UnchartIt: An Interactive Framework for Program Recovery from Charts

Daniel Ramos
INESC-ID / IST, U. Lisboa
Lisboa, Portugal
daniel.r.ramos@tecnico.ulisboa.pt

Jorge Pereira
INESC-ID / IST, U. Lisboa
Lisboa, Portugal
jorge.m.s.pereira@tecnico.ulisboa.pt

Inês Lynce
INESC-ID / IST, U. Lisboa
Lisboa, Portugal
ines.lynce@tecnico.ulisboa.pt

Vasco Manquinho
INESC-ID / IST, U. Lisboa
Lisboa, Portugal
vasco.manquinho@tecnico.ulisboa.pt

Ruben Martins
Carnegie Mellon University
Pittsburgh, USA
rubenm@andrew.cmu.edu

ABSTRACT

Charts are commonly used for data visualization. Generating a chart usually involves performing data transformations, including data pre-processing and aggregation. These tasks can be cumbersome and time-consuming, even for experienced data scientists. Reproducing existing charts can also be a challenging task when information about data transformations is no longer available.

In this paper, we tackle the problem of recovering data transformations from existing charts. Given an input table and a chart, our goal is to automatically recover the data transformation program underlying the chart. We divide our approach into four steps: (1) data extraction, (2) candidate generation, (3) candidate ranking, and (4) candidate disambiguation. We implemented our approach in a tool called UnchartIt and evaluated it on a set of 50 benchmarks from Kaggle. Experimental results show that UnchartIt successfully ranks the correct data transformation among the top-10 programs in 92% of the benchmarks. To disambiguate the top-ranking programs, we use our new interactive procedure, which successfully disambiguates 98% of the ambiguous benchmarks by asking on average fewer than 2 questions to the user.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Recovering Data Transformations from Charts, Program Synthesis, Interactive Disambiguation

ACM Reference Format:
Daniel Ramos, Jorge Pereira, Inês Lynce, Vasco Manquinho, and Ruben Martins. 2020. UnchartIt: An Interactive Framework for Program Recovery from Charts. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416613

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6768-4/20/09.
https://doi.org/10.1145/3324884.3416613

1 INTRODUCTION

In the last decade, data analysis has become one of the most important tools for organizations to drive their decisions. The huge demand for domain experts has led many data analysts with limited programming knowledge to be recruited. Thus, in the last years, several tools [13, 14, 23, 25, 39, 45] have been developed to aid inexperienced analysts in automating some programming tasks. These tools work by example: the user provides a set of input-output examples, and the tool finds a program that maps the inputs into the output. However, the development of tools that work directly with visual elements has remained unexplored. Hence, if a user prefers to express his intent through visual elements (e.g., providing an input table and a bar chart), there is no tool that is able to reverse engineer the necessary data manipulations in order to reproduce it.

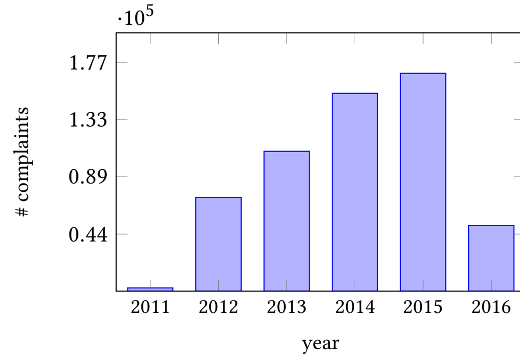
In this paper, we propose UnchartIt, a tool for reverse engineering the necessary table manipulations to generate a given chart. Note that, to the best of our knowledge, this is the first tool for automatic generation of data manipulations that directly uses visual elements. In this work, we consider that the user can provide the image of a bar chart and the raw data from which the chart was generated. Although we only consider bar charts (one of the most common chart types [3]), the proposed ideas can be easily generalized to other types of charts or graphical elements. Moreover, we also address how to automatically extract the necessary information from the chart, how to adapt program synthesis tools to this new challenging problem, as well as how to disambiguate several programs while minimizing user interactions. Furthermore, experimental results on real-world instances from Kaggle show that UnchartIt is able to reverse engineer how to build the chart presented by the user. Specifically, the correct data transformation program is ranked among the top ten programs returned by UnchartIt in 92% of the instances. To select the user's intended program from the top ten ranked programs, UnchartIt interacts with the user by asking either yes or no questions, or multiple-choice questions, and successfully returns the correct program in 98% of the ambiguous instances.

This paper makes the following main contributions:

- The first fully automated tool called UnchartIt that synthesizes table manipulations from bar charts.
- Automated input generation methods to disambiguate a set of programs that minimize the number of user interactions.

date_received	product	...
08/30/2013	Mortgage	...
08/30/2013	Mortgage	...
08/30/2013	Credit reporting	...
08/30/2013	Student loan	...
08/30/2013	Debt collection	...
08/30/2013	Credit card	...
08/30/2013	Credit card	...
08/30/2013	Debt collection	...

(a) Sample of the consumer complaints table (175.39MB).



(b) Bar chart with yearly number of consumer complaints.

Figure 1: Consumer complaints data from 2011 to 2016.

- Experimental results on real-world benchmarks that show the success of the proposed approaches.

The paper is structured as follows: Section 2 defines the research challenge and motivates the problem with a concrete example. Section 3 addresses the problem of data extraction from visual elements and Section 4 explains the necessary changes to program synthesizers to solve this new research problem. Next, Section 5 addresses how to rank the programs generated by the synthesizer. Section 6 proposes new models on how to disambiguate the top-n ranked programs for two different user interaction models. Section 7 presents the experimental results on a set of real data from different domains. Section 8 briefly reviews related work. Finally, the paper concludes in Section 9.

2 MOTIVATION

Consider the sample of the consumer complaints database shown in Figure 1a. The database contains complaints submitted to the Consumer Financial Protection Bureau between 2011 and 2016. Figure 1b shows a bar chart with the number of complaints received in each year.

Suppose that Alice, a data analyst with low programming skills, needs to elaborate a report on an updated version of the consumer complaints database.[1] As a reference, she received an old report written by a former employee. This report contains a variety of charts, including Figure 1b, but not the programs from which the charts originated. Therefore, Alice's task is to recover the programs necessary to reproduce the report's charts. If Alice has the programs to generate the charts, she can update them whenever new data is added to the database.

In this paper, we describe UnchartIt, a new tool that can automatically recover a program from a given chart for people like Alice. To recover a program from a chart, Alice needs to provide the raw data from which the chart originated and an image of the chart. Figure 2 illustrates the UnchartIt architecture. Given a pair (data, chart), UnchartIt starts by extracting data from the chart, thereby creating a tabular representation of the chart. Since this step involves automatically interpreting a chart, the resulting table is prone to contain imprecisions. For instance, from the chart of Figure 1b, UnchartIt generates Table 1. In contrast, Table 2 contains the real table underlying the chart of Figure 1b. Note that the numerical data of the extracted table is imprecise and the bar labels are missing.

Table 1: Table obtained from the bar chart of Figure 1b.

col_0	col_1
bar_0	2345.18
bar_1	72255.90
bar_2	108303.62
bar_3	153090.18
bar_4	168929.33
bar_5	50954.98

Table 2: Real table inherent to the bar chart of Figure 1b.

year	# complaints
2011	2549
2012	72523
2013	108273
2014	153138
2015	168621
2016	50853

After obtaining a tabular representation of the chart, UnchartIt starts the candidate program generation step. During this stage, UnchartIt uses two major components: (1) the program generator, and (2) the program decider. The program generator enumerates candidate programs and provides them to the program decider. The program decider evaluates the candidate programs, decides if they are good candidates, and provides feedback to the program generator. Note that the program decider does not have access to the real table underlying the chart, but rather to an approximation of the real table extracted from the chart image. Therefore, the program decider cannot simply discard candidates because they do not map the raw input data into the imprecise table it extracted in the previous step. Instead, it decides to keep or discard candidates using a weaker criterion: a candidate program is kept if and only if its output on the input data has the same number of rows and columns as the extracted table. For example, using the consumer complaints data from Figure 1a and the extracted table shown in Table 1, UnchartIt finds 7 different programs whose output on the input data has the same structure as Table 1 (6 rows and 2 columns).

After generating a pool of candidates, UnchartIt assigns each candidate program a score using a cost function and ranks the programs according to their costs. Since it is possible that the best-ranking program does not correspond to the program the user

[1] https://www.consumerfinance.gov/data-research/consumer-complaints/

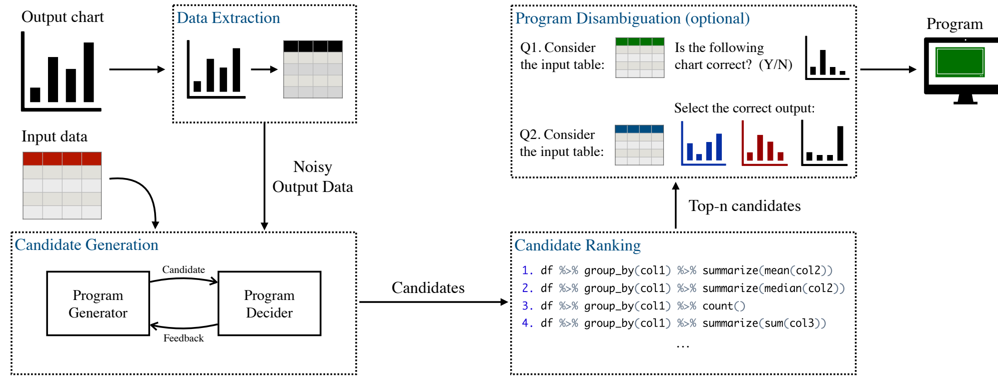
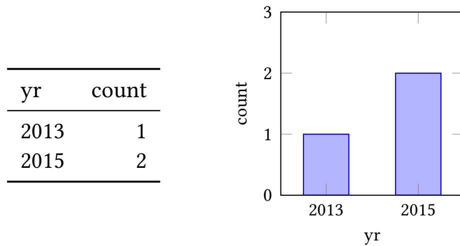


Figure 2: UNCHARTIT architecture.

Consider the following input table:

date_received	date_sent
24/07/2015	03/12/2011
10/01/2013	01/12/2012
23/01/2015	17/07/2014

Q: Is the following output table / chart correct? (Y/N)



yr	count
2013	1
2015	2

Figure 3: User interaction example.

desires, UNCHARTIT gives the user the option to answer a series of questions in order to disambiguate the top-n programs of the rank. In particular, UNCHARTIT has two different user interaction models to disambiguate programs. In the first interaction model, UNCHARTIT asks the user to pick the correct output (from a set of options) for a given input. In the second interaction model, UNCHARTIT asks the user if a given test case is correct or not (yes/no question). In both approaches, UNCHARTIT automatically generates small test cases that minimize the number of user interactions. For example, after enumerating the 7 different candidate programs for the specification given by the pair (Figure 1a, Figure 1b), UNCHARTIT only needs to ask the user 3 questions before returning the correct program using the yes/no interaction model.

Figure 3 illustrates the user interaction in the yes/no interaction model. Given a small intput table (automatically generated by UNCHARTIT), the user just needs to confirm if the given chart corresponds to the correct output. This example corresponds to the last question of the user interaction for the problem in Figure 1, and the returned program is shown in Figure 4.

```
l1 <- df %>% mutate(date = mdy(date_received))
l2 <- l1 %>% mutate(yr = year(date))
l3 <- l2 %>% group_by(yr)
l4 <- l3 %>% summarise(count = n())
l5 <- l4 %>% ggplot(aes(x=yr, y=count)) + geom_col()
```

Figure 4: Program returned by UNCHARTIT for the instance given by Figure 1 after the interaction of Figure 3.

3 DATA EXTRACTION

The first step in our pipeline requires us to transform the chart's image into a standard tabular representation (e.g., transforming the chart of Figure 1b into Table 1). This is a crucial step because there can be many different ways of depicting the same information. By transforming the chart into a standard tabular representation, we can build a simpler validation mechanism to decide the suitability of candidate programs to the given specification. Let T_e denote the table extracted from the chart. Given a program $\mathcal{P}$ that outputs a table T_p when $\mathcal{P}$ is executed on the input data, then we can compare T_e against T_p in order to evaluate the quality of program $\mathcal{P}$. In this section, we focus on extracting data from bar charts. Nevertheless, the techniques discussed here can be adapted to work with other chart types.

3.1 WebPlotDigitizer

WEBPLOTDIGITIZER [34] is one of the most prominent tools for manual and automatic extraction of data from charts. In particular, WEBPLOTDIGITIZER can automatically extract numerical data from simple 2D bar charts (i.e., without stacked and grouped bars). To extract data from a bar chart, WEBPLOTDIGITIZER requires 3 sets of parameters: (1) the chart's image; (2) the pixel location of two different points (P_1, P_2) over the continuous axis along the bars, and their corresponding values on that axis (Figure 5 shows an example); (3) the width in pixels of the bars (Δx), and the height in pixels of the highest bar (Δval).

WEBPLOTDIGITIZER combines these sets of parameters to extract the numerical data from the charts, but it does not extract the labels of the bars.

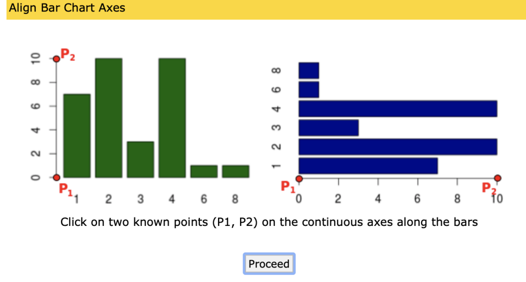


Figure 5: First step of WebPlotDigitizer's axis calibration. WebPlotDigitizer uses P1 and P2 to find the value of any point on the axis by performing a linear interpolation.

3.2 Neural Data Extraction

An alternative approach to traditional chart digitization algorithms, such as those employed by WebPlotDigitizer, is to use machine learning. Tools such as ReVision [36] and ChartSense [24] have previously used machine learning algorithms to discern between chart types (e.g., deciding whether an image contains a bar chart or a scatter plot). However, these tools often rely on traditional algorithms to extract data from the charts. In this section, we propose to leverage state-of-the-art Convolution Neural Networks (CNNs) to retrieve data from bar charts.

CNNs are known for their huge success in modern computer vision systems. A computer vision benchmark of particular relevance on which CNNs shine is the ImageNet challenge [35]. The purpose of the ImageNet challenge is to assess the performance of algorithms on classification and object detection tasks. The EfficientNet-B7 fulfills the current state-of-the-art performance on ImageNet challenge. The EfficientNets [37] are a family of eight CNNs ranging from EfficientNet-B0 to EfficientNet-B7. The "X" in EfficientNet-BX indicates the network's complexity: the higher the "X", the more complex the network is.

Our goal is to leverage the architecture of one of the EfficientNets by replacing its output layer with a custom layer that suits our task. One important piece of information that our adapted EfficientNet should extract is the number of bars of a given chart. To retrieve this information, we add n nodes to the network's output layer, each one representing the probability that the given chart has $i \in \{1, 2, ..., n\}$ bars. To achieve a probability distribution, the n nodes use a softmax activation function:

$$\sigma_i(\mathbf{z}) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}, \quad i = 1, 2, ..., n \qquad \mathbf{z} = [z_1, z_2, ..., z_n] \in \mathbb{R}^n \tag{1}$$

where $\sigma_i(\mathbf{z})$ is the output of the i'th node, indicating the probability that the given chart has i bars. We extract the number of bars of a given chart by taking the argmax of the obtained probability distribution:

$$\hat{y} = \underset{i \in \{1,2,...,n\}}{\operatorname{argmax}} \; \sigma_i(\mathbf{z}) \tag{2}$$

Besides the number of bars, we should also retrieve the bars' heights. We can do this by adding n more nodes to the network's output layer (nodes $n+1$ to $2n$) with the following activation functions:

$$p_i(x) = \max(0, \min(x, 1)), \qquad i = n+1, n+2, ..., 2n \tag{3}$$

where p_i is the output of the i'th node, and it indicates how full is the $i-n$'th bar with respect to the maximum possible height ($p_i = 1$ means that the $i-n$'th bar is full, and $p_i = 0$ means the $i-n$'th bar is empty). Using the height of each bar, we calculate its value by doing a linear interpolation between the axis maximum and lowest values:

$$\hat{y}_i = \begin{cases} L + p_{i+n}(H - L), & \text{if } 1 \le i \le \hat{y} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

where $\hat{y}_i$ is the value of i'th bar, H is the axis maximum value, and L is the axis lowest value. H and L are both user-provided inputs.

Similarly to WebPlotDigitizer, we do not extract the labels of each bar. Finally, to train the network, we minimize the sum of a categorical cross-entropy loss (nodes 1 to n) with a mean squared error (nodes $n+1$ to $2n$).

4 PROGRAM SYNTHESIS

After extracting the table from the chart, UnchartIt starts searching for candidate programs that can potentially transform the input table into the table described by the chart. This problem can be seen as a program synthesis problem where the goal is to find a program that satisfies a given specification. Program synthesis has been successfully used in many applications (e.g., string manipulations [10, 33], list manipulations [2, 15], and table transformations [14, 39]) as well as in commercial applications (e.g., Flash Fill feature in Microsoft Excel [20]). *Programming-by-example* (PBE) synthesis [21] is the most common approach for program synthesis where the synthesizer takes as specification a set of input-output examples and searches for a program that maps each input to the corresponding output. In our case, since the extracted table in the previous step is only an approximation of the real table underlying the chart, we cannot use this criterion to accept or reject programs.

UnchartIt modifies the open-source Trinity synthesis framework [28] to tackle the problem of recovering data transformations from charts. The synthesis process used in UnchartIt is very similar to the one proposed for Morpheus [14]. First, we created a Domain Specific Language (DSL) for the data transformation domain. Note that a DSL is just a useful intermediate representation for the program that abstracts from some syntactic details of the programming language. However, there is a direct correspondence from the DSL symbols to terms in the programming language syntax. The syntax of a DSL is described through a context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$, where V is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols, R is a finite relation from V to $(V \cup \Sigma)^*$ called the production rules, and S is the start symbol. Each terminal symbol $\sigma \in \Sigma$ is either a function, a variable, a constant, or a special character (e.g., parenthesis or comma). Each production rule $\rho \in R$ corresponding to a function is represented in form $\mathcal{A}_0 \rightarrow \beta(\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_n)$, where β is a function, and $\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_n \in V$ are its arguments. A program is a string $P \in (\Sigma \cup V)^*$, such that $S \overset{*}{\Rightarrow} P$.

Example 4.1. The following grammar represents a subset of the DSL used by UNCHARTIT.

$$tab \rightarrow \texttt{summarize}(tab, opt, col) \mid \texttt{group_by}(tab, col) \mid x_0$$
$$tab \rightarrow \texttt{count}(tab) \mid \texttt{top_n}(tab, col) \mid \texttt{bottom_n}(tab, col)$$
$$opt \rightarrow \texttt{mean} \mid \texttt{median} \mid \texttt{sum}$$
$$col \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 10 \mid \ldots$$

where `summarize`, `group_by`, `count`, `top_n`, and `bottom_n` are functions of the dplyr library for R, and x_0 represents the program's input. $\texttt{summarize}(\texttt{group_by}(x_0, 1), \texttt{mean}, 2)$ is an example of a program in this DSL. This program groups all the lines with the same first column representation and for each group it computes the mean of the numerical values in the second column.

Our full DSL has extra primitives to support common table transformations for bar charts. For instance, it allows cleaning data by removing empty cells, supports ordering the values in a column, normalization of values, and extraction of data within a given date. Our DSL is functional, thus our programs do not contain loops.

Second, we wrote logical specifications for each library function in our DSL using as properties the number of rows, columns, and groups. These specifications are a complement to the DSL and describe the relation on the number of columns, rows, and groups between the input and output table after using a library function. UNCHARTIT can then take advantage of the pruning and learning techniques implemented in TRINITY and prune equivalent infeasible programs that share the same logical specifications.

Example 4.2. Consider the function `summarise`. This function aggregates the data in each group, which is composed by a set of rows. Let $r = \texttt{summarise}(a, \texttt{mean}, 2)$ be the output of running `summarise` on table `a`. For any execution of `summarise`, we know that the number of columns of the resulting table `r` will be at most the number of columns in table `a`. Moreover, the number of rows and groups in table `r` will equal the number of groups in table `a`. Hence, we can write the following logical specifications that describe the relation on the number of columns, rows and groups between table `r` and table `a`:

- columns(r) ≤ columns(a)
- rows(r) = groups(a)
- groups(r) = groups(a)

Third, since our output table has numerical imprecisions, we modified the search of the program synthesizer to enumerate all programs within a time limit that have an output table with the same number of rows and columns as the extracted table. Even though the extracted table has numerical imprecisions, the shape of the table is usually correct. Instead of selecting a single program, UNCHARTIT maintains a list of programs that satisfies the row and column constraints. All programs are ranked using the metrics from Section 5.

5 RANKING CANDIDATE SOLUTIONS

In order to rank the generated candidates, we assign a cost to each program: the highest-ranking program is the program with the lowest cost. In this section, we present two possible cost functions to rank the candidate programs.

Table 3: Re-scaled extracted table.

col_0	col_1
bar_1	0.0106
bar_2	0.3269
bar_3	0.4901
bar_4	0.6927
bar_5	0.7644
bar_6	0.2306

Table 4: Re-scaled output table.

year	# complaints
2011	0.0115
2012	0.3282
2013	0.4899
2014	0.6929
2015	0.7630
2016	0.2301

Recall that the data extraction mechanisms described in Section 3 do not extract labels, only the bar values. Thus, we only consider the numerical data extracted from the chart to calculate a program's cost. We propose to measure the quality of programs by comparing the extracted bar values to those of the program output. Before calculating the cost of the program, we re-scale the bar values using the axis maximum and minimum values of the chart. This scaling allows us to have a standardized range of costs independent of the chart's scale. We re-scale each bar to a value between 0 and 1 using the following function:

$$f(y) = \frac{y - L}{H - L} \tag{5}$$

where H and L are the axis maximum and minimum values, respectively. Two possible cost functions are the mean absolute error (MAE) and the mean squared error (MSE).

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |f(y_i) - f(\hat{y_i})|^1 \tag{6}$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} |f(y_i) - f(\hat{y_i})|^2 \tag{7}$$

where n is the number of bars, y_i is the i'th bar value of the program's output, and $\hat{y}_i$ is the i'th bar value obtained from the data extraction mechanism.

Example 5.1. Consider a recovery task where the data extraction mechanism generates Table 1 on the chart of Figure 1b. Consider also that a candidate program outputs Table 2 when applied to the input data. To calculate the program's cost we start by re-scaling the bars' values. In this case, the chart's maximum and minimum values are $H = 2.21 \times 10^5$ and $L = 0$, respectively. Thus, using the re-scaling function (5), we get Tables 3 and 4. Using the mean absolute error from (6), the program would have the following cost:

$$\begin{aligned}\text{MAE} =& \frac{1}{6}\Big(|0.0106 - 0.0115| + |0.3269 - 0.3282| + \\ & |0.4901 - 0.4899| + |0.6927 - 0.6929| + \\ & |0.7644 - 0.7630| + |0.2306 - 0.2301|\Big) = 0.00075\end{aligned}$$

In order to rank the candidates, the cost of each program is calculated and the candidates are ordered. If two different programs have the same cost, the smaller program[2] is ranked higher.

[2] A program $\mathcal{P}_1$ is considered smaller than a program $\mathcal{P}_2$ if $\mathcal{P}_1$ uses fewer operators from the DSL than $\mathcal{P}_2$.

```
1  int main() {
2    int a = read(); // (a_0 = α)
3    int b = read(); // (b_0 = β)
4    int c = 3;      // (c_0 = 3)
5
6    if (a+b == 3)   // (a_0 + b_0 ≠ 3) ⟹ (c_1 = c_0)
7      c += 10;      // (a_0 + b_0 = 3) ⟹ (c_1 = c_0 + 10)
8    return c;       // (o_P = c_1)
9  }
```

Figure 6: Symbolic representation of a program in C.

6 PROGRAM DISAMBIGUATION

The proposed ranking functions are helpful in selecting promising candidate programs. However, in some cases, the highest ranked program is not the desired solution. Therefore, given the top-n ranked programs, we propose to interact with the user in order to select a program that corresponds to the user's intent.

This section starts by briefly reviewing Satisfiability Modulo Theories (SMT) and how SMT can be used to formalize a symbolic execution of a program. Next, two different user interaction models are presented. For each interaction model, we formalize how to automatically generate an input test case that differentiates among the candidate programs. Finally, we refer how fuzzing techniques can also be used to this end.

6.1 Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) problem is a generalization of the well-known Propositional Satisfiability (SAT) problem. Given a decidable first-order theory $\mathcal{T}$, a $\mathcal{T}$-atom is a ground atomic formula in $\mathcal{T}$. A $\mathcal{T}$-literal is either a $\mathcal{T}$-atom t or its complement $\neg t$. A $\mathcal{T}$-formula is similar to a propositional formula, but a $\mathcal{T}$-formula is composed of $\mathcal{T}$-literals instead of propositional literals. Given a $\mathcal{T}$-formula ϕ, the SMT problem consists of deciding if there exists a total assignment over the variables of ϕ such that ϕ is satisfied. Depending on the theory $\mathcal{T}$, the variables can be of type integer, real, Boolean, among other domains.

The Maximum Satisfiability Modulo Theories (MaxSMT) is the optimization version of the SMT problem. In MaxSMT, the goal is to find an assignment that optimizes a given objective function, such that an SMT formula is satisfied. In the literature, MaxSMT is sometimes defined over a set of hard and soft formulas [30]. However, it can also be defined as optimizing an objective function [5]. For ease of understanding, we use the latter formalization.

6.2 Symbolic Representation of Programs

Symbolic execution is a technique that allows executing a program with symbolic values instead of concrete values. In essence, given a program $\mathcal{P}$, one can build an SMT formula $\phi_{\mathcal{P}}$ that represents the symbolic execution of $\mathcal{P}$. Hence, $\phi_{\mathcal{P}}$ represents all possible executions of program $\mathcal{P}$ when all possible input values are considered.

Example 6.1. Consider the program $\mathcal{P}$ in Figure 6 with two input variables (a and b). To generate an SMT formula to represent the symbolic execution of $\mathcal{P}$, we start by converting the program to a static single assignment (SSA) form. In SSA form, a new variable is created for each assignment in the program. For example, since variable c is assigned twice (lines 4 and 7), we create two instances of c: c_0, and c_1, used to represent the value of c after each assignment. Moreover, each input is assigned a symbolic value: $a_0 = \alpha$, and $b_0 = \beta$. Note that the symbolic values α and β represent all possible values that can be assigned to a and b, respectively. Finally, we build the SMT formula that represents the program's execution flow. For program $\mathcal{P}$ the formula is as follows:

$$
\begin{aligned}
\phi_{\mathcal{P}} := {} & (a_0 = \alpha) \land (b_0 = \beta) \land (c_0 = 3) \land \\
& ((a_0 + b_0 \neq 3) \implies (c_1 = c_0)) \land \\
& ((a_0 + b_0 = 3) \implies (c_1 = c_0 + 10)) \\
& (o_{\mathcal{P}} = c_1)
\end{aligned}
$$

Symbolic execution is often used to check a given property of a program. Let $\mathcal{P}$ be a program and $o_{\mathcal{P}}$ denotes the symbolic representation of the return value of $\mathcal{P}$. It is possible to check if there is an execution of $\mathcal{P}$ that returns 0 by using an SMT solver to check the satisfiability of ϕ_{ret0}, where $\phi_{ret0} = \phi_{\mathcal{P}} \land (o_{\mathcal{P}} = 0)$. Observe that if the SMT solver finds ϕ_{ret0} to be unsatisfiable, then there is no input of $\mathcal{P}$ such that $\mathcal{P}$ returns 0. Otherwise, if the SMT solver provides a satisfying assignment for ϕ_{ret0}, then the assignment to the symbolic representation of the inputs of $\mathcal{P}$ contains the concrete input values (i.e. the input test case) for when $\mathcal{P}$ returns 0.

Symbolic execution can also be used to differentiate between two programs $\mathcal{P}_1$ and $\mathcal{P}_2$. Let $\phi_{\mathcal{P}_i}$ be the SMT formula that corresponds to the symbolic execution of program $\mathcal{P}_i$. Let I_i represent the input and o_i the output of $\mathcal{P}_i$. Hence, we can built a formula ϕ_{eq} such as:

$$\phi_{eq} = \phi_{\mathcal{P}_1} \land \phi_{\mathcal{P}_2} \land (I_1 = I_2) \land (o_1 \neq o_2) \tag{8}$$

Observe that if ϕ_{eq} is satisfiable, then there is an input test case for which $\mathcal{P}_1$ and $\mathcal{P}_2$ provide different outputs. As a result, one can ask the user to answer which is the correct output and disambiguate between $\mathcal{P}_1$ and $\mathcal{P}_2$. Otherwise, if ϕ_{eq} is unsatisfiable, then there is no input test case that differentiates between $\mathcal{P}_1$ and $\mathcal{P}_2$ and the programs are deemed equivalent.

6.3 User Interaction Models

UnchartIt is able to rank candidate programs, but the best ranked program might not correspond to the user's intent. Moreover, it can occur that the ranking value is the same for two different programs. Hence, our goal is to interact with the user in order to correctly select the desired program among the top ranked candidates.

In UnchartIt, we define two different user interaction models. The Options model shows the user an input table, as well as several output options corresponding to the output of candidate programs for that input table. In this case, the user selects the correct output among the several options. If the selected output still corresponds to the output of several candidate programs, additional rounds of questions are performed to disambiguate solely among those programs. In the Options model, ideally, there is a single input table such that each candidate program produces a different output. In this best-case scenario, a single question is sufficient to disambiguate among the candidate programs. On the other hand, the Options model requires the user to solve the problem for the given input table in order to select the correct option.

$$\text{min. } \sum_{i=1}^{n}\sum_{j=i+1}^{n} b_{ij} \tag{9}$$
$$\text{s.t. } \phi_{\mathcal{P}_1} \wedge \ldots \wedge \phi_{\mathcal{P}_n} \tag{10}$$
$$\forall_{i,j\in\{1..n\},i<j} : I_i = I_j \tag{11}$$
$$\forall_{i,j\in\{1..n\},i<j} : (o_i = o_j) \Leftrightarrow (b_{ij}) \tag{12}$$
$$\bigvee_{i=1}^{n}\bigvee_{j=i+1}^{n} \neg b_{ij} \tag{13}$$

Figure 7: Input generation for the OPTIONS model.

UNCHARTIT also implements the Y/N user interaction model. In this case, the user is presented with an input table and an output. Next, the user answers yes or no, depending if the output is correct for that input table. Note that the user only needs to check the correctness of a single output option. In the Y/N model, the goal is to split the set of candidate programs in two, such that the output of half the candidate programs matches the proposed output, while the other half produces a different output. If it is always possible to split the set of programs in two, the number of questions in the Y/N interaction model would be $O(lg(n))$.

6.4 Model Formalization

Section 6.3 presented the OPTIONS and the Y/N user interaction models implemented in UNCHARTIT. In this section, we propose two MaxSMT formalizations that allow us to automatically generate input examples for both user models.

In the OPTIONS user model, in order to minimize the number of user interactions, the goal is to find a small input test case such that all the top-n ranked programs provide a different output. Figure 7 presents a MaxSMT formulation to solve the problem of finding an input that maximizes the pairwise differences between the n programs to disambiguate. In this formula, we encode the symbolic representation of all n candidate programs (10) and force the input of all programs to be the same (11). Moreover, for each pair of programs $\mathcal{P}_i$ and $\mathcal{P}_j$ we create a Boolean variable b_{ij} that is assigned to 1 if and only if the outputs of programs $\mathcal{P}_i$ and $\mathcal{P}_j$ are the same (12). Note that inputs that do not differentiate any pair of programs are excluded (13). Since the goal is to minimize the number of variables b_{ij} assigned to 1 (9), any optimal solution of this formulation will find an assignment to the input variables I_i (corresponding to an input test case) that maximizes the pairwise difference between the n programs. Ideally, the solution for the formulation in Figure 7 contains all variables b_{ij} assigned value 0.

On the Y/N interaction model, the goal is to identify an input test case I such that the set of n programs is split into two sets A and B with half programs in each set. Moreover, for test case I, all programs $\mathcal{P}_i \in A$ would provide the same output $\mathcal{P}_i(I)$, and all programs $\mathcal{P}_j \in B$ would provide a different output (i.e. $\mathcal{P}_i(I) \neq \mathcal{P}_j(I)$) than the programs in A.

Figure 8 contains a formulation that splits a given set of n programs into two sets (A and B). As in the previous model, this formulation includes the symbolic representation of all n programs (15), the program inputs are constrained to be the same (16) and Boolean

$$\text{min. } \left|\sum_{i=1}^{n} p_i^A - \sum_{i=1}^{n} p_i^B\right| \tag{14}$$
$$\text{s.t. } \phi_{\mathcal{P}_1} \wedge \ldots \wedge \phi_{\mathcal{P}_n} \tag{15}$$
$$\forall_{i,j\in\{1..n\},i<j} : I_i = I_j \tag{16}$$
$$\forall_{i,j\in\{1..n\},i<j} : (o_i = o_j) \Leftrightarrow (b_{ij}) \tag{17}$$
$$\forall_{i,j\in\{1..n\},i<j} : (b_{ij}) \Rightarrow \left((p_i^A \wedge p_j^A) \vee (p_i^B \wedge p_j^B)\right) \tag{18}$$
$$\forall_{i,j\in\{1..n\},i<j} : (\neg b_{ij}) \Rightarrow (\neg p_i^A \vee \neg p_j^A) \tag{19}$$
$$\forall_{i\in\{1..n\}} : p_i^A + p_i^B = 1 \tag{20}$$
$$\sum_{i=1}^{n} p_i^B \leq n-1 \tag{21}$$

Figure 8: Input generation for the Y/N model.

variables b_{ij} are assigned to 1 if and only if the output of program $\mathcal{P}_i$ is equal to the output of program $\mathcal{P}_j$ (17). Additionally, for each program $\mathcal{P}_i$ two new Boolean variables are created p_i^A and p_i^B, denoting if program $\mathcal{P}_i$ belongs to set A or to set B, respectively.

In our formulation, if two programs $\mathcal{P}_i$ and $\mathcal{P}_j$ produce the same output, then they both have to be assigned to the same set (18). Moreover, if two programs $\mathcal{P}_i$ and $\mathcal{P}_j$ produce different outputs (i.e. variable b_{ij} is 0), then at most one of them can be in set A (19). Therefore, as a result of constraints (18) and (19), all programs in set A must produce the same output. Furthermore, each program must be assigned to one and only one set (20). Constraint (21) is used to make sure that if there is an input that differentiates among programs, then not all programs are assigned to set B and a partition is produced. Finally, our formulation's goal is to minimize the difference between the number of programs in each set (14).

6.5 Input Constraints

In the symbolic representation of a program, each input is associated with a symbolic value that represents an arbitrary concrete value that can be assigned to that input. Since the inputs of our programs are tables, each symbolic value represents a table with a number of rows and columns. However, allowing input tables with an arbitrary structure can be a problem. For instance, it would not be feasible to ask the user to verify or select the correct output for a large input table. Therefore, we impose restrictions on the structure of the input tables we allow in the symbolic representation of our programs. In our case, the columns are restricted to those that are relevant in at least one of the programs to disambiguate. For instance, if we want to disambiguate 2 programs that only use the first and the last columns of the input table, then the input table to be generated only contains data for those 2 columns. Moreover, since we want to obtain small input tables, the number of rows must also be bounded.

The table's content must also be restricted since each table entry should be associated with a meaningful value to the user. For instance, if the user expects a given column to contain country names, then the only concrete values we should allow on that column are country names. In order to generate inputs that are familiar to the user, we base our distinguishing inputs on the input table the user

provided. In UnchartIt, the following rules are used to decide the available values for each column: (a) in columns of strings we restrict the available values to those present in the respective column of the input table; (b) in columns of integers, floats, and dates we restrict the values to the interval between the minimum and maximum values of the respective column of the input table.

6.6 Input Generation

The MaxSMT formulations proposed in section 6.4 give us a theoretical guarantee that the resulting distinguishing input is the best possible input for the respective interaction model. However, an issue with both approaches is that our MaxSMT formulas grow exponentially with the number of programs to disambiguate. In scenarios where it is necessary to disambiguate a large number of programs, one might sacrifice optimality in order to have a meaningful user interaction.

There is a plethora of input generation methods commonly used for program testing [9]. For example, in the context of UnchartIt, one could apply delta debugging [42, 43] on the example input table to try to generate a smaller input table that would differentiate the programs. However, the input tables provided in our test cases can be very large, resulting in a very time-consuming procedure.

Another alternative is to use fuzzing-based methods [31]. Instead of building a MaxSMT formula, we can generate random inputs (guided by the example input table) until we find an optimal solution for a given interaction model or a time limit is reached. For example, using the Options interaction model, we can randomly generate inputs until an example that disambiguates all programs is found. Otherwise, if a time limit is reached, the generated input that splits the programs in a larger number of sets is returned. In UnchartIt, this technique was also implemented as a stand alone method to disambiguate programs. Moreover, a hybrid method was also developed that first applies fuzzing-based techniques and then applies the proposed MaxSMT models when the number of programs to disambiguate is small.

7 EXPERIMENTAL RESULTS

In order to evaluate our approach, we collected 50 benchmarks from Kaggle,[3] a popular website for data scientists with diverse open datasets. Each benchmark is comprised of a pair (table, bar chart). The experimental results presented in this section aim to answer the following research questions:

Q1. How effectively can UnchartIt recover programs from real data?
Q2. How long do we have to explore the search space to find good candidates?
Q3. How does the Neural Network approach compare to the WebPlotDigitizer's approach?
Q4. How many questions do we have to ask the user in order to distinguish the best ranking programs, using the two interaction models?

The results described herein were obtained from an Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz, with 64GB of RAM, running Debian GNU/Linux 10.

[3]https://www.kaggle.com/

Implementation. UnchartIt integrates several tools and technologies. In particular, our neural data extraction mechanism is implemented using the Keras framework [6]. Furthermore, our candidate generator is implemented on top of the Trinity synthesis framework [28]. While the candidate generator uses the `R` language (version 3.5.2), the program disambiguation is performed in `C`. For that, all of our DSL operators have an equivalent implementation in `C` so that the symbolic representation of the programs can be generated using CBMC [7], a Bounded Model Checker for `C`. Since CBMC generates Boolean formulas, the final MaxSMT formula only contains Boolean variables. As a result, the Open-LinSBPS [8] solver was used instead of a generic MaxSMT solver. Finally, the number of rows of the generated input tables was bounded to 5.

Benchmarks. The average and median size of the input table files is 16.52MB and 1MB, respectively. However, there are much larger instances in our benchmark set. The motivational example in Section 2 has one of the largest input tables (Table 1a), containing 175.39MB, 715,437 rows and 18 columns. Moreover, the median number of rows and columns is 10,841 and 13, respectively, whereas the mean number of rows and columns is 71,293.92 and 17.66, respectively. Regarding the bar charts, the number of bars of each chart varies between 2 and 15 bars. Every solution involves grouping the data by some column, and then summarizing each group using an aggregate function (e.g., `median`, `min` or `max`). Some solutions require operations such as calculating the days between two dates or filtering *null* values. It might also be necessary to normalize the values of a numerical column, or selecting only the top ranking rows.

Data Extraction. When evaluating a data extraction procedure for bar charts we must consider its two outputs: the number of bars, and the bars' values. Thus, to measure its accuracy we use two metrics: the percentage of plots in which the procedure successfully retrieved the number of bars, and the mean absolute error of the bar's values. To test both WebPlotDigitizer and the neural network we used the bar charts of the benchmarks.

Recall that WebPlotDigitizer requires a considerable amount of input. Before extracting the bars' values, it is necessary to mark the pixel location of two different points along the vertical axis of each bar chart and the values of the respective points. WebPlotDigitizer's bar extraction algorithm also requires tuning parameters before extracting the bars. It was found that the parameters that worked best with our benchmarks were $\Delta x = 30$ and $\Delta val = 500$. Using these parameters, WebPlotDigitizer successfully retrieved the number of bars in 96% of the instances and achieved a mean absolute error of 0.002201.

To evaluate the accuracy of the neural-based data extraction, we trained an adapted version of the EfficientNet-B1. We generated a set of 90,000 bar charts of various forms and split it into training (90%) and validation (10%) sets. To train the network, we used RAdam [27] coupled with Lookahead [44] using the default parameters of the respective papers. We used batch sizes of 15 and a maximum number of epochs of 100, but we performed early stopping once the validation loss stopped decreasing. On the benchmarks, EfficientNet-B1 retrieved the correct number of bars in 92% of the instances. Considering the mean absolute error, the network has a

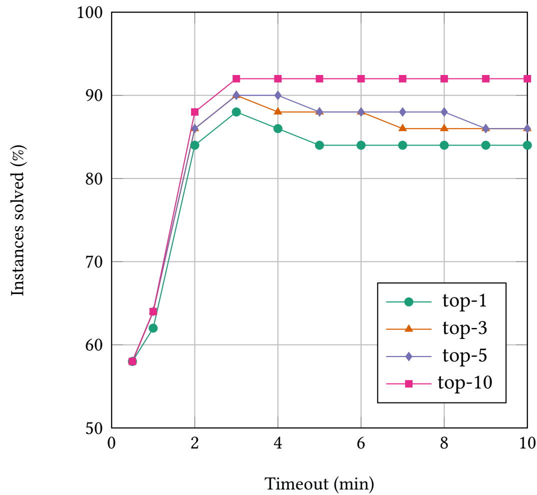


Figure 9: Success rate with different timeouts, using WebPlotDigitizer and the MAE ranking.

Table 5: Success rate for a time limit of 3 minutes.

	WebPlotDigitizer		EfficientNet-B1	
	MAE	MSE	MAE	MSE
top-1	88%	86%	66%	66%
top-3	90%	88%	72%	72%
top-5	90%	88%	76%	74%
top-10	92%	92%	78%	80%

mean absolute error of 0.037356. Although the adapted EfficientNet-B1 is not as accurate as WebPlotDigitizer, it is important to note that it requires significantly less input from the user.

Candidate Generation and Ranking. Since it is not feasible to explore the whole program space, UnchartIt terminates when a given time limit is reached. In Figure 9, we show the success rate for different timeouts. The top-1, top-3, top-5, and top-10 lines show the number of benchmarks in which the correct solution was ranked first (top-1), among the first three (top-3), five (top-5), and ten (top-10) programs, respectively. We can see that UnchartIt performs best when using a timeout of 3 minutes, and it does not improve thereafter. In fact, the percentage of correct programs in top-1 decreases with higher time limits. This occurs due to the fact that if we explore the search space longer, we are more prone to finding programs that overfit to the cost function (especially programs with a high number of lines). Moreover, there might be other programs with more lines of code that are equivalent to the overfitting (e.g., adding a filter operation that does nothing on the input table). Since these spurious programs have the same cost of the overfitting program, they push the solution downwards. Overall, UnchartIt is able to find programs up to 7 lines of code within the time limit, which is the same order to magnitude as other state of the art tools for table manipulation [14].

Table 5 shows the success rate with a timeout of 3 minutes, using the two data extraction mechanisms, and the two ranking functions. We can see that both ranking functions perform similarly, regardless of the data extraction mechanism. The correct solution is the top ranked program in 88% of the instances when using WebPlotDigitizer, and the MAE ranking function. Using the adapted EfficientNet-B1 neural network we obtain slightly worse results. Nonetheless, we can still rank the correct solution on the top-10 in 80% of the instances. When using WebPlotDigitizer this value increases to 92%. Recall that WebPlotDigitizer is more precise than EfficientNet-B1 with respect to the numerical extracted values and number of extracted bars. However, EfficientNet-B1 is a fully automated process, while WebPlotDigitizer needs the user to indicate the precise location of two pixels in the chart image and tune some parameters before extracting data.

In the best performing approach (WebPlotDigitizer + MAE), there are 8% instances in which a correct solution was not ranked among the top-10. These benchmarks correspond to 2 instances in which the number of bars was incorrectly extracted and 2 instances in which 3 minutes is not sufficient to find a correct candidate.

Program Disambiguation. To ascertain that UnchartIt returns a correct program, the top-10 ranking programs are to be disambiguated by interacting with the user. UnchartIt integrates two interaction schemes: the Options and the Y/N model. For each model, questions can be generated using the following approaches: (1) MaxSMT; (2) Fuzzing; (3) Hybrid Approach. In the hybrid approach, we combine fuzzing and MaxSMT as follows: if we need to disambiguate more than 5 programs, then we use fuzzing. Otherwise, we use MaxSMT. In our experiments, we consider the top-10 programs (using WebPlotDigitizer) generated for each instance. From the 50 instances, we consider 48 instances, since for one instance we only generated one candidate, and there was another instance for which we did not generate a single candidate.

Figure 10 shows the average time necessary to generate the best possible question with a timeout of 3 minutes per question. We can see that when using fuzzing, we either find the best question very quickly, or we cannot find it within the time limit. In the Options model, fuzzing can only stop early when it finds an input test case for which all programs provide a different output. However, that input test case might not exist. The same occurs for the Y/N model, where an input test case that splits the set of programs in half might not exist. However, the proposed MaxSMT formulation is able to detect these cases. We can also see that the hybrid approach generates questions faster than the MaxSMT approach. This happens because the formulas generated by CBMC grow exponentially with the number of programs to disambiguate. Thus, the first MaxSMT call usually takes much longer than the remaining calls. However, fuzzing is particularly effective when the number of programs is larger. Hence, by using fuzzing in the first call, we reduce the time necessary to generate the first question, thereby reducing the average time to generate all questions.

Table 6 presents statistics on the number of questions asked to the user using the two interaction models and the three different implementations. Observe that we can disambiguate 47 out of the 48 instances using the hybrid approach. Although the average number

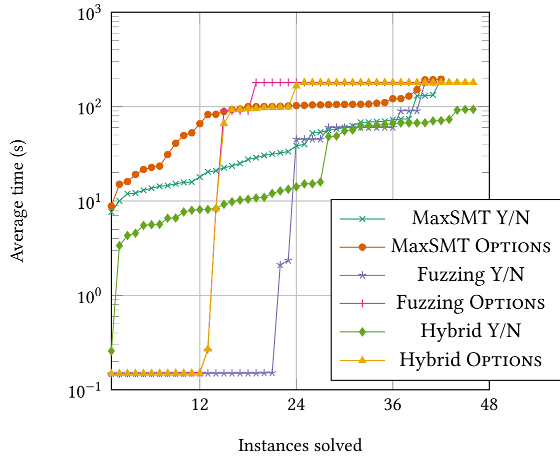


Figure 10: Average time necessary to generate a question using the different interaction models and implementations.

Table 6: Median ($\tilde{x}$), mean ($\overline{x}$), standard deviation (σ) of the number of questions asked. Number of instances solved (n).

	MaxSMT		Fuzzing		Hybrid	
	Options	Y/N	Options	Y/N	Options	Y/N
$\tilde{x}$	1	3	1	3	1	3
$\overline{x}$	1.53	3.28	1.17	3	1.32	3.23
σ	0.631	0.854	0.537	0.698	0.556	0.813
n	43	43	42	42	47	47

of questions using fuzzing is slightly smaller, fuzzing can only disambiguate 42 instances, since it cannot prove the bounded program equivalence. Thus, fuzzing presents inconclusive results to the user in 6 instances. The same happens to the MaxSMT approach, where sometimes the given time limit is not enough to prove the program equivalence for the bounded input.

Threats to Validity. Since our tool is limited to bar charts, our techniques may not generalize for other types of charts. For other types of charts, the data extraction stage must be adapted. However, if the data extraction procedure from other chart types results in imprecisions similar to those found in bar charts, one can expect a similar success rate.

The other issue is the simulation of the user interaction. In this paper, we assume the user would select the correct answer in each question. However, it is not clear how difficult it is for the user to answer the generated questions, since an empirical study of user interaction was not performed. Additionally, we bound the time limit to 3 minutes in order to generate a question. For tighter time limits, the MaxSMT solver might produce a solution that is far from optimal. As an alternative, an incomplete solver might be used instead. In general, incomplete solvers cannot prove optimality, but are able to provide a good enough solution within tighter time limits.

8 RELATED WORK

In this section, we briefly discuss prior work that is closely related to our approach, in the context of program verification, program synthesis, and interactive program synthesis.

8.1 Program Verification

The goal of program verification is to formally prove that a certain specification or property holds for all executions of the program. The last few decades have seen a significant improvement in verification tools based on SAT and SMT [4]. In this work, we leverage bounded model checking tools [7, 16] to either prove the equivalence between programs or find a counterexample that disambiguates the programs. In our context, since the tables for the disambiguation phase are small, it is possible to completely unroll all loops and check if programs are equivalent to a bounded input. Even though program equivalence of C programs has been studied before [12, 18, 19], to the best of our knowledge this is the first application of it for disambiguation of programs written in a real-world programming language in the context of program synthesis.

8.2 Program Synthesis

Program synthesizers for table transformations work by combining enumerative search and pruning techniques over a space of programs defined by a DSL. Scythe [39] generates `SQL` queries from examples and prunes the search using equivalence classes. Morpheus [14] synthesizes table transformations for the `R` language and uses logical specifications for each library function combined with SMT-based reasoning to prune the search space. Neo [13] generalizes Morpheus to other domains and incorporates learning from failed synthesis attempts which further prunes the search space. Trinity [28] is a program synthesizer framework that makes it easier to build new program synthesizers while taking advantage of pruning and learning techniques based on SMT reasoning [13, 14].

Viser [40] is built on top of Trinity for the domain of plot visualization. It takes as input a table and a trace that partially describes the plot. For instance, in the case of a bar chart, the trace describes the height of some bars. The specification used in Viser is not as strong as in traditional PBE systems since it does not involve a concrete output table but a set of table inclusion constraints. In contrast, UnchartIt takes as input a chart image instead of a trace. Given a chart image, we perform data extraction and our output table will have numerical imprecisions that are not part of the result of the table transformation program. Moreover, since we are tackling the problem of recovering data transformations from existing plots, the user would not be able to provide the trace of the plot required by Viser.

Another application of program synthesis to visualization is the inference of graphics programs from hand-drawn images and synthesis of the corresponding LaTeX code that generates that image [11]. This approach combines techniques from deep learning and program synthesis. They learn a convolutional neural network that proposes a set of traces in the form of primitive drawings (e.g., line, circle, rectangle) that explains the image. These primitive drawings serve as specification, and a program synthesizer is then used to generate a program that generalizes these traces with conditionals and loops. Even though our approach can also use a neural

network for data extraction, our synthesis goal is very different since it requires a sequence of table transformation operations and not trace generalization.

8.3 Interactive Program Synthesis

Since PBE systems have incomplete specifications, it is often required to do an interactive step with the user in order to find the correct program. There are different forms of user interaction, but the most commonly used by program synthesizers are: (i) the user provides additional examples to the program synthesizer until there is no more ambiguity [29, 41], (ii) the synthesizer returns a ranked list of programs to be selected by the user [17, 26, 41], (iii) the synthesizer creates a distinguishing input and asks the user for feedback [29, 32, 38].

There are different ways to create a distinguishing input, i.e. an input for which at least two programs have a different output. One approach is to randomly generate distinguishing inputs [29, 38]. This is similar to our input generation approach described in Section 6.6. Another approach that is closer to our work is done by Ji et al. [22]. They sample the space of valid programs and encode the problem into SMT to determine an input that minimizes the number of programs that have the same output for a given input. Afterward, they ask the user to provide the correct output for that input. This approach is similar to our Options model where we also minimize the number of different outputs for the same input. Our interactive approach can be seen as a generalization of Ji et al. [22] work. First, we show how to formalize the optimization problem with MaxSMT. Second, we show that different user interactions can be formalized in this way, namely the Options and Y/N user interaction models. Third, we use symbolic model checking to encode programs written in real-world programming languages to SMT, whereas the previous approach uses programs from the Syntax-Guided Synthesis Competition (SyGuS) [1] that are expressed using the SMT language and restricted to SMT constructs.

9 CONCLUSIONS AND FUTURE WORK

Data visualization is crucial for data analysts. However, many data analysts are not proficient programmers and it is often the case that data analysts are unable to generate a given chart from the data.

The main contribution of this paper is a comprehensive approach to handle the problem of recovering data transformations from charts. UnchartIt receives the input data and an output chart (generated from the input data) and can automatically find the underlying table transformation to build the chart. Experimental results on real data from Kaggle show that UnchartIt can find and rank the correct program in the top-10 programs in 92% of instances. To reduce the ambiguity of the programs returned by UnchartIt, we developed a new interactive synthesis procedure that can disambiguate 98% of the ambiguous instances by asking on average fewer than 2 questions to the user.

UnchartIt is the first tool for automatic generation of data transformations that directly uses visual elements. An integrated prototype of our tool will become available online soon.[4]

For future work, we propose to extend UnchartIt with other visual elements in the input examples besides bar charts. Ideally, one should be able to use an hand-drawn image of a chart instead of a digital chart image. Currently, the chart labels are not yet used. However, labels provide useful information on the chart interpretation. Hence, we also propose to extend the data extraction model to identify and use the labels in the chart image to direct the program synthesis process.

[4] http://sat.inesc-id.pt/unchartit/

ACKNOWLEDGMENTS

This work was partially supported by NSF award number 1762363 and by Portuguese national funds through FCT, Fundação para a Ciência e a Tecnologia, under PhD grant SFRH/BD/150688/2020 and projects UIDB/50021/2020, DSAIPA/AI/0044/2018, and project ANI 045917 funded by FEDER and FCT.

REFERENCES

[1] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. IOS Press, 1–25.
[2] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proc. International Conference on Learning Representations.*
[3] Leilani Battle, Peitong Duan, Zachery Miranda, Dana Mukusheva, Remco Chang, and Michael Stonebraker. 2018. Beagle: Automated Extraction and Interpretation of Visualizations from the Web. In *Proc. Conference on Human Factors in Computing Systems*. ACM, 594.
[4] Dirk Beyer, Matthias Dangl, and Philipp Wendler. 2018. A Unifying View on SMT-Based Software Verification. *Journal of Automated Reasoning* 60, 3 (2018), 299–335.
[5] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. *ν*Z - An Optimizing SMT Solver. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 194–199.
[6] François Chollet et al. 2015 (accessed May 8, 2020). Keras. https://keras.io.
[7] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176.
[8] Emir Demirovic and Peter J. Stuckey. 2019. Techniques Inspired by Local Search for Incomplete MaxSAT and the Linear Algorithm: Varying Resolution and Solution-Guided Search. In *Proc. International Conference Principles and Practice of Constraint Programming*. Springer, 177–194.
[9] Frank Elberzhager, Alla Rosbach, Jürgen Münch, and Robert Eschbach. 2012. Reducing test effort: A systematic mapping study on existing approaches. *Inf. Softw. Technol.* 54, 10 (2012), 1092–1106.
[10] Kevin Ellis and Sumit Gulwani. 2017. Learning to Learn Programs from Examples: Going Beyond Program Structure. In *Proc. International Joint Conference on Artificial Intelligence*. ijcai.org, 1638–1645.
[11] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Proc. Annual Conference on Neural Information Processing Systems*. 6062–6071.
[12] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *Proc. International Conference on Automated Software Engineering*. ACM, 349–360.
[13] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 420–435.
[14] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 422–436.
[15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 229–239.
[16] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. 2018. ESBMC 5.0: An Industrial-Strength C Model Checker. In *Proc. International Conference on Automated Software Engineering*. ACM, 888–891.
[17] Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. 2014. CodeHint: dynamic and interactive synthesis of code snippets. In *Proc. International Conference on Software Engineering*. ACM, 653–663.
[18] Benny Godlin and Ofer Strichman. 2008. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica* 45, 6 (2008), 403–439.

[19] Benny Godlin and Ofer Strichman. 2009. Regression verification. In *Proc. Design Automation Conference*. ACM, 466–471.

[20] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 317–330.

[21] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119.

[22] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question Selection for Interactive Program Synthesis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.

[23] Zhongjun Jin, Michael R. Anderson, Michael J. Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *Proc. International Conference on Management of Data*. ACM, 683–698.

[24] Daekyoung Jung, Wonjae Kim, Hyunjoo Song, Jeongin Hwang, Bongshin Lee, Bo Hyoung Kim, and Jinwook Seo. 2017. ChartSense: Interactive Data Extraction from Chart Images. In *Proc. Conference on Human Factors in Computing Systems*. ACM, 6706–6717.

[25] Dmitri V. Kalashnikov, Laks V. S. Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *Proc. International Conference on Management of Data*. ACM, 337–350.

[26] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proc. SIGCHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 3363–3372.

[27] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. 2019. On the Variance of the Adaptive Learning Rate and Beyond. *CoRR* abs/1908.03265 (2019).

[28] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An Extensible Synthesis Framework for Data Science. *PVLDB* 12, 12 (2019), 1914–1917.

[29] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proc. Symposium on User Interface Software & Technology*. ACM, 291–301.

[30] Robert Nieuwenhuis and Albert Oliveras. 2006. On SAT Modulo Theories and Optimization Problems. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*. Springer, 156–169.

[31] Peter Oehlert. 2005. Violating Assumptions with Fuzzing. *IEEE Secur. Priv.* 3, 2 (2005), 58–62.

[32] Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd D. Millstein. 2018. FlashProfile: a framework for synthesizing data profiles. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 150:1–150:28.

[33] Mohammad Raza and Sumit Gulwani. 2017. Automated Data Extraction Using Predictive Program Synthesis. In *Proc. AAAI Conference on Artificial Intelligence*. AAAI Press, 882–890.

[34] Ankit Rohatgi. 2019 (accessed May 8, 2020). WebPlotDigitizer, Version 4.2. https://automeris.io/WebPlotDigitizer.

[35] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[36] Manolis Savva, Nicholas Kong, Arti Chhajta, Fei-Fei Li, Maneesh Agrawala, and Jeffrey Heer. 2011. ReVision: automated classification, analysis and redesign of chart images. In *Proc. Annual ACM Symposium on User Interface Software*. ACM, 393–402.

[37] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proc. International Conference on Machine Learning*. 6105–6114.

[38] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Interactive Query Synthesis from Input-Output Examples. In *Proc. International Conference on Management of Data*. ACM, 1631–1634.

[39] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 452–466.

[40] Chenglong Wang, Yu Feng, Rastislav Bodík, Alvin Cheung, and Isil Dillig. 2020. Visualization by example. *PACMPL* 4, POPL (2020), 49:1–49:28.

[41] Kuat Yessenov, Shubham Tulsiani, Aditya Krishna Menon, Robert C. Miller, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A colorful approach to text processing by example. In *Proc. Symposium on User Interface Software and Technology*. ACM, 495–504.

[42] Andreas Zeller. 2001. Automated Debugging: Are We Close. *IEEE Computer* 34, 11 (2001), 26–31.

[43] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.

[44] Michael R. Zhang, James Lucas, Jimmy Ba, and Geoffrey E. Hinton. 2019. Lookahead Optimizer: k steps forward, 1 step back. In *Proc. Annual Conference on Neural Information Processing Systems*. 9593–9604.

[45] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *Proc. International Conference on Automated Software Engineering*. IEEE, 224–234.