Using Dynamic Monitoring to Synthesize Models of Applications That Access Databases

JIASI SHEN, MIT EECS & CSAIL, USA
MARTIN RINARD, MIT EECS & CSAIL, USA

1 INTRODUCTION

We previously developed KONURE [Shen and Rinard 2018], a tool that uses active learning to infer the functionality of database applications. An alternative approach is to observe the inputs, outputs, and database traffic from a running system in normal use and then synthesize a model of the application from this information. To evaluate these two approaches, we present ETCH, which uses information from typical usage scenarios to synthesize a model of the functionality of database applications whose computation can be expressed in the KONURE DSL. ETCH observes the user inputs, database traffic, and outputs.

1.1 Input Collection Methodology

To collect the runtime inputs and outputs, ETCH sets up a proxy between the user and the application. This proxy records each user-provided input, along with a snapshot of the database contents when the input is sent to the database.

A user interacts with the application as follows. Initially, the user may populate the contents of the backend database as they desire. Then, the user launches the application and sends commands to the application via the ETCH proxy. For applications with a graphical user interface, the user clicks around on the interface, following the visual instructions provided by the application. For applications with a command line interface, the user executes the commands in a way that a typical user would use the application. If desired, the user may shut down the applications and populate the database with new contents, then restart the applications. The user stops interacting with the application when they reaches an understanding of the application's functionality that is satisfactory for a typical user.

During these user interactions, the user-provided inputs (including the corresponding database snapshots) are collected by ETCH. ETCH then uses these user-provided inputs to construct a program that is consistent with the observed functionality of the application.

1.2 Definitions

In the discussion below, we follow the terminology used in the KONURE technical report [Shen and Rinard 2018].

Each user-provided input, along with the snapshot of the database contents, forms a *value assignment*. Recall [Shen and Rinard 2018] that a value assignment contains information about the values to assign to the inputs and the database. The input information maps an input parameter to a concrete value. The database information maps a database location (identified by a table, a row number, and a column) to a concrete value.

Each such value assignment is used by ETCH to execute the application. The application, during execution, interacts with an external database using SQL queries. ETCH intercepts all these queries and uses them, along with the value assignment, to construct an *abstract trace*. ETCH then converts this abstract trace into a list of *deduplicated traces* (Algorithm 2).

Authors' addresses: Jiasi Shen, MIT EECS & CSAIL, USA, jiasi@csail.mit.edu; Martin Rinard, MIT EECS & CSAIL, USA, rinard@csail.mit.edu.

Algorithm 1

Input: U is a list of value assignments provided by the human user.

Output: A program in the Konure DSL that is consistent with the observed functionality of the application executing with U.

```
 1: procedure Construct($U$)
 2:     $P \leftarrow$ Nil
 3:     for $A \in U$ do
 4:         $F \leftarrow$ FlattenedTraces($A$)
 5:         for $\langle T, l \rangle \in F$ do
 6:             $P \leftarrow$ MergeTrace($T, l, 1, P$)
 7:         end for
 8:     end for
 9:     return $P$
10: end procedure
```

Etch uses all the user-provided value assignments to execute the application and collect traces. Then, Etch uses all the collected traces to construct a program, in the Konure DSL, that is consistent with the observed functionality of the application (Algorithm 1). Finally, Etch converts the Konure DSL program into an executable Python program.

1.3 Difference Between Etch and Konure

A key difference between Etch and Konure is the set of value assignments used for executing the application. Etch uses the value assignments that a human user provides. Konure, in contrast, uses active learning to automatically choose the value assignments that are important for the algorithm. Therefore, if we use Etch and Konure to analyze the same application, these two tools may generate different programs.

Recall that Konure explores all paths of the application to identify a unique equivalence class of Konure DSL programs. The human Etch user, however, may not explore the application as exhaustively as Konure does. Hence, the quality of the Etch-constructed program depends heavily on how the human Etch user explores various paths of the application.

Conversely, if the human user provides Etch with the same set of value assignments that Konure chooses, then Etch and Konure will generate equivalent programs.

2 THE ETCH CONSTRUCTION ALGORITHM

The Etch construction algorithm takes a list of value assignments, uses them to execute the application, and produces a Konure DSL program that is consistent with all these executions. The entry point is the Construct procedure in Algorithm 1.

To process each value assignment, the first step is to execute the application and collect a list of deduplicated traces from the execution. For this purpose, Etch calls the FlattenedTraces procedure in Algorithm 2. Recall from the Konure technical report [Shen and Rinard 2018] that procedure GeneralizeTrace converts a concrete trace into an abstract trace. Also recall that procedure DetectLoops detects loops in the abstract trace and returns the location for each iteration of each loop. The procedure AllWaysToKeepOneIteration is similar to the procedure KeepOneIteration in the Konure technical report [Shen and Rinard 2018], except that it returns all the legitimate deduplicated traces (instead of returning only one deduplicated trace). Specifically, for each loop found in the abstract trace, the procedure AllWaysToKeepOneIteration chooses

Algorithm 2

Input: A is a value assignment for the input parameters and the database contents.

Output: F is a list of pairs ($\langle T, l \rangle$) that represent all the deduplicated traces collected from executing the application with A. In each pair in F, T is a deduplicated trace that contains only one iteration of any loop. l is a list of locations in T that are originally found to be the starting location of loops in the abstract trace.

```
1: procedure FlattenedTraces($A$)
2:     $\langle \sigma_i, \sigma_d \rangle \leftarrow A$
3:     Populate the database with contents as specified in $\sigma_d$
4:     Execute the application with input parameters as specified in $\sigma_i$
5:     $T_c \leftarrow$ The concrete trace collected from the execution
6:     $T_a \leftarrow$ GeneralizeTrace($T_c$)                            ▷ Abstract trace
7:     $l_a \leftarrow$ DetectLoops($T_a$)
8:     $F \leftarrow$ AllWaysToKeepOneIteration($T_a, l_a$)      ▷ List of all deduplicated traces
9:     return $F$
10: end procedure
```

each iteration of the loop in turn, recursively entering the chosen iteration. For each legitimate deduplicated trace, the procedure keeps track of the locations that were originally the starting locations of loops. The procedure returns the list of all deduplicated traces along with their loop information.

Etch uses these traces to construct a program in the Konure DSL. The algorithm starts with an empty program and progressively updates the program with the given traces, one at a time, by calling the procedure MergeTrace in Algorithm 3. This procedure incorporates a new trace into an existing program, so that the resulting program is consistent with all traces seen in the history. This procedure performs a pattern match on the existing program and then invokes procedures in Algorithms 4, 5, 6, 7, and 8 accordingly. The procedure SameSkeleton returns whether two queries share the same skeleton, that is, if they may be generalized from the same SQL query in the application. When two queries share the same skeleton, the procedure MergeQuery uses this skeleton to construct a new query where the source locations (the ASrc nonterminal symbol for abstract traces [Shen and Rinard 2018]) are the intersections of those used by the two incoming queries. This procedure also collects metadata on historical results retrieved from these queries. The procedure MergedResults looks up and returns the metadata collected by the procedure MergeQuery.

After Etch returns from the Construct procedure in Algorithm 1, Etch uses the constructed Konure DSL program to generate an executable Python program. Note that queries in the constructed Konure DSL program may still have ambiguity regarding source locations. Specifically, a query may potentially refer to multiple source locations that are not equivalent. In this case, some of the potential source locations are correct while others are not. Etch enumerates all the potential choices until finding a program that produces outputs consistent with all the observed executions. Etch then returns this program's Python version.

Algorithm 3

Input: T is a trace collected from the FLATTENEDTRACES procedure, which contains a list of query-result pairs ($\langle q, r \rangle$) collected from an execution of the application. In each query-result pair, the query (q) is an instance of the nonterminal symbol "AQuery" in the syntax for abstract traces that represents an SQL query that the application sent to the database. The result (r) is an integer representing the number of rows retrieved by the corresponding query.

Input: l is the list of loop information for T.

Input: i is a positive integer.

Input: P is the fragment of the application, expressed in the KONURE DSL that corresponds to the tail of T starting from the i-th query.

Output: The fragment of the application, expressed in the KONURE DSL, after updating P with the tail of trace T starting from the i-th query.

```
procedure MergeTrace($T, l, i, P$)
    if $P$ : "Nil" then                                  ▷ $P$ is an empty program
        return MergeTraceNil($T, l, i$)
    else if $P$ : "$q^P$" then                            ▷ $P$ contains a single query
        return MergeTraceEnd($T, l, i, P$)
    else if $P$ : "$q_1^P\ q_2^P\ b^P$" then              ▷ $P$ starts with at least two queries
        return MergeTraceSequence($T, l, i, P$)
    else if $P$ : "if $q^P$ then $b_t^P$ else $b_f^P$" then   ▷ $P$ starts with a conditional statement
        return MergeTraceIf($T, l, i, P$)
    else if $P$ : "for $q^P$ do $b_t^P$ else $b_f^P$" then    ▷ $P$ starts with a loop
        return MergeTraceFor($T, l, i, P$)
    end if
end procedure
```

Algorithm 4

Input: T is a trace collected from the FLATTENEDTRACES procedure, which contains a list of query-result pairs ($\langle q, r \rangle$) collected from an execution of the application. In each query-result pair, the query (q) is an instance of the nonterminal symbol "AQuery" in the syntax for abstract traces that represents an SQL query that the application sent to the database. The result (r) is an integer representing the number of rows retrieved by the corresponding query.
Input: l is the list of loop information for T.
Input: i is a positive integer.
Output: The fragment of the application, expressed in the KONURE DSL, after updating P with the tail of trace T starting from the i-th query.

```
procedure MergeTraceNil($T, l, i$)
    $\langle\langle q_1^T, r_1^T\rangle, \ldots, \langle q_i^T, r_i^T\rangle, \langle q_{i+1}^T, r_{i+1}^T\rangle, \ldots, \langle q_k^T, r_k^T\rangle\rangle \leftarrow T$
    if $i = k$ then
        return "$q_i^T$"
    else                                                    ▷ $i < k$
        $b \leftarrow$ MergeTraceNil($T, l, i+1$)
        if $l$ says $T$ has a loop starting at $i$ then     ▷ Production "Block := For"
            return "for $q_i^T$ do $b$ else Nil"
        else                                                ▷ Production "Block := Query Block"
            return "$q_i^T$ $b$"
        end if
    end if
end procedure
```

Algorithm 5

Input: T is a trace collected from the FLATTENEDTRACES procedure, which contains a list of query-result pairs ($\langle q, r\rangle$) collected from an execution of the application. In each query-result pair, the query (q) is an instance of the nonterminal symbol "AQuery" in the syntax for abstract traces that represents an SQL query that the application sent to the database. The result (r) is an integer representing the number of rows retrieved by the corresponding query.
Input: l is the list of loop information for T.
Input: i is a positive integer.
Input: P is the fragment of the application, expressed in the KONURE DSL that corresponds to the tail of T starting from the i-th query. $\underline{P \text{ is of the form "} q^P \text{".}}$
Output: The fragment of the application, expressed in the KONURE DSL, after updating P with the tail of trace T starting from the i-th query.

```
procedure MERGETRACEEND($T, l, i, P$)
    $\langle\langle q_1^T, r_1^T\rangle, \ldots, \langle q_i^T, r_i^T\rangle, \langle q_{i+1}^T, r_{i+1}^T\rangle, \ldots, \langle q_k^T, r_k^T\rangle\rangle \leftarrow T$    ▷ SAMESKELETON($q_i^T, q^P$) is True
    $q \leftarrow$ MERGEQUERY($q_i^T, r_i^T, q^P$)
    if $i = k$ then    ▷ Production "Block := Query Block := Query $\epsilon$"
        return "$q$"
    else    ▷ $i < k$
        $b \leftarrow$ MERGETRACENIL($T, l, i + 1$)
        if $l$ says $T$ has a loop starting at $i$ then    ▷ Production "Block := For"
            return "for $q$ do $b$ else Nil"
        else    ▷ Production "Block := If"
            if $r_i^T > 0$ then
                return "if $q$ then $b$ else Nil"
            else
                return "if $q$ then Nil else $b$"
            end if
        end if
    end if
end procedure
```

Algorithm 6

Input: T is a trace collected from the FLATTENEDTRACES procedure, which contains a list of query-result pairs ($\langle q, r\rangle$) collected from an execution of the application. In each query-result pair, the query (q) is an instance of the nonterminal symbol "AQuery" in the syntax for abstract traces that represents an SQL query that the application sent to the database. The result (r) is an integer representing the number of rows retrieved by the corresponding query.
Input: l is the list of loop information for T.
Input: i is a positive integer.
Input: P is the fragment of the application, expressed in the KONURE DSL that corresponds to the tail of T starting from the i-th query. $\underline{P \text{ is of the form "} q_1^P\ q_2^P\ b^P \text{"}}$.
Output: The fragment of the application, $\overline{\text{expressed in the KONURE}}$ DSL, after updating P with the tail of trace T starting from the i-th query.

```
procedure MERGETRACESEQUENCE($T, l, i, P$)
    $\langle\langle q_1^T, r_1^T\rangle, \ldots, \langle q_i^T, r_i^T\rangle, \langle q_{i+1}^T, r_{i+1}^T\rangle, \ldots, \langle q_k^T, r_k^T\rangle\rangle \leftarrow T$    ▷ SAMESKELETON($q_i^T, q_1^P$) is True
    $q \leftarrow$ MERGEQUERY($q_i^T, r_i^T, q_1^P$)
    if $i = k$ then                                                ▷ Production "Block := If"
        if $r_i^T > 0$ then return "if $q$ then Nil else $q_2^P\ b^P$"
        else return "if $q$ then $q_2^P\ b^P$ else Nil"
        end if
    else                                                           ▷ $i < k$
        if $l$ says $T$ has a loop starting at $i$ then            ▷ Production "Block := For"
            if $\exists r.\ r \in$ MERGEDRESULTS($q_1^P$) $\wedge\ r > 0$ then
                $b_t \leftarrow$ MERGETRACE($T, l, i + 1$, "$q_2^P\ b^P$")
            else
                $b_t \leftarrow$ MERGETRACENIL($T, l, i + 1$)
            end if
            if $0 \in$ MERGEDRESULTS($q_1^P$) then $b_f \leftarrow$ "$q_2^P\ b^P$"
            else $b_f \leftarrow$ "Nil"
            end if
            return "for $q$ do $b_t$ else $b_f$"
        else
            if $r_i^T \in$ MERGEDRESULTS($q_1^P$) or SAMESKELETON($q_{i+1}^T, q_2^P$) then
                                                                   ▷ Production "Block := Query Block"
                $b \leftarrow$ MERGETRACE($T, l, i + 1$, "$q_2^P\ b^P$")
                return "$q\ b$"
            else                                                   ▷ Production "Block := If"
                $b \leftarrow$ MERGETRACENIL($T, l, i + 1$)
                if $r_i^T > 0$ then return "if $q$ then $b$ else $q_2^P\ b^P$"
                else return "if $q$ then $q_2^P\ b^P$ else $b$"
                end if
            end if
        end if
    end if
end procedure
```

Algorithm 7

Input: T is a trace collected from the FLATTENEDTRACES procedure, which contains a list of query-result pairs ($\langle q, r \rangle$) collected from an execution of the application. In each query-result pair, the query (q) is an instance of the nonterminal symbol "AQuery" in the syntax for abstract traces that represents an SQL query that the application sent to the database. The result (r) is an integer representing the number of rows retrieved by the corresponding query.

Input: l is the list of loop information for T.

Input: i is a positive integer.

Input: P is the fragment of the application, expressed in the KONURE DSL that corresponds to the tail of T starting from the i-th query. P is of the form "`if` q^P `then` b_t^P `else` b_f^P".

Output: The fragment of the application, expressed in the KONURE DSL, after updating P with the tail of trace T starting from the i-th query.

```
procedure MergeTraceIf($T, l, i, P$)
    $\langle\langle q_1^T, r_1^T\rangle, \ldots, \langle q_i^T, r_i^T\rangle, \langle q_{i+1}^T, r_{i+1}^T\rangle, \ldots, \langle q_k^T, r_k^T\rangle\rangle \leftarrow T$        ▷ SameSkeleton($q_i^T, q^P$) is True
    if $i = k$ then
        return $P$
    else                                                        ▷ $i < k$
        $q \leftarrow$ MergeQuery($q_i^T, r_i^T, q^P$)
        if $l$ says $T$ has a loop starting at $i$ then         ▷ Production "Block := For"
            $b \leftarrow$ MergeTrace($T, l, i + 1, b_t^P$)
            return "for $q$ do $b$ else $b_f^P$"
        else                                                    ▷ Production "Block := If"
            if $r_i^T > 0$ then
                $b \leftarrow$ MergeTrace($T, l, i + 1, b_t^P$)
                return "if $q$ then $b$ else $b_f^P$"
            else
                $b \leftarrow$ MergeTrace($T, l, i + 1, b_f^P$)
                return "if $q$ then $b_t^P$ else $b$"
            end if
        end if
    end if
end procedure
```

Algorithm 8

Input: T is a trace collected from the FLATTENEDTRACES procedure, which contains a list of query-result pairs ($\langle q, r\rangle$) collected from an execution of the application. In each query-result pair, the query (q) is an instance of the nonterminal symbol "AQuery" in the syntax for abstract traces that represents an SQL query that the application sent to the database. The result (r) is an integer representing the number of rows retrieved by the corresponding query.
Input: l is the list of loop information for T.
Input: i is a positive integer.
Input: P is the fragment of the application, expressed in the KONURE DSL that corresponds to the tail of T starting from the i-th query. P is of the form "`for` q^P `do` b_t^P `else` b_f^P".
Output: The fragment of the application, expressed in the KONURE DSL, after updating P with the tail of trace T starting from the i-th query.

```
1:  procedure MERGETRACEFOR($T, l, i, P$)
2:      $\langle\langle q_1^T, r_1^T\rangle, \ldots, \langle q_i^T, r_i^T\rangle, \langle q_{i+1}^T, r_{i+1}^T\rangle, \ldots, \langle q_k^T, r_k^T\rangle\rangle \leftarrow T$    ▷ SAMESKELETON($q_i^T, q^P$) is True
3:      if $i = k$ then
4:          return $P$
5:      else                                                        ▷ $i < k$
                                                                    ▷ Production "Block := For"
6:          $q \leftarrow$ MERGEQUERY($q_i^T, r_i^T, q^P$)
7:          if $r_i^T > 0$ then
8:              $b \leftarrow$ MERGETRACE($T, l, i + 1, b_t^P$)
9:              return "for $q$ do $b$ else $b_f^P$"
10:         else
11:             $b \leftarrow$ MERGETRACE($T, l, i + 1, b_f^P$)
12:             return "for $q$ do $b_t^P$ else $b$"
13:         end if
14:     end if
15: end procedure
```

3 EXPERIMENTAL RESULTS

We present an experiment that compares the human user-provided inputs and the inputs chosen by the Konure inference algorithm.

3.1 Methodology

We used Etch to construct programs for the applications in the Konure technical report [Shen and Rinard 2018].[1] For each application command, we ran Etch twice, each time using a different set of value assignments: The first time uses the value assignments collected by the Etch proxy when the human user interacted with the application. The second time uses the value assignments chosen by the Konure inference algorithm.

The outcomes of running Etch are executable Python programs, each program consistent with the observed functionality of the application. We measured the quality of these Etch-constructed programs based on the following metrics.

Output comparison metrics: For each command of each application, we compared the outputs from the following programs: (1) the Etch-constructed program using value assignments from the human user, (2) the Etch-constructed program using value assignments from Konure, (3) the Konure-inferred program, and (4) the original application.

We executed all these programs using the following sets of value assignments: (1) the value assignments collected from human user interaction and (2) the value assignments chosen by Konure during inference. For each set of value assignments, we executed all programs and compared their output differences. [2]

We then counted the number of value assignments for which the outputs differed. The outputs can differ in the following ways.

- The Etch-constructed program may *miss* output values because it misses a branch of a conditional statement. When a value assignment causes the program to enter a branch that Etch did not observe, the original application produces certain outputs but the Etch-constructed program performs a no-op for that branch.
- The Etch-constructed program may *miss* output values even when the program contains the query for retrieving these values: Etch may have not observed an execution where this query returns nonempty data. Hence, Etch have not observed any data from this query appear in the outputs. The Etch-constructed program, therefore, lacks the corresponding output statements.
- The Etch-constructed program can produce *extra* output values because it incorrectly references certain data fields. This error can happen when the observed traces are ambiguous, that is, when the trace (or the output) contains values that have multiple potential source locations that are not equivalent. The Etch-constructed program may reference any one of these source locations that produces correct outputs for all the observed value assignments. However, this source location is not guaranteed to be correct. An incorrect reference may be exposed when the Etch-constructed program executes with an unobserved value assignment. In this case, the program may use the incorrect data to produce extra outputs or to perform incorrect queries that affect the execution further.

[1] We exclude the Kandan commands get_channels_id_activities_id, get_me, and get_users_id because these commands did not occur in the user interaction session. We exclude the Kandan command get_channels_id_attachments because the user triggered a functionality in this command that could not be expressed in the Konure DSL.

[2]Technically, the outputs from the original application consist of templates of text (in HTML, JSON, or other formats) filled in with result-specific values. We preprocessed these outputs by extracting output values from the text.

A key insight for using the Konure value assignments is that the they are comprehensive enough for Konure to identify a unique equivalent class of programs in the Konure DSL, by observing the application outputs and the database traffic. Hence, if the Etch-constructed program is inaccurate in any way that affects the program outputs, the inaccuracy will manifest as incorrect outputs or incorrect database queries [3] when the Etch-constructed program processes the Konure value assignments.

Code comparison metrics: For each command of each application, we statically compared the Etch-constructed programs (two programs, generated with human value assignments and with Konure-chosen value assignments, respectively) and the Konure-inferred program side by side. Note that whenever two Konure DSL programs are equivalent in terms of the observable database queries and outputs, they must be syntactically identical except for potentially equivalent data fields referenced in queries and outputs. Recall that both Etch and Konure generate Python programs directly from the Konure DSL. Thus, we can compare the syntactical difference between these two programs to quantify how their functionality differ.

Specifically, we analyzed the Etch-constructed program code and counted the missing or different components compared to the Konure-inferred program:

- The number of missing conditional branches.
- The number of missing loops.
- The number of missing or different statements that perform SQL queries. Each such query is originally extracted from the database traffic between the application and the backend database.[4] Recall that Etch uses the observed queries to construct an abstract trace. This process, as documented in the Konure technical report [Shen and Rinard 2018], transforms each SQL query into a query skeleton (the query excluding concrete values) along with references to source locations (the potential origins of these concrete values). Etch eventually constructs a Python program with statements that use these SQL query skeletons and references variables that hold earlier data.
- The number of missing or different output statements. Each such statement outputs the data from a source location.

Recall that the difference between the Etch-constructed program and the Konure-inferred program is caused by the use of value assignments for observing the application. Thus, these comparisons allow us to visualize how the human user and the Konure inference algorithm explore the hidden components of the application.

3.2 Results

For each command of each application, we present two figures. Here, the first figure presents the quality metrics for the Etch-constructed program using value assignments provided by the human user and captured by the Etch proxy. The second figure presents the quality metrics for the Etch-constructed program using value assignments chosen by the Konure inference algorithm. Both figures compare the Etch-constructed programs to the program that Konure inferred for the application command.

Each such figure contains six sub-figures. In all these sub-figures, the horizontal axes represent the number of value assignments processed by Etch. For example, a data point at horizontal mark *n* corresponds to the DSL program constructed by Etch based on only the first *n* user-provided value assignments. The user-provided value assignments are ordered by the time that they were captured by the Etch proxy.

[3]Note that observing only the outputs is insufficient for exposing all inaccuracies.

[4]Like Konure, Etch also intercepts this traffic.

Output comparison results: Among each group of six sub-figures, the first two sub-figures (a–b) present the output comparison results.

Sub-figure (a) presents the results for the output comparison by executing the Etch-constructed program with all human user-provided value assignments and comparing the outputs against the original application. The vertical axes represent the number of executed value assignments for which the Etch-constructed program produces incorrect outputs. The legends are:

- Line "user_input_incorrect_total" represents the total number of human-provided value assignments for which the Etch-constructed program produces incorrect outputs.
- Line "user_input_miss_out" represents the number of human-provided value assignments for which the Etch-constructed program produces incorrect outputs that miss values.
- Line "user_input_more_out" represents the number of human-provided value assignments for which the Etch-constructed program produces incorrect outputs that have extra values.

Sub-figure (b) presents the results for the output comparison by executing the Etch-constructed program with all Konure-chosen value assignments and comparing the outputs against the original application. The vertical axes represent the number of executed value assignments for which the Etch-constructed program produces incorrect outputs. The legends are:

- Line "active_input_incorrect_total" represents the total number of Konure-chosen value assignments for which the Etch-constructed program produces incorrect outputs.
- Line "active_input_miss_out" represents the number of Konure-chosen value assignments for which the Etch-constructed program produces incorrect outputs that miss values.
- Line "active_input_more_out" represents the number of Konure-chosen value assignments for which the Etch-constructed program produces incorrect outputs that have extra values.

We note that the Konure-inferred program produced correct outputs for all of these value assignments.

Code comparison results: The next four sub-figures (c–f) present the code comparison results.

Sub-figure (c) presents the results for the code comparison between the Etch-constructed program against the Konure-inferred program, focusing on conditional branches. The vertical axis represents the number of branches that the Etch-constructed program differs from the Konure-inferred program. The legends are:

- Line "miss_branch_total" represents the total number of conditional branches in the Konure-inferred program that are missing in the Etch-constructed program.
- Line "miss_branch_empty" represents the number of branches in the Konure-inferred program that condition on a query retrieving empty data that are missing in the Etch-constructed program.
- Line "miss_branch_exist" represents the number of branches in the Konure-inferred program that condition on a query retrieving nonempty data that are missing in the Etch-constructed program.

Sub-figure (d) presents the results for the code comparison between the Etch-constructed program against the Konure-inferred program, focusing on loops. The vertical axis represents the number of loops that the Etch-constructed program differs from the Konure-inferred program. The legends are:

- Line "miss_loop_total" represents the total number of loops in the Konure-inferred program that are missing in the Etch-constructed program.
- Line "miss_loop_with_no_query" represents the number of loops in the Konure-inferred program that are missing in the Etch-constructed program and the Etch-constructed program does not have any query that corresponds to the loop body.

- Line "miss_loop_has_query" represents the number of loops in the Konure-inferred program that are missing in the Etch-constructed program and the Etch-constructed program has queries, but not any control structure, that correspond to the loop body.
- Line "miss_loop_has_if" represents the number of loops in the Konure-inferred program that are missing in the Etch-constructed program and the Etch-constructed program has a conditional branch in the place of the loop.

Sub-figure (e) presents the results for the code comparison between the Etch-constructed program against the Konure-inferred program, focusing on statements that perform SQL queries. The vertical axis represents the number of SQL query statements that the Etch-constructed program differs from the Konure-inferred program. The legends are:

- Line "miss_query" represents the number of SQL query statements in the Konure-inferred program that are missing in the Etch-constructed program.
- Line "diff_query_use_source" represents the number of SQL query statements in the Konure-inferred program that are present in the Etch-constructed program but these queries reference different source locations.[5]
- Line "diff_query_use_list" represents the number of SQL query statements in the Konure-inferred program that are present in the Etch-constructed program, referencing the same source locations, but these references differ in whether they use only a single value or the entire list of the previously-retrieved data.

Sub-figure (f) presents the results for the code comparison between the Etch-constructed program against the Konure-inferred program, focusing on statements that produce outputs. The vertical axis represents the number of output statements that the Etch-constructed program differs from the Konure-inferred program. The legends are:

- Line "miss_out_total" represents the total number of output statements in the Konure-inferred program that are missing[6] in the Etch-constructed program.
- Line "miss_out_with_no_query" represents the number of output statements in the Konure-inferred program that are missing in the Etch-constructed program and the Etch-constructed program does not have the corresponding queries that retrieve the output values.
- Line "miss_out_has_query_miss_exist" represents the number of output statements in the Konure-inferred program that are missing in the Etch-constructed program even though the Etch-constructed program has the corresponding queries that retrieve the output values.
- Line "diff_out_stmt" represents the number of output statements in the Etch-constructed program but not in the Konure-inferred program.

We next present results for each command of each application.

Figure 1 presents the measured quality of the Etch-constructed program for the Kandan [kan 2018] `get_channels` command, using the value assignments provided by the human Etch user. The quality is measured in comparison to the Konure-inferred program. Figure 2 presents the measured quality of the program that Etch constructed using the value assignments provided by Konure.

Figure 3 presents the measured quality of the Etch-constructed program for the Kandan `get_channels_id_activities` command, using human-provided value assignments. Figure 4 presents the measured quality of the program that Etch constructed using the value assignments provided by Konure.

[5]Two different source locations may or may not be equivalent, depending on whether they always hold the same data.
[6]A missed output statement may or may not affect the final output, due to potentially duplicated output statements.

Figure 5 presents the measured quality of the Etch-constructed program for the Kandan `get_users` command, using human-provided value assignments. Figure 6 presents the measured quality of the program that Etch constructed using the value assignments provided by Konure.

Figure 7 presents the measured quality of the Etch-constructed program for the student registration system's `liststudentcourses` command, using human-provided value assignments. Figure 8 presents the measured quality of the program that Etch constructed using the value assignments provided by Konure.

Figure 9 presents the measured quality of the Etch-constructed program for the Blog application [rai 2018] `get_article_id` command, using human-provided value assignments. Figure 10 presents the measured quality of the program that Etch constructed using the value assignments provided by Konure.

Figure 11 presents the measured quality of the Etch-constructed program for the Blog application `get_articles` command, using human-provided value assignments. Figure 12 presents the measured quality of the program that Etch constructed using the value assignments provided by Konure.

3.3 Discussion

In the output comparison sub-figures (a–b), each positive count shows that the Etch-constructed program produced incorrect outputs for certain value assignments. [7] The programs constructed with human user value assignments produced correct outputs for the observed functionality (Figures 1a, 3a, 5a, 7a, 9a, and 11a). However, these programs are not completely correct when tested with the Konure-chosen value assignments (Figures 1b, 5b, and 9b). In contrast (as expected), the programs constructed with Konure value assignments produced correct outputs for all the Konure-chosen value assignments (Figures 2a, 2b, 4a, 4b, 6a, 6b, 8a, 8b, 10a, 10b, 12a, and 12b). This is also the case for the Konure-inferred programs.

In the code comparison sub-figures (c–f), each positive count represents that the Etch-constructed program differs from the Konure-inferred program. Most of the programs constructed with human user value assignments did not converge to the correct programs (Figures 1, 3, 5, and 9), except when the programs are relatively simple (Figures 7 and 11). Specifically, these programs missed conditional branches (Figures 1c, 3c, 5c, and 9c), and loops (Figures 1d). As a result, these programs also contained inaccurate queries (Figures 1e, 3e, and 5e) and inaccurate output statements (Figures 1f, 5f, and 9f). Most of these differences cause the program outputs to differ, except for the following cases:

- When the Etch-constructed program misses a conditional branch, the program may still produce the correct outputs if the rest of the execution does not produce outputs regardless of the branch taken. Specifically, the corresponding Konure-inferred branch may not contain any query or may not produce any outputs. Meanwhile, while the Etch-constructed enters the incorrect branch, it may happen to not produce any outputs (Figure 3c). When a missing branch does not produce outputs regardless of the branch taken, the missing branch does not change the program outputs.[8]
- When the Etch-constructed program misses an SQL query statement, the corresponding Konure-inferred query's results may not be used to produce any outputs or to provide intermediate results for any query affecting outputs (Figure 3e). Even when the corresponding

[7]An incorrect Etch-constructed program is guaranteed to produce incorrect database *traffic* for at least one Konure-chosen value assignment. However, an incorrect Etch-constructed program may not produce incorrect *outputs* for any Konure-chosen value assignment.

[8]Note that the missing branch still affects the database traffic.

KONURE-inferred query produces outputs, these outputs may happen to be a subset of other existing output values (Figure 7d). When a missing query does not affect the program outputs, the missing query does not change the program outputs.[9]

- When the ETCH-constructed program references an ambiguous source location, the referenced source location may be equivalent to the corresponding KONURE-inferred source location (Figures 7e and 8e). In this case, the equivalent source locations are interchangeable and do not change the program behavior.
- When the ETCH-constructed program misses an output statement, the corresponding KONURE-inferred outputs may happen to be a subset of other existing output values (Figure 6f). In this case, the missing output statement does not change the set of output values produced by the program.

In contrast (as expected), the programs constructed with KONURE value assignments all converged to the correct program equivalent to the KONURE-inferred program. These results highlight the effectiveness of the inputs chosen by the KONURE inference algorithm.

Taking a closer look at the lines in these plots, most of the lines are non-increasing when ETCH processes more value assignments. This general trend implies that, as ETCH observes more executions of the application, it is able to construct more accurate programs that capture the application functionality. There are, however, a few situations where the lines can increase.

- In sub-figures (a) and (b), the lines "user_input_miss_out" and "active_input_miss_out" can increase (Figure 3a) when ETCH fluctuates between multiple ambiguous source locations that are not equivalent for certain unseen value assignments.
- In sub-figures (a) and (b), the lines "user_input_more_out" and "active_input_more_out" can increase (Figures 3a, 3b, 4a, and 4b) when ETCH fluctuates between multiple ambiguous source locations that are not equivalent or when ETCH reaches a new branch but does not have enough information to eliminate ambiguity.
- In sub-figures (a) and (b), the lines "user_input_incorrect_total" and "active_input_incorrect_total" can increase because of the two points above.
- In sub-figure (d), the lines "miss_loop_has_query" and "miss_loop_has_if" can increase (Figures 1d, 2d, and 7d) when ETCH reaches a new branch in the KONURE-inferred program that contains a loop but ETCH has not yet seen multiple iterations of this loop, hence ETCH does not have enough information to construct the loop structure.
- In sub-figure (e), the lines "diff_query_use_source" and "diff_query_use_list" can increase (Figures 1e, 2e, 3e, 4e, 5e, 6e, 7e, and 8e) when ETCH fluctuates between multiple ambiguous source locations or when ETCH reaches a new branch but does not have enough information to eliminate ambiguity.
- In sub-figure (f), the line "miss_out_has_query_miss_exist" can increase (Figures 2f, 4f, 6f, 7f, and 8f) when ETCH reaches a new branch in the KONURE-inferred program that contains a query that produces outputs but ETCH has not yet seen this query return any data, hence ETCH has not yet observed any output produced by this query.
- In sub-figure (f), the line "diff_out_stmt" can increase (Figure 1f, 3f, 4f, 5f, and 9f) when ETCH reaches a new branch but does not have enough information to eliminate ambiguity.

4 CONCLUSION

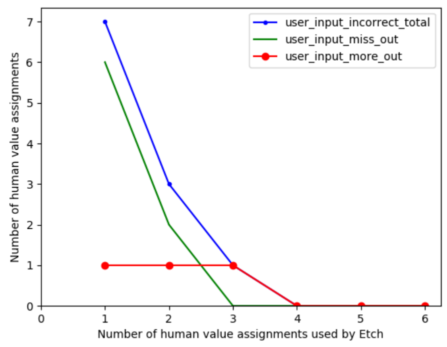
We present ETCH, a system that observes the inputs, outputs, and database traffic from a running system in normal use and then synthesizes a model of the application from this information. Preliminary results indicate that the human user-provided inputs may be insufficient for exploring

[9]Note that the missing query still affects the database traffic.

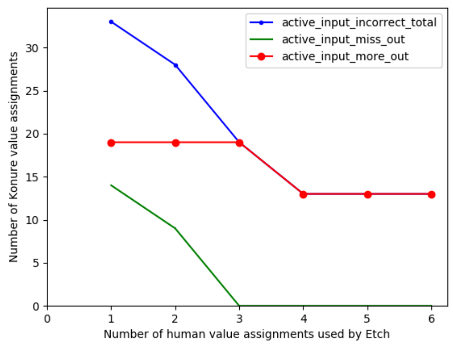
the functionality in rare usage scenarios, especially for more sophisticated applications. When the observations are insufficient, Etch may synthesize incorrect programs. Because Konure uses active learning to systematically build a model of the program, if the semantics of the program conforms to the Konure DSL, then Konure will correctly infer a program that correctly captures this unique semantics.

REFERENCES

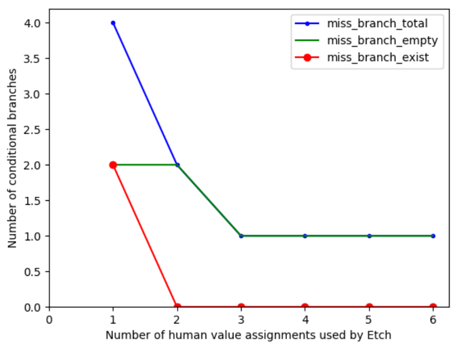
2018. Getting Started with Rails. http://guides.rubyonrails.org/getting_started.html.
2018. Kandan – Modern Open Source Chat. https://github.com/kandanapp/kandan.
Jiasi Shen and Martin Rinard. 2018. Using Active Learning to Synthesize Models of Applications That Access Databases. http://hdl.handle.net/1721.1/117593. Technical report.



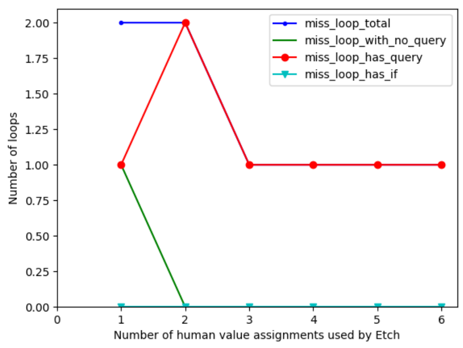
(a) Output comparison using value assignments provided by the human ETCH user



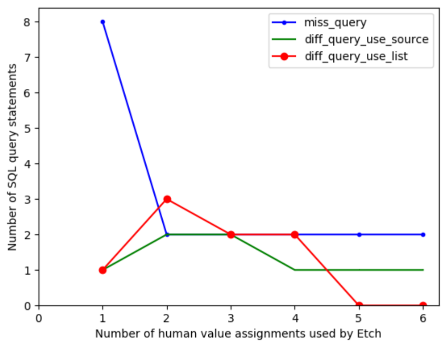
(b) Output comparison using value assignments chosen by KONURE



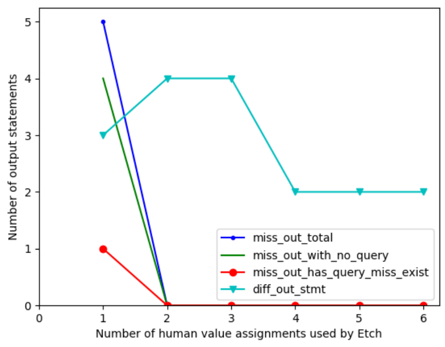
(c) Code comparison on conditional branches



(d) Code comparison on loops

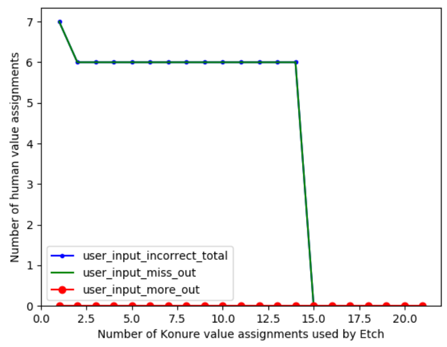


(e) Code comparison on statements that perform SQL queries

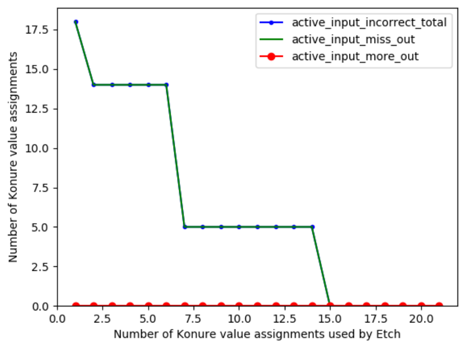


(f) Code comparison on statements that produce outputs

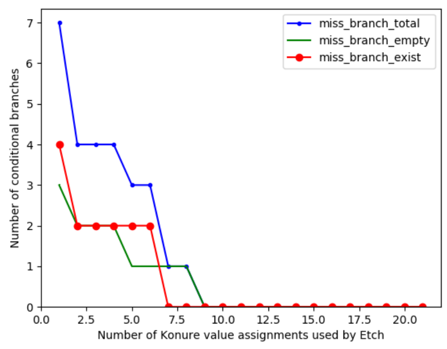
Fig. 1. ETCH uses human inputs to construct kandan command `get_channels`



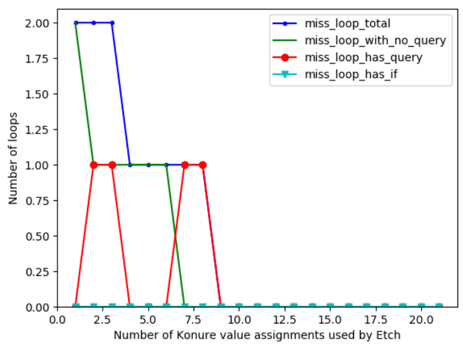
(a) Output comparison using value assignments provided by the human Etch user



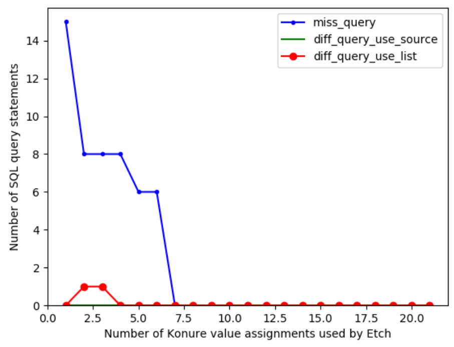
(b) Output comparison using value assignments chosen by Konure



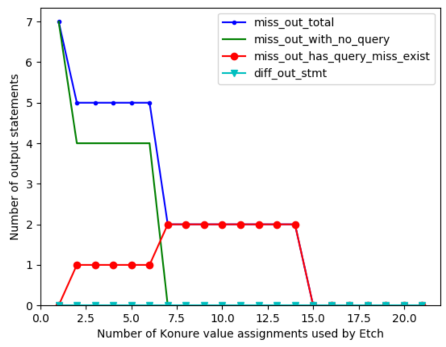
(c) Code comparison on conditional branches



(d) Code comparison on loops

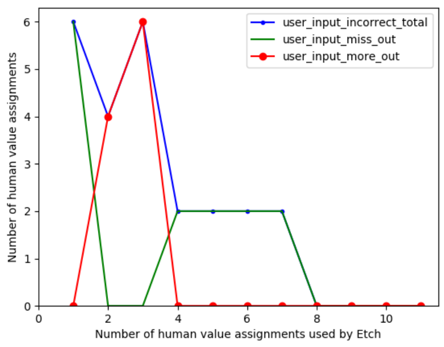


(e) Code comparison on statements that perform SQL queries

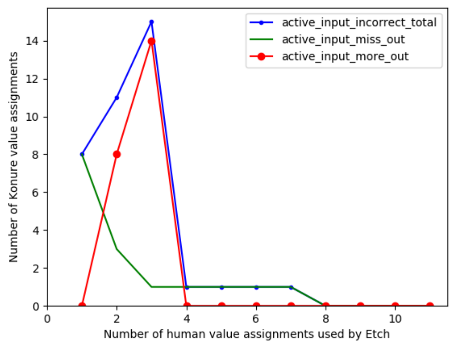


(f) Code comparison on statements that produce outputs

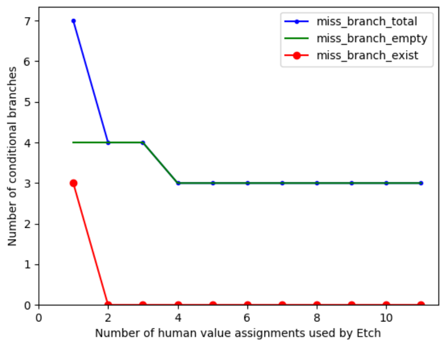
Fig. 2. Etch uses Konure inputs to construct kandan command `get_channels`



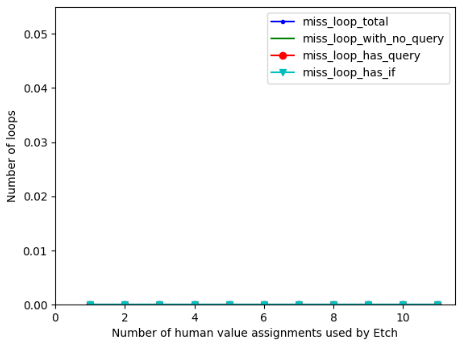
(a) Output comparison using value assignments provided by the human Etch user



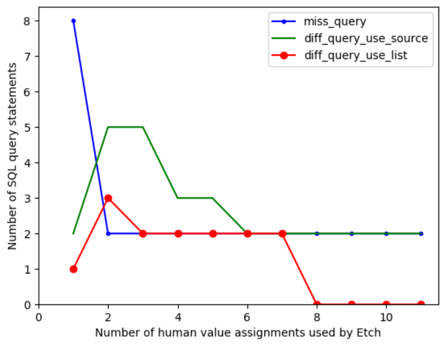
(b) Output comparison using value assignments chosen by Konure



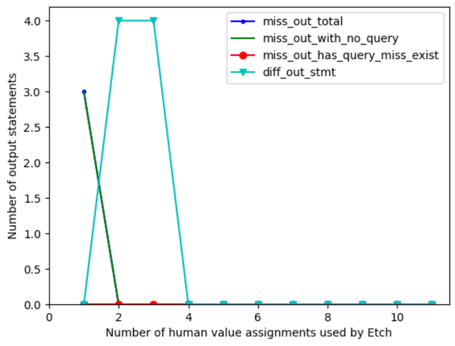
(c) Code comparison on conditional branches



(d) Code comparison on loops

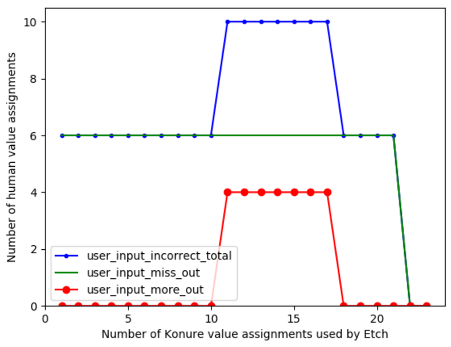


(e) Code comparison on statements that perform SQL queries

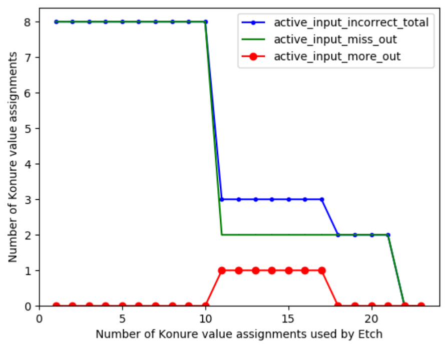


(f) Code comparison on statements that produce outputs

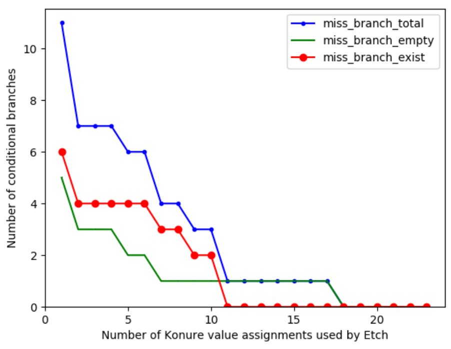
Fig. 3. Etch uses human inputs to construct kandan command `get_channels_id_activities`



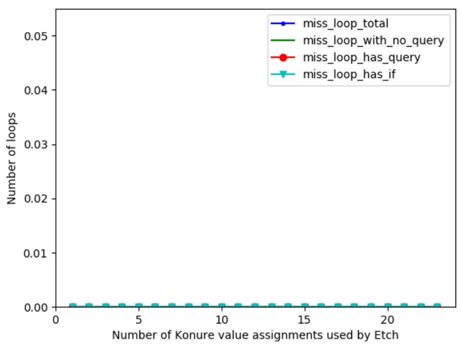
(a) Output comparison using value assignments provided by the human Etch user



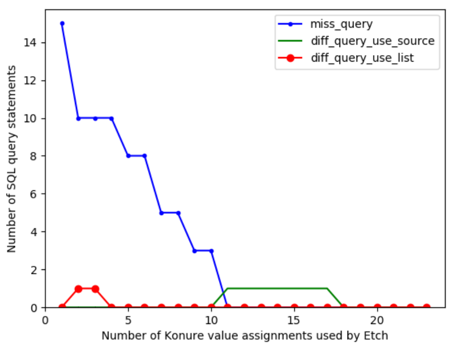
(b) Output comparison using value assignments chosen by Konure



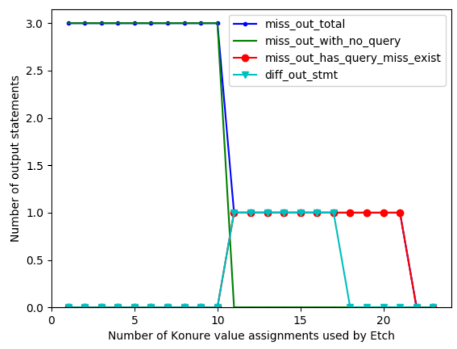
(c) Code comparison on conditional branches



(d) Code comparison on loops

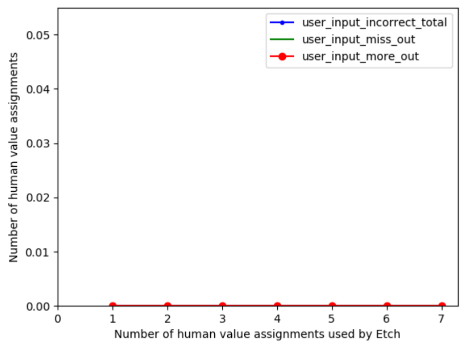


(e) Code comparison on statements that perform SQL queries

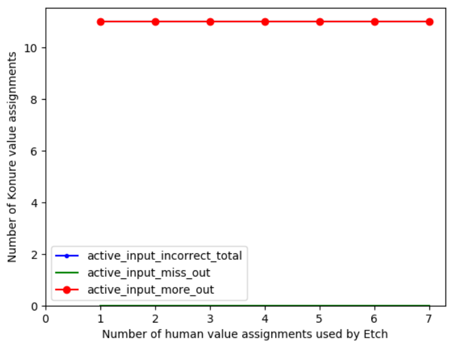


(f) Code comparison on statements that produce outputs

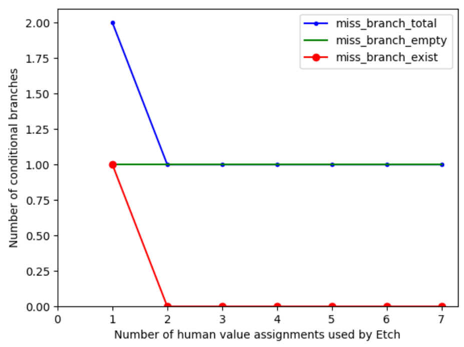
Fig. 4. Etch uses Konure inputs to construct kandan command `get_channels_id_activities`



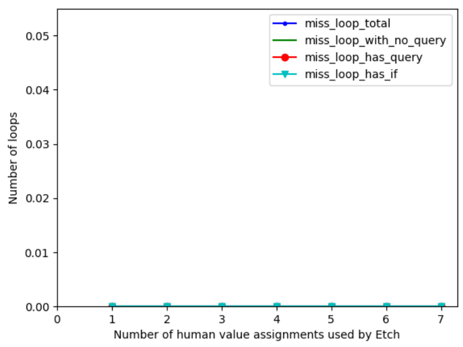
(a) Output comparison using value assignments provided by the human Etch user



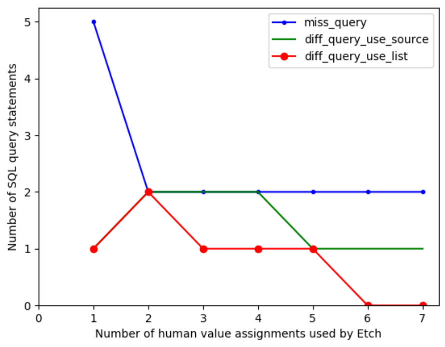
(b) Output comparison using value assignments chosen by Konure



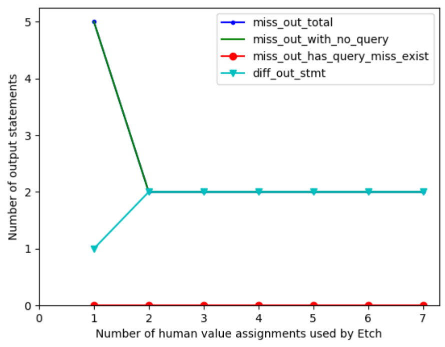
(c) Code comparison on conditional branches



(d) Code comparison on loops

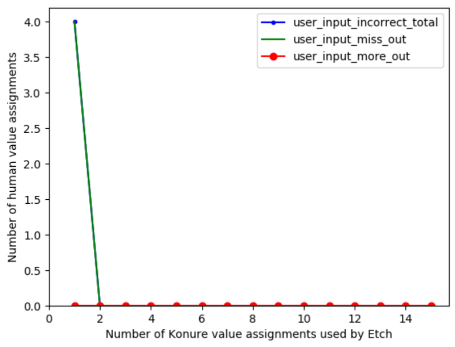


(e) Code comparison on statements that perform SQL queries

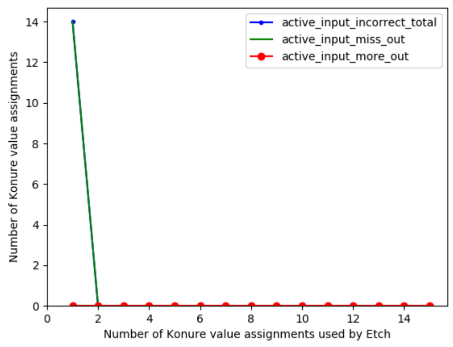


(f) Code comparison on statements that produce outputs

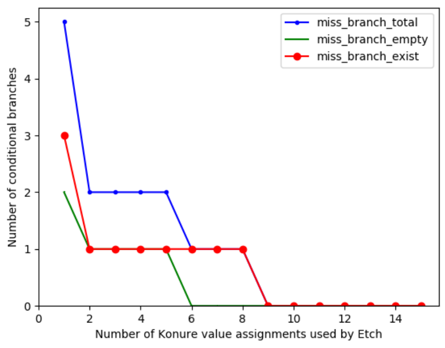
Fig. 5. Etch uses human inputs to construct kandan command `get_users`



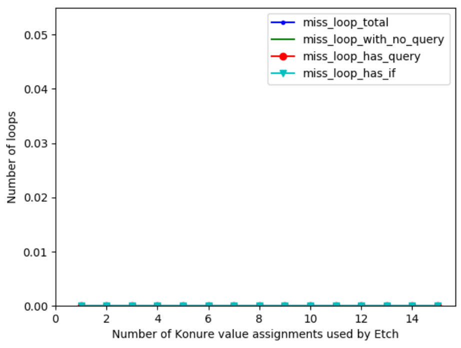
(a) Output comparison using value assignments provided by the human Etch user



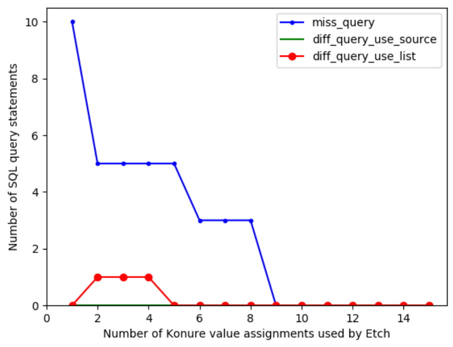
(b) Output comparison using value assignments chosen by Konure



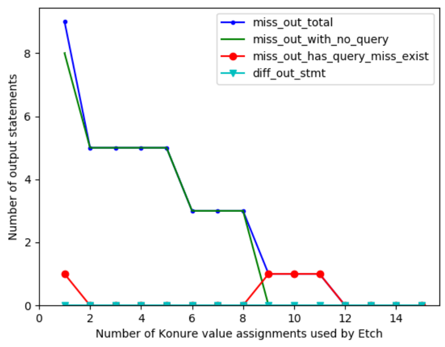
(c) Code comparison on conditional branches



(d) Code comparison on loops

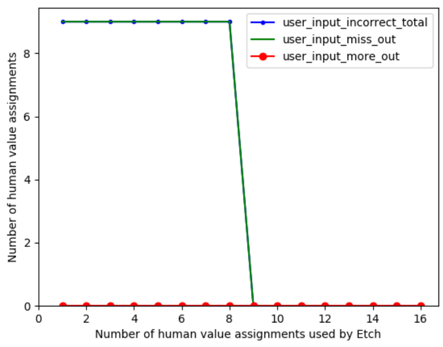


(e) Code comparison on statements that perform SQL queries

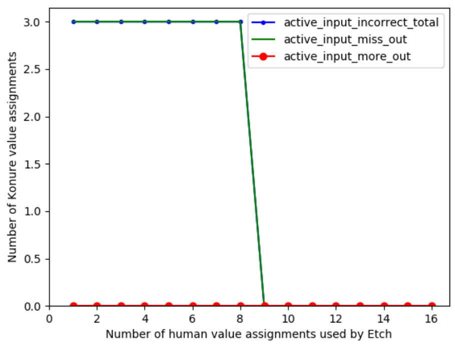


(f) Code comparison on statements that produce outputs

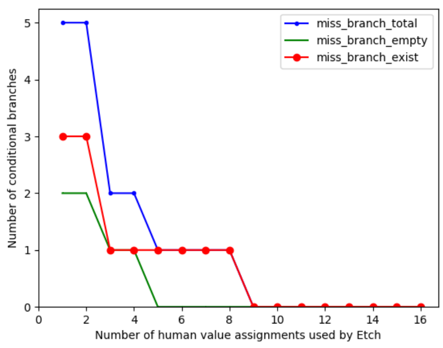
Fig. 6. Etch uses Konure inputs to construct kandan command get_users



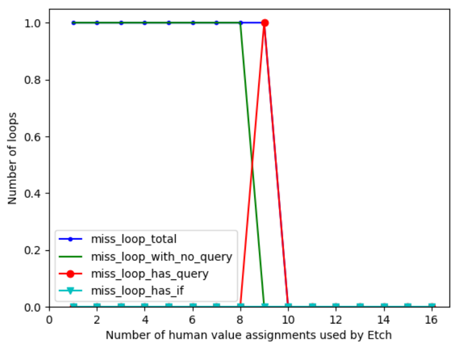
(a) Output comparison using value assignments provided by the human Etch user



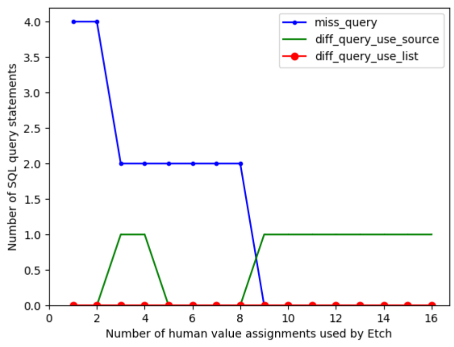
(b) Output comparison using value assignments chosen by Konure



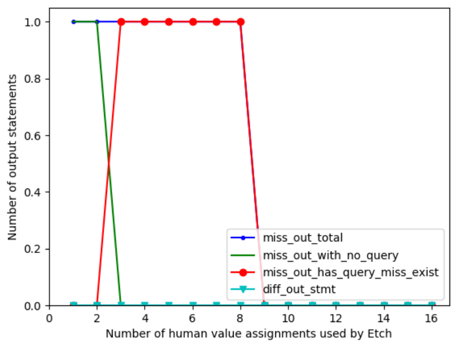
(c) Code comparison on conditional branches



(d) Code comparison on loops

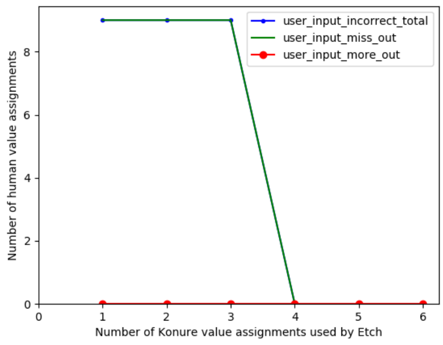


(e) Code comparison on statements that perform SQL queries

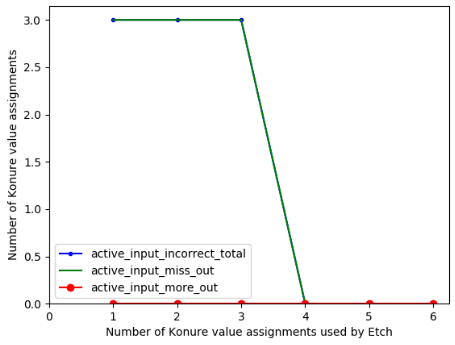


(f) Code comparison on statements that produce outputs

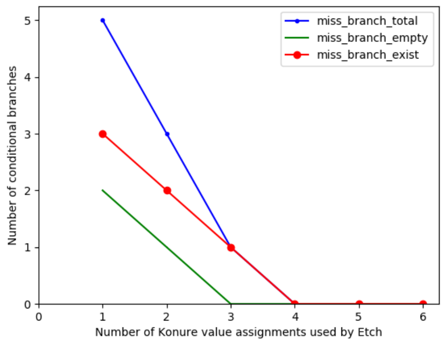
Fig. 7. Etch uses human inputs to construct studentdb command `liststudentcourses`



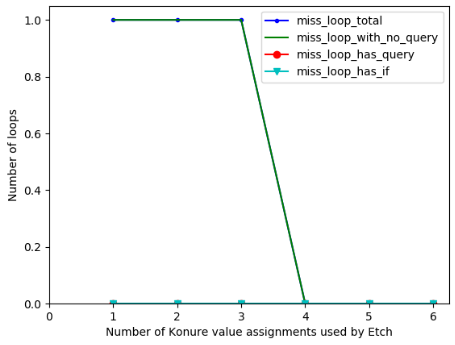
(a) Output comparison using value assignments provided by the human Etch user



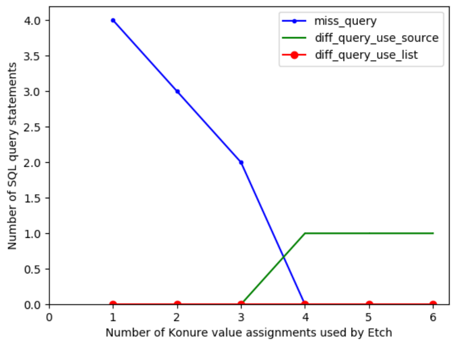
(b) Output comparison using value assignments chosen by Konure



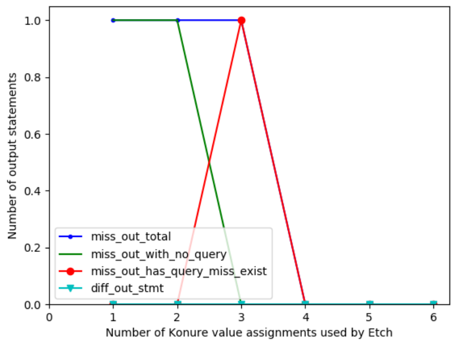
(c) Code comparison on conditional branches



(d) Code comparison on loops

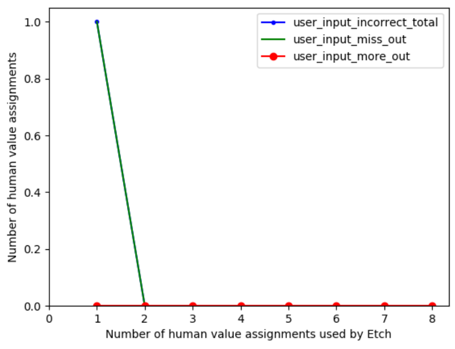


(e) Code comparison on statements that perform SQL queries

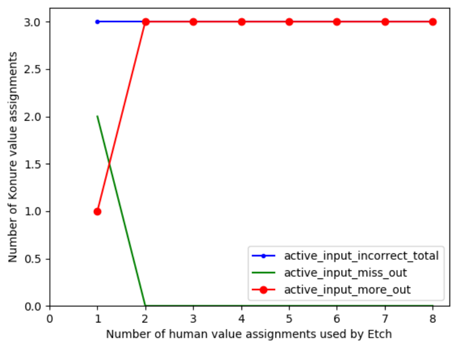


(f) Code comparison on statements that produce outputs

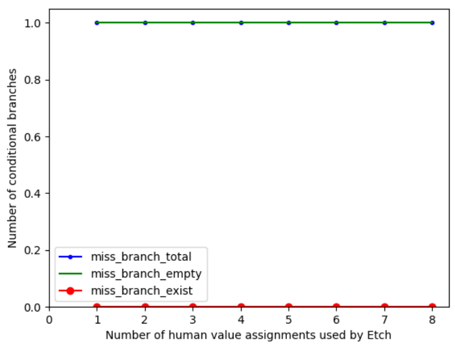
Fig. 8. Etch uses Konure inputs to construct studentdb command `liststudentcourses`



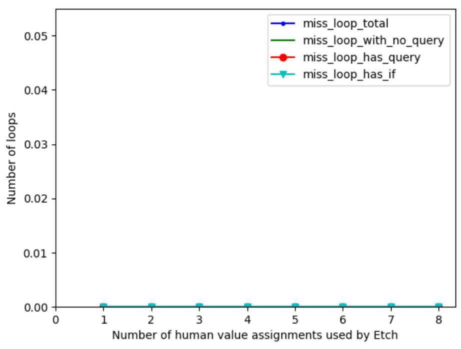
(a) Output comparison using value assignments provided by the human Etch user



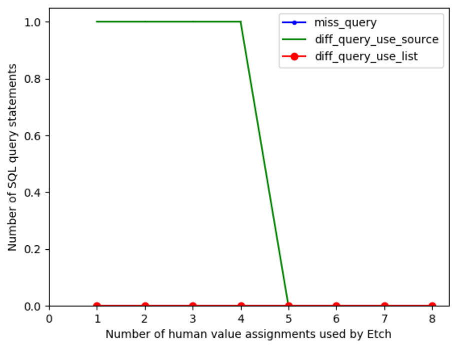
(b) Output comparison using value assignments chosen by Konure



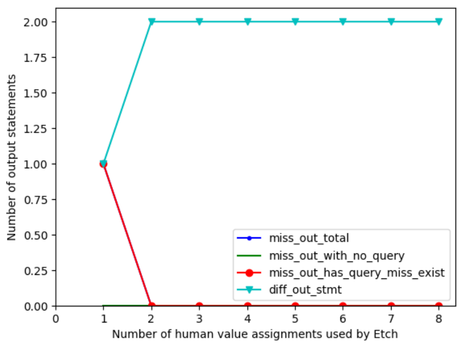
(c) Code comparison on conditional branches



(d) Code comparison on loops

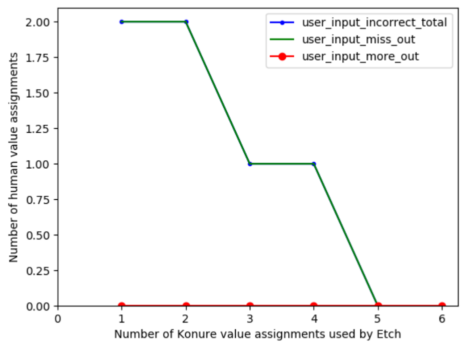


(e) Code comparison on statements that perform SQL queries

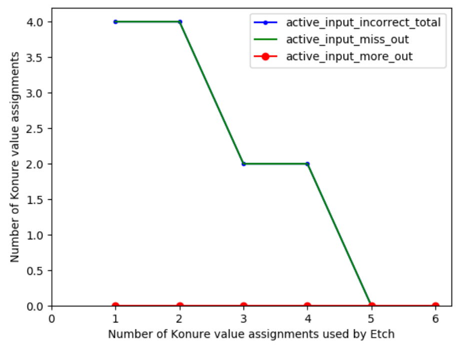


(f) Code comparison on statements that produce outputs

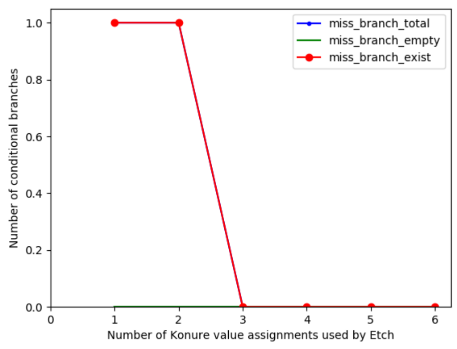
Fig. 9. Etch uses human inputs to construct blog command `get_article_id`



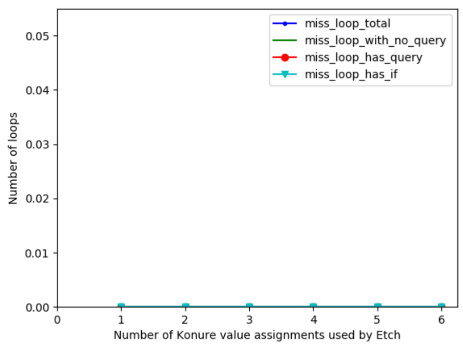
(a) Output comparison using value assignments provided by the human Etch user



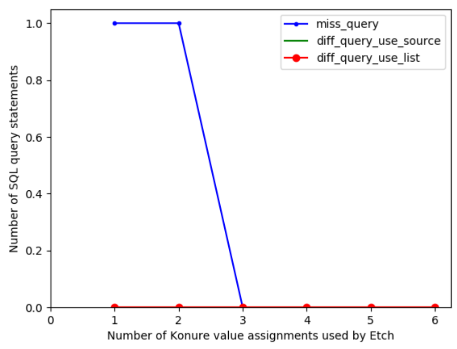
(b) Output comparison using value assignments chosen by Konure



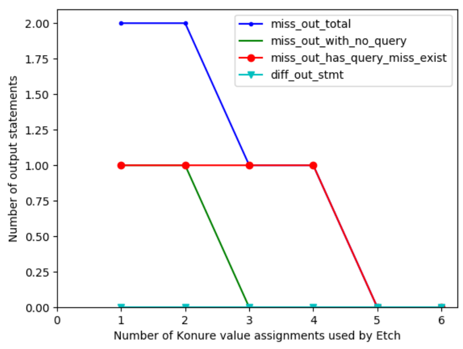
(c) Code comparison on conditional branches



(d) Code comparison on loops

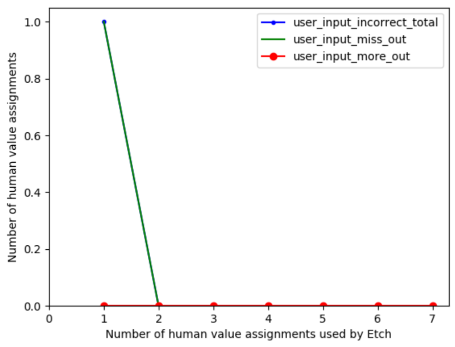


(e) Code comparison on statements that perform SQL queries

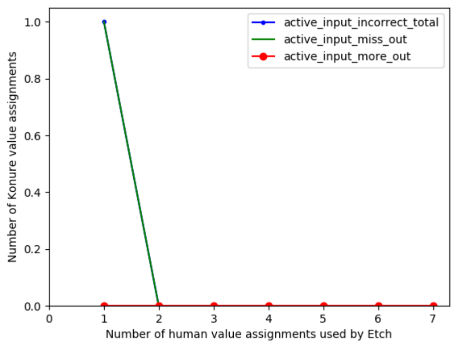


(f) Code comparison on statements that produce outputs

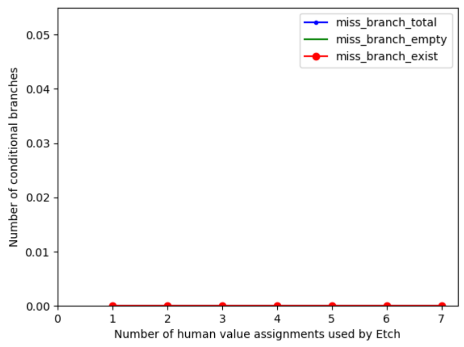
Fig. 10. Etch uses Konure inputs to construct blog command `get_article_id`



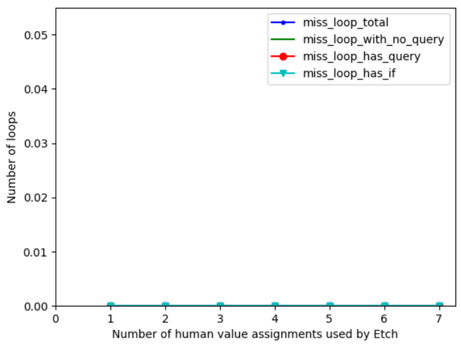
(a) Output comparison using value assignments provided by the human Etch user



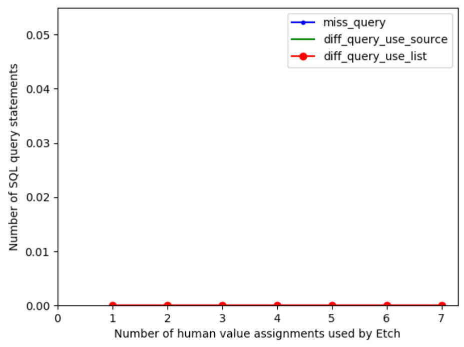
(b) Output comparison using value assignments chosen by Konure



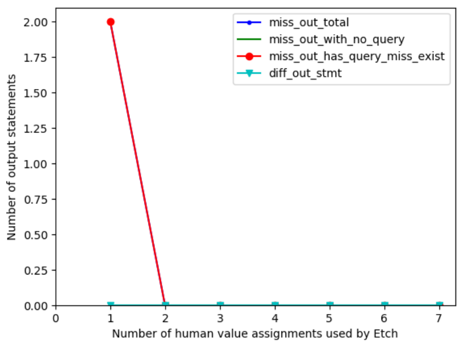
(c) Code comparison on conditional branches



(d) Code comparison on loops

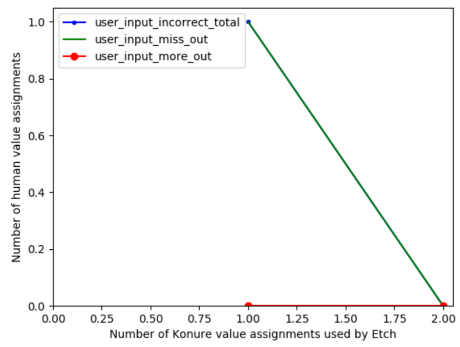


(e) Code comparison on statements that perform SQL queries

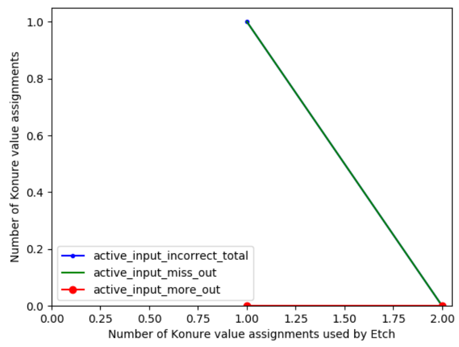


(f) Code comparison on statements that produce outputs

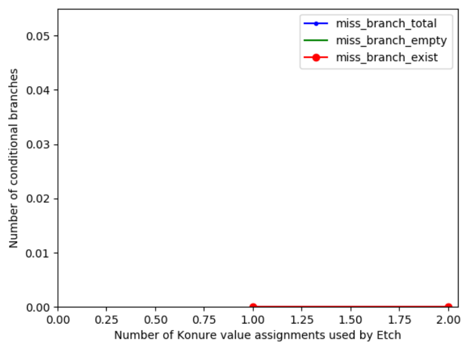
Fig. 11. Etch uses human inputs to construct blog command `get_articles`



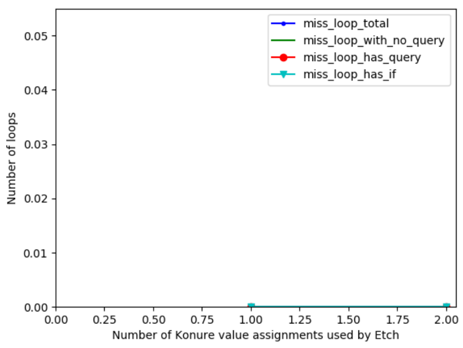
(a) Output comparison using value assignments provided by the human Etch user



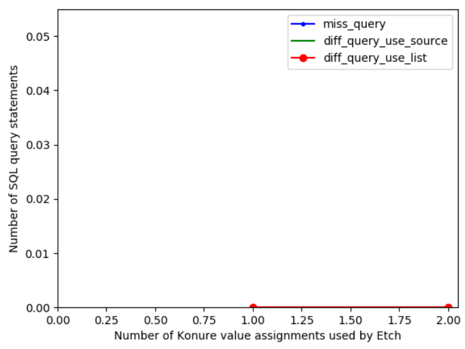
(b) Output comparison using value assignments chosen by Konure



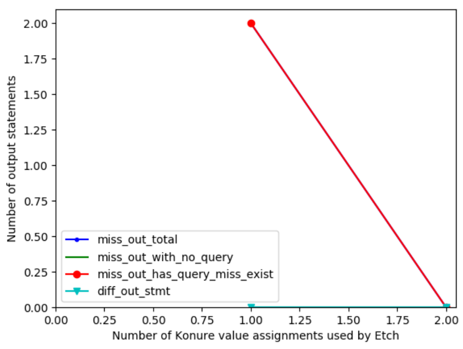
(c) Code comparison on conditional branches



(d) Code comparison on loops



(e) Code comparison on statements that perform SQL queries



(f) Code comparison on statements that produce outputs

Fig. 12. Etch uses Konure inputs to construct blog command `get_articles`