similar to UNIFI, REGEXREPLACE is also limited by the expressive power of regular expressions and cannot support advanced conditionals. As such, it covered 46/47 ($\sim$ 98%) test cases.

User Effort Results — As the *Step* metric is a potentially noisy measure of user effort, it is more reasonable to check whether CLX costs more or less effort than other baselines, rather than to compare absolute *Step* numbers. The aggregated result is shown in Table 7. It suggests CLX often requires less or at least equal user effort than both PBE systems. Compared to REGEXREPLACE, CLX almost always costs less or equal user effort. A detailed discussion about the user effort on CLX and comparison with other systems is in Appendix E in the technical report [12].

8 RELATED WORK

Data Transformation — FLASHFILL (now a feature in Excel) is an influential work for syntactic transformation by Gulwani [6]. It designed an expressive string transformation language and proposed the algorithm based on version space algebra to discover a program in the designed language. It was recently integrated to PROSE SDK released by Microsoft. A more recent PBE project, TDE [9], also targets string transformation. Similar to FLASHFILL, TDE requires the user to verify at the instance level and the generated program is unexplainable to the user. Other related PBE data cleaning projects include [11, 24].

Another thread of seminal research including [21], WRANGLER [13] and TRIFACTA created by Hellerstein et al. follow a different interaction paradigm called "predictive interaction". They proposed an inference-enhanced visual platform supporting many different data wrangling and profiling tasks. Based on the user selection of columns, rows or text, the system intelligently suggests possible data transformation operations, such as Split, Fold, or pattern-based extraction operations.

Pattern Profiling — In our project, we focus on clustering ad hoc string data based on structures and derive the structure information. The LEARNPADS [4] project is somewhat related. It presents a learning algorithm using statistics over symbols and tokenized data chunks to discover pattern structure. LEARNPADS assumes that all data entries follow a repeating high-level pattern structure. However, this assumption may not hold for some of the workload elements. In contrast, we create a bottom-up pattern discovery algorithm that does not make this assumption. Plus, the output of LEARNPADS (i.e., PADS program [3]) is hard for a human to read, whereas our pattern cluster hierarchy is simpler to understand. Most recently, DATAMARAN[5] has proposed methodologies for discovering structure information in a data set

whose record boundaries are unknown, but for the same reasons as LearnPADS, Datamaran is not suitable for our problem.

Program Synthesis — Program synthesis has garnered wide interest in domains where the end users might not have good programming skills or programs are hard to maintain or reuse including data science and database systems. Researchers have built various program synthesis applications to generate SQL queries [16, 20, 27], regular expressions [1, 17], data cleaning programs [6, 28], and more.

Researchers have proposed various techniques for program synthesis. [7, 10] proposed a constraint-based program synthesis technique using logic solvers. However, constraint-based techniques are mainly applicable in the context where finding a satisfying solution is challenging, but we prefer a high-quality program rather than a satisfying program. Version space algebra is another important technique that is applied by [6, 14, 15, 18]. [2] recently focuses on using deep learning for program synthesis. Most of these projects rely on user inputs to reduce the search space until a quality program can be discovered; they share the hope that there is one simple solution matching most, if not all, user-provided example pairs. In our case, transformation plans for different heterogeneous patterns can be quite distinct. Thus, applying the version space algebra technique is difficult.

9 CONCLUSION AND FUTURE WORK

Data transformation is a difficult human-intensive task. PBE is a leading approach of using computational inference to reduce human burden in data transformation. However, we observe that standard PBE for data transformation is still difficult to use due to its laborious and unreliable verification process.

We proposed a new data transformation paradigm CLX to alleviate the above issue. In CLX, we build data patterns to help the user quickly identify both well-formatted and ill-formatted data which immediately saves the verification time. CLX also infers regexp replace operations as the desired transformation, which many users are familiar with and boosts their confidence in verification.

We presented an instantiation of CLX with a focus on data pattern transformation including (1) a pattern profiling algorithm that hierarchically clusters both the raw input data and the transformed data based on data patterns, (2) a DSL, UniFi, that can express many data pattern transformation tasks and can be interpreted as a set of simple regular expression replace operations, (3) algorithms inferring a correct UniFi program.

We presented two user studies. In a user study on data sets of various sizes, when the data size grew by a factor of 30, the user verification time required by CLX grew by 1.3× whereas that required by FlashFill grew by 11.4×. The comprehensibility user study shows the CLX users achieved a success rate about twice that of the FlashFill users. The results provide good evidence that CLX greatly alleviates the verification issue.

Although building a highly-expressive data pattern transformation tool is not the central goal of this paper, we are happy to see that the expressive power and user effort efficiency of our initial design of CLX is comparable to those of FlashFill in a simulation study on a large test set in another test.

CLX is a data transformation paradigm that can be used not only for data pattern transformation but other data transformation or transformation tasks too. For example, given a set of heterogeneous spreadsheet tables storing the same information from different organizations, CLX can be used to synthesize programs converting all tables into the same standard format. Building such an instantiation of CLX will be our future work.

10 ACKNOWLEDGMENTS

REFERENCES

[1] A Blackwell. 2001. SWYN: A visual representation for regular expressions. *Your Wish Is My Command: Programming by Example* (2001), 245–270.

[2] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy I/O. *arXiv preprint arXiv:1703.07469* (2017).

[3] Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-specific Language for Processing Ad Hoc Data. In *PLDI*.

[4] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. 2008. From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data. In *POPL*.

[5] Yihan Gao, Silu Huang, and Aditya G. Parameswaran. 2018. Navigating the Data Lake with Datamaran: Automatically Extracting Structure from Log Datasets. In *SIGMOD*.

[6] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *POPL*.

[7] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *PLDI*.

[8] William R Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *PLDI*.

[9] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations. In *PVLDB*.

[10] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *ICSE*.

[11] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *SIGMOD*.

[12] Zhongjun Jin, Michael Cafarella, H. V. Jagadish, Sean Kandel, Michael Minar, and Joseph M. Hellerstein. 2018. CLX: Towards verifiable PBE data transformation. *arXiv preprint arXiv:1803.00701 [cs.DB]* (2018).

[13] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *CHI*.

[14] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1 (2003), 111–156.

[15] Tessa A Lau, Pedro M Domingos, and Daniel S Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration. In *ICML*.

[16] Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. In *PVLDB*.

[17] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and HV Jagadish. 2008. Regular expression learning for information extraction. In *EMNLP*.

[18] Tom M Mitchell. 1982. Generalization as search. *Artificial intelligence* 18, 2 (1982), 203–226.

[19] NYC OpenData. 2017. Times Square Food & Beverage Locations" data set. https://opendata.cityofnewyork.us/. (2017).

[20] Li Qian, Michael J Cafarella, and HV Jagadish. 2012. Sample-driven schema mapping. In *SIGMOD*.

[21] Vijayshankar Raman and Joseph M Hellerstein. 2001. Potter's wheel: An interactive data cleaning system. In *VLDB*, Vol. 1. 381–390.

[22] Microsoft Research. 2017. Microsoft Program Synthesis using Examples SDK. https://microsoft.github.io/prose/. (2017).

[23] Jorma Rissanen. 1978. Modeling by shortest data description. *Automatica* 14, 5 (1978), 465–471.

[24] Rishabh Singh. 2016. BlinkFill: Semi-supervised Programming by Example for Syntactic String Transformations. In *PVLDB*.

[25] Rishabh Singh and Sumit Gulwani. 2015. Predicting a correct program in programming by example. In *CAV*.

[26] Syntax-Guided Synthesis. 2017. SyGuS-COMP 2017: The 4th Syntax Guided Synthesis Competition took place as a satellite event of CAV and SYNT 2017. http://www.sygus.org/SyGuS-COMP2017.html. (2017).

[27] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI*.

[28] Bo Wu and Craig A Knoblock. 2015. An Iterative Approach to Synthesize Data Transformation Programs.. In *IJCAI*.

[29] Kenny Zhu, Kathleen Fisher, and David Walker. 2012. Learnpads++: Incremental inference of ad hoc data formats. *Practical Aspects of Declarative Languages* (2012), 168–182.