Using Active Learning to Synthesize Models of Applications That Access Databases

Jiasi Shen
MIT EECS & CSAIL
USA
jiasi@csail.mit.edu

Martin C. Rinard
MIT EECS & CSAIL
USA
rinard@csail.mit.edu

Abstract
We present Konure, a new system that uses active learning to infer models of applications that access relational databases. Konure comprises a domain-specific language (each model is a program in this language) and associated inference algorithm that infers models of applications whose behavior can be expressed in this language. The inference algorithm generates inputs and database contents, runs the application, then observes the resulting database traffic and outputs to progressively refine its current model hypothesis. Because the technique works with only externally observable inputs, outputs, and database contents, it can infer the behavior of applications written in arbitrary languages using arbitrary coding styles (as long as the behavior of the application is expressible in the domain-specific language). Konure also implements a regenerator that produces a translated Python implementation of the application that systematically includes relevant security and error checks.

CCS Concepts → Software and its engineering → Source code generation; Domain specific languages; Software reverse engineering.

Keywords Active learning, Program inference, Program regeneration

1 Introduction
Progress in human societies is cumulative — each new generation builds on technology, knowledge, and experience accumulated over previous generations. Software collectively comprises one valuable store of human knowledge and experience as concretely realized in applications and software components. But there is currently no easy way to extract this knowledge and experience from its original context to productively deploy it into the new contexts that inevitably arise as societies evolve over time.

We present a new approach that uses active learning to infer models that capture the full core functionality of target applications or components. These models comprise a mobile reification of the original functionality that can then be regenerated to obtain a new, clean version of the functionality specialized for immediate deployment into new languages, systems, or contexts. The regeneration can also improve the application or component by (1) discarding coding errors, (2) automatically inserting security and/or privacy checks into the regenerated code, and/or (3) improving the performance by applying optimizations appropriate for the new platform or context. In the longer term, active learning plus regeneration may also enable new development methodologies that work with simple prototype implementations as (potentially noisy) specifications, then use regeneration to automatically obtain clean, efficient implementations specialized for the specific context into which they will be deployed.

Applications that access databases are ubiquitous in computing systems. Such applications translate commands from the application domain into operations on the database, with the application constructing strings that it then passes to the database to implement the operations. Web servers, which accept HTTP commands from web browsers and interact with back-end databases to retrieve relevant data, are one particularly prominent example of such applications. These applications are written in a range of languages, often quickly become poorly-understood legacy software, and, because they are typically directly exposed to Internet traffic, have been a prominent target for security attacks [12, 17, 34, 42, 43, 49, 50, 55]. Such applications therefore comprise a particularly compelling target for active learning plus regeneration.
1.1 Konure

We present a new system, Konure, that implements active learning plus regeneration for applications that access relational databases. Konure systematically constructs database contents and application inputs, runs the application with the database and inputs, then observes the resulting database traffic and outputs to infer a model of application behavior.

**Domain-Specific Language:** To make the inference problem tractable, Konure works with a domain-specific language (DSL) that (1) captures common application behavior and (2) supports a hierarchical inference algorithm that progressively explores application behavior to infer the model. The inference algorithm (conceptually) maintains a current hypothesis as a sentential form of the grammar that defines the DSL. At each step it selects a nonterminal in this sentential form, constructs inputs and database contents that enable it to determine the one production to apply to this nonterminal that is consistent with the behavior of the application, configures the database, runs the application, then observes the resulting database traffic and outputs to refine the hypothesis by applying the inferred production to the nonterminal. Although we designed the DSL to be an internal representation that is invisible to users, it is straightforward to provide direct access to the DSL so that users may write programs directly in the DSL.

**Guarantees:** If the application conforms to one of the models defined by the DSL, then the algorithm is guaranteed to (1) terminate and (2) produce an inferred program that correctly models the full core functionality of the application. Because Konure interacts with the application only via its inputs, outputs, and observed database interactions, it can infer and regenerate applications written in any language or in any coding style or methodology.

**Benefits:** Because the model captures core application functionality, it can help developers explore and better understand this functionality. Konure can also regenerate the application into a potentially different language and systematically applying coding patterns and additional checks that are known to be safe. Konure therefore targets several use cases: (1) security and/or performance through safe regenerated code, (2) portability to new platforms, (3) reverse engineering, and (4) program understanding.

1.2 Key Inferrability Properties

The design of the Konure DSL, together with its associated top-down inference algorithm, is a central contribution of this paper. We next outline several key properties of the design that enable inferrability via active learning.

In general, programs contain statements linked together by control and data flow. To promote control-flow inferrability, each statement in the DSL executes a query that is directly observable in the intercepted database traffic. All control flow is tied directly to the query results — If statements test if their query retrieves empty data; For statements iterate over all rows that their query retrieves, with all iterations independent. These properties help Konure generate a focused, tractably small sequence of inputs and database contents that (1) finds and traverses all relevant control-flow paths and (2) completely resolves each For loop with a single execution of two or more iterations.

To promote data flow inferrability, all data flows directly from either input parameters or retrieved query results to executed queries or outputs. Konure infers the data flow by matching concrete values in executed queries or outputs against the input parameter or retrieved query result with the same value. Konure eliminates potential data flow ambiguities by populating the input parameters and database contents with appropriately distinct concrete values.

The DSL is designed to enable the formulation of all properties of interest as quantifier-free SMT formulas. Konure leverages this property to construct inputs and databases that explore all relevant control-flow paths and deliver the distinct values that enable Konure to infer the data flow.

1.3 Experimental Results

We present case studies applying Konure to five applications: Fulcrum Task Manager [2], Kandan Chat Room [4], Enki Blogging Application [1], Blog [3], and a student registration application developed by a hostile DARPA Red Team to test SQL injection attack detection and nullification techniques. Our results show that Konure is able to successfully infer and regenerate commands that these applications use to retrieve data from the database.
1.4 Contributions
This paper makes the following contributions:

- **Inference Algorithm**: It presents a new algorithm for inferring the behavior of database-backed applications. Conceptually, the algorithm works with hypotheses represented as sentential forms of the grammar of Konure DSL.

At each stage the algorithm systematically constructs database contents and application inputs, runs the application, and observes the resulting database traffic and outputs to resolve a selected nonterminal in the current hypothesis. This approach enables Konure to work effectively with unbounded model spaces to infer models that capture the core functionality of the target class of applications.

- **DSL Design**: It presents a DSL for capturing specific computational patterns typically implemented by database-backed applications. The inference algorithm and DSL are designed together to enable an effective active learning algorithm that leverages the structure of the DSL to iteratively refine hypotheses represented as sentential forms in the DSL grammar.

- **Soundness and Completeness**: It presents a key theorem that states that if the behavior of the application conforms to the DSL, then the inference algorithm infers a program that correctly captures the full core functionality of the application.

- **Regeneration**: It shows how to regenerate new versions of the application that implement safe computational patterns and contain appropriate safety and security checks. The regenerator encapsulates the knowledge required to work effectively in the target domain and can eliminate coding errors that lead to incorrect application behavior or security vulnerabilities.

- **Experimental Results**: It presents results using Konure to infer and regenerate commands written in Ruby on Rails (RoR) and Java. The results highlight Konure’s ability to infer and regenerate robust, safe Python implementations of commands originally coded in other languages.

Database: The database contains: (1) a student table, which contains student ID (primary key), first name, last name, and password, (2) a teacher table, which contains teacher ID (primary key), first name, and last name, (3) a course table, which contains course ID (primary key), name, course number, and teacher ID, and (4) a registration table, which contains student ID and course ID.

First Execution: The Konure inference algorithm configures an empty database, then executes the application with the command "liststudentcourses -s 0 -p 1," which sets input parameters s and p to 0 and 1, respectively. Konure uses a transparent proxy (Figure 1) to observe the resulting database traffic, which the proxy collects as the concrete trace of the execution (Figure 3a). The query uses the constant '0', which comes from the input parameter s, and retrieves no data from the (empty) database. For this execution, the application produces no output.

Based on this information, Konure rewrites the concrete trace to replace concrete values (such as '0') with origin locations, which identify the source of each value. The result is a corresponding abstract trace (Figure 3b). This abstract trace contains a query q1 that selects all columns from the student table. The selection criterion is that the student ID must equal the input parameter s. Konure derives the origin locations by matching concrete values in the concrete trace against input values and values in the database.

Konure DSL: Figure 4 presents the (abstract) grammar for the Konure DSL. A program consists of a sequence of Query statements potentially terminated by an If or For statement. An If statement does not test an arbitrary condition — it instead only tests if the Query in the condition retrieves empty or nonempty data. Similarly, a For statement does not iterate over an arbitrary list — it instead iterates over the rows in its Query, executing its else clause if its Query retrieves zero rows. These restrictions (among others, Section 3.1) are key to the inferability of the DSL.

First Production: The first execution generated a single query (Figure 3a). Konure determines if this query came from a Seq, If, or For statement as follows. Working with the abstract trace in Figure 3b, Konure generates three sets of constraints. Each set specifies input parameters and database contents. The first set specifies that the query retrieves zero rows. The second specifies that the query retrieves at least one row. The third specifies that the query retrieves at least two rows. Konure invokes an SMT solver to obtain a context for each set of constraints. Each context identifies inputs and database values that satisfy the constraints.

In the example the third set of constraints is unsatisfiable, because the query accesses the primary key and there is at most one row for each value of the primary key. The first and second sets of constraints are satisfiable and therefore produce viable contexts. Konure executes the application in each of these contexts. Figures 3a and 5a present the recorded concrete traces; Figures 3b and 5b present the corresponding...
abstract traces. These traces indicate that the observable behavior of the application differs depending on whether the first Query retrieves no rows (Figures 3a and 3b) or at least one row (Figures 5a and 5b). Konure concludes that the first Query comes from an If statement and produces the first hypothesis in Figure 6. This hypothesis corresponds to applying an If production to the topmost Prog nonterminal.

Second Production: Konure next resolves the $P_3$ nonterminal in the first hypothesis. Working with the abstract trace in Figure 5b, Konure generates three sets of constraints that (1) force the first query (q1) to retrieve at least one row (this constraint forces the application to execute the then branch of the topmost If statement) and (2) force the second query (q2) to retrieve no rows, at least one row, and at least two rows, respectively. Once again, the first two sets of constraints produce viable contexts; the third is unsatisfiable.

Figure 5a presents the trace from the execution in which the second query retrieves no rows; Figure 7a presents the trace from the execution in which the second query retrieves at least one row. Because the traces differ (similarly to the above First Production), Konure resolves the nonterminal $P_1$ to an If statement. Figure 8 presents the resulting hypothesis.

Third Production: Konure next resolves the $P_3$ nonterminal. Working with the abstract trace produced by the previous step (Figure 7b), Konure generates constraints that force the application to execute $P_3$ once again with zero, at least one, or at least two rows retrieved by the first query in $P_3$ (q3 in Figure 7b). The solver generates viable contexts for all three sets of constraints. For the context with at least two rows retrieved, Konure collects the trace in Figure 9.

In this execution the third query retrieves two rows. The Konure loop detection algorithm examines the trace, detects the repetitive pattern in the last four queries, concludes that

(a) Concrete trace from the first execution. The database is empty and the query retrieves zero rows.

(b) Abstract trace from the first execution, converted from the concrete trace in Figure 3a. The conversion replaces the constant ‘0’ with its origin location, the input parameter $s$.

![Figure 3. First execution trace](image)

---

(a) Concrete trace from the second execution. The context is configured to ensure that the first query retrieves at least one row.

(b) Abstract trace from the second execution, converted from the concrete trace in Figure 5a. The conversion replaces the constants ‘5’ and ‘6’ with their origin locations, input parameters $s$ and $p$.

![Figure 5. Second execution trace](image)

(a) Concrete trace from the third execution. The context is configured so that the first and second queries retrieve at least one row and the third query retrieves zero rows.

(b) Abstract trace from the third execution, converted from the concrete trace in Figure 7a. The conversion replaces the constants ‘1’ and ‘2’ with their origin locations, input parameters $s$ and $p$.

![Figure 6. Hypothesis after resolving the topmost Prog nonterminal to an If statement.](image)

![Figure 7. Third execution trace](image)

![Figure 8. Hypothesis after resolving $P_1$ (Figure 6).](image)
The third query retrieves two rows. The final four queries are grammars as noisy specifications of just such an obscure corner case bug. We view such programs during the inference [51, 61]. The SQL injection attack programs that contain obscure corner-case bugs not exercised during the inference [51, 61]. The SQL injection attack vulnerability present in the original program. A model of the command. It can then regenerate the context (Col = Orig). Selecting from multiple tables corresponds to an SQL join operation. The query stores the retrieved data in a unique variable (y) for later use. All variables must be defined before they are used.

Developed Understanding: In a deployed system, we expect that developers would be given examples and documentation that outlines the Konure DSL and the model of computation. We expect that this information, along with experience using Konure, would enable developers to work productively with Konure using programs written in their language of choice.

3 Design
The Konure inference algorithm is constructive [11] — instead of enumerating candidate solutions, the algorithm constructs the solution progressively every time Konure finds an interesting behavior of the application. Conceptually, the algorithm starts with a Prog nonterminal as its initial hypothesis, then progressively resolves Prog nonterminals until it completely infers the program. The algorithm (conceptually) maintains a sentential form in the Konure DSL, with nonterminals denoting hidden parts that are left to infer. The inference proceeds by expanding nonterminals until it obtains a complete program. As Konure recursively traverses the paths through the program as expressed in the DSL, it maintains path constraints that lead to the next part of the program to infer. Instead of maintaining the current hypothesis as an explicit sentential form, Konure represents the hypothesis implicitly in the data structures and recursive structure of the inference algorithm as it executes.

3.1 Konure Domain-Specific Language
Konure infers application functionality that can be expressed in the Konure DSL. We present the grammar for the Konure DSL in Figure 4. Each query in this DSL performs an SQL select operation that retrieves data from specified columns in specified tables. Our current DSL supports SQL where clauses that select rows in which one column has the same value as another column (Col = Col) or the same value as a value in the context (Col = Orig). Selecting from multiple tables corresponds to an SQL join operation. The query stores the retrieved data in a unique variable (y) for later use. All variables must be defined before they are used.

To enable the Konure inference algorithm to effectively distinguish If statements from Seq statements, Konure requires the two branches of each If statement to start with queries that have different skeletons (or one of the branches must be empty). To facilitate effective loop detection, Konure requires the first query after any query that may retrieve multiple rows to have a skeleton that is distinct from all subsequent queries. Konure also requires that the program have no nested loops. Each Print statement is associated with a query and only prints values retrieved by its query.

![Figure 9. Concrete trace from an execution to resolve P3 (Figure 8). The third query retrieves two rows. The final four queries are generated by a loop that iterates over the retrieved two rows.](image)

![Figure 10. Hypothesis after resolving P3 (Figure 8).](image)
Definition 1. The skeleton of a program \( P \in \text{Prog} \) is a program that is syntactically identical to \( P \) except for replacing syntactic components derived from the Orig nonterminal (Figure 4) with an empty placeholder \( \diamond \).

Definition 2. For any program \( P \in \text{Prog} \), \( \bar{P} \) is the semantically equivalent program obtained from \( P \) by discarding unreachable branches in If and For statements, downgrading For statements with empty loop bodies or loop bodies that execute at most once to If statements, and downgrading If statements with an unreachable branch or two semantically equivalent branches to Seq statements.

Definition 3. For any program \( P \in \text{Prog} \), \( \mathcal{T}(P) \) is the set of queries in \( P \) that retrieve at least two rows in some execution.\(^1\) \( \mathcal{R}(P) \) is the set of all queries \( Q \) in \( P \) with two subsequent queries \( Q_1 \) and \( Q_2 \) such that \( Q_1 \) immediately follows \( Q \) in the program, \( Q_1 \) does not appear as the first query of an else branch of an If or For statement, \( Q_2 \) occurs after \( Q_1 \) in the program, and \( Q_1 \) and \( Q_2 \) have the same skeleton. \( D(P) \) is a predicate that is true if and only if the two branches of all conditional statements in \( P \) start with queries with different skeletons (or one of the branches is empty).

Definition 4 (The Konure DSL). We define the Konure DSL as the set of programs \( \mathcal{K} \subset \text{Prog} \) defined as:

\[
\mathcal{K} = \{ \bar{P} \mid P \in \text{Prog}, \mathcal{T}(\bar{P}) \cap \mathcal{R}(\bar{P}) = \emptyset, D(\bar{P}) = \text{true} \}
\]

The first restriction, \( \mathcal{T}(\bar{P}) \cap \mathcal{R}(\bar{P}) = \emptyset \), states that if a query may retrieve multiple rows from the database, then the next query does not share a skeleton with any other subsequent query in the program. This restriction facilitates loop detection by eliminating repeated queries that do not come from iterations of the same loop (Section 3.2).\(^2\) The second restriction, \( D(\bar{P}) = \text{true} \), states that the two branches of any If statement in \( \bar{P} \) must start with queries with different skeletons (or one of the branches must be empty). Intuitively, this restriction enables Konure to efficiently distinguish Seq from If statements (Section 3.5).

Because of the focused expressive power of the Konure DSL, it is possible to decide all relevant conditions statically, rewrite \( P \) to \( \bar{P} \), and determine if \( \bar{P} \in \mathcal{K} \). Note that because programs \( P \in \mathcal{K} \) may reference values using distinct but semantically equivalent variables, \( \mathcal{K} \) is not a true canonical form, i.e., there may be distinct but semantically equivalent programs in \( \mathcal{K} \). It is possible, however, to eliminate such equivalences by replacing each variable with the first semantically equivalent variable to occur in the program. This transformation is implementable with an SMT solver and eliminates distinct but semantically equivalent programs to deliver a true canonical form for the Konure DSL.

1 A query will never retrieve more than one row if, for example, it selects rows that have a specific primary key value.

2 Our implemented Konure prototype deploys a more sophisticated loop detection algorithm that enables it to relax this restriction.

Expressiveness and Limitations: The DSL captures a wide range of applications that display data from a database by retrieving data based on inputs and database contents. Meanwhile, these applications are restrictive enough to be inferred efficiently. To support more sophisticated control-flow logic in database-backed applications, we anticipate that an inference algorithm would need to access more runtime information or have more domain knowledge. Negative examples that are straightforward to support include applications with SQL queries that involve relational comparisons (besides equality and membership checks), simple arithmetics, constants, or aggregate functions. It is straightforward because (1) we use an SMT solver that supports solving constraints involving these operators and (2) the operators are directly present in the intercepted SQL queries. Because experience with SMT solvers in other contexts shows that these solvers readily support formulas with these kinds of operators and constraints, we do not anticipate any significant performance issues with this extension.

3.2 Konure Inference Algorithm

We present the Konure inference algorithm (Algorithm 1) for a program \( P \) that implements a single command. For programs with multiple commands, Konure uses Algorithm 1 to infer each command in turn. The algorithm configures an empty database, sets the parameters to distinct values, invokes Algorithm 2 to run the program and obtain an initial trace, then invokes Algorithm 3 to recursively infer the program. The inference algorithm works with deduplicated annotated traces \( t \) that record one iteration of each executed loop, so that the structure of the trace matches the corresponding path through the program.

Definition 5. A context \( \sigma = \langle \sigma_I, \sigma_D, \sigma_R \rangle \in \text{Context} \) (Figure 11) contains value mappings for the input parameters \( \sigma_I \in \text{Input} \), database contents \( \sigma_D \in \text{Database} \), and results retrieved by database queries \( \sigma_R \in \text{Result} \). The input context \( \sigma_I \) maps input parameter variables \( x \in \text{Variable} \) to concrete values. The database context \( \sigma_D \) maps database locations (identified by a table name, a row number, and a column name) to concrete values. The results context \( \sigma_R \) maps each query result variable \( y \in \text{Variable} \) to a list of rows, with each value in each row identified by the table and column from which it was retrieved.

Definition 6. We denote the concrete trace from executing a program \( P \in \text{Prog} \) in context \( \sigma \in \text{Context} \) as \( \sigma(P) \in \text{CTrace} \) (Figure 12a). A concrete trace contains the intercepted SQL traffic (specifically, the queries \( CQuery^* \) and corresponding retrieved rows \( CData^* \)) and observed outputs \( CVal^* \).

Definition 7. \( \boxed{P} \) denotes the black box executable of a program \( P \in \text{Prog} \), i.e., executing \( \boxed{P} \) in context \( \sigma \in \text{Context} \) produces the concrete trace \( \sigma(P) \). Note that Konure does not access the source code of \( P \) when it executes \( \boxed{P} \).
Each node in the loop layout tree corresponds to a query in $P$. Each node represents whether a loop in $P$ iterates over the corresponding query multiple times. In particular, when a loop in $P$ iterates over a query multiple times, the query’s corresponding node in the loop layout tree has multiple subtrees, with each subtree corresponding to an iteration of the loop. We convert a list of query-result pairs into a loop layout tree in DetectLoops, which we discuss below.

**Definition 12.** An annotated trace is an ordered list of annotated query tuples. Each tuple, denoted as $(Q, r, \lambda)$, has three components obtained from a query $Q \in Query$. The first component is the query $Q$. The second component is the number of rows retrieved by $Q$ during an execution. The third component is the annotated information of whether a loop was found to iterate over data retrieved by $Q$. If such loop was found then $\lambda$ is a nonnegative integer that indicates the iteration index. If no such loop was found then $\lambda = \text{NotLoop}$. Each path from the root of the loop layout tree to a leaf generates a corresponding annotated trace.

**Definition 13.** A path constraint $W = ((Q_1, r_1, s_1), \ldots, (Q_n, r_n, s_n))$, consists of a sequence of queries $Q_1, \ldots, Q_n \in Query$, row count constraints $r_1, \ldots, r_n$, and boolean flags $s_1, \ldots, s_n$. Each $r_i$ specifies the range of the number of rows in a query result, denoted as one of $(= 0), (\geq 1)$, or $(\geq 2)$. Each $s_i$ is true if a loop iterates over the corresponding retrieved rows and false otherwise.

**Definition 14.** An annotated trace $t$ is consistent with path constraint $W$, denoted as $t \sim W$, if the path specified in $W$ is not longer than $t$, each query in $t$ matches the corresponding query in $W$, and each row count in $t$ matches the corresponding requirement in $W$.

**GetTrace:** Algorithm 2 takes an executable program $P$, path constraint $W$, and context $\sigma$ as parameters. It first invokes Execute, which runs $P$ in context $\sigma$ to obtain the flat list $e$ of query-result pairs converted from the concrete trace that $P$ generates when it runs. It then invokes DetectLoops, which runs the Konure loop detection algorithm to produce the loop layout tree $l$. Finally, MATCHPATH generates an annotated trace that corresponds to a path through $l$ consistent with the path constraint $W$.

**Execute:** The Execute procedure takes an executable program $P$ and a context $\sigma = \langle \sigma_I, \sigma_D, \sigma_R \rangle \in Context$. It first populates the database with contents specified in $\sigma_D$ and then executes $P$ with input parameters specified in $\sigma_I$. It collects the outputs and database traffic, i.e., the concrete trace $\sigma(P)$ (Figure 1). Execute converts the concrete trace into an abstract trace, converts the abstract trace into a list of query-result pairs, then returns this list of pairs.

**DetectLoops:** The DetectLoops procedure takes a list of query-result pairs and constructs a loop layout tree. (1) If the first query retrieves $r \geq 2$ rows, DetectLoops checks if the skeleton of the second query is repeated exactly $r$ times in the tail of the trace. If the repetitions match, DetectLoops...
Algorithm 1 Infer an executable program

Input: \( P \) is the executable of a program \( P \in X \).
Output: Program equivalent to \( P \).

1: \textbf{procedure} INFER\( \langle P \rangle \)
2: \hspace{1em} \( \sigma \leftarrow \text{Database empty, input parameters distinct} \)
3: \hspace{1em} \( t \leftarrow \text{GETTRACE}(P, \text{Nil}, \sigma) \)
4: \hspace{1em} \textbf{return} INFER\( \langle P \rangle \text{Nil}, t \)
5: \textbf{end procedure} 

Algorithm 2 Execute a program and deduplicate the trace according to a path constraint

Input: \( P \) is the executable of a program \( P \in X \).
Input: \( W \) is a path constraint.
Input: \( \sigma \) is a context that satisfies \( W \).
Output: Annotated trace \( t \), \( t \sim W \), from executing \( P \) with \( \sigma \).

1: \textbf{procedure} \text{GETTRACE}(P, W, \sigma) \n2: \hspace{1em} \( e \leftarrow \text{EXECUTE}(P, \sigma) \)
3: \hspace{1em} \( l \leftarrow \text{DETECTLOOPS}(e) \)
4: \hspace{1em} \textbf{return} MATCHPATH(l, W) \n5: \textbf{end procedure}

determines that a loop iterates over the first query in the trace, splits the trace into \( r \) segments that each correspond to an iteration of the loop, recursively constructs a loop layout tree for each segment, and then inserts the recursively constructed loop layout trees as the children of the first query. (2) In all other scenarios, DETECTLOOPS determines that no loop iterates over the first query in the trace, recursively constructs a loop layout tree for the tail of the trace, and then inserts the recursively constructed loop layout tree as the child of the first query of the trace.

\textbf{INFER\( \langle P \rangle \): Algorithm 3} implements the main \textsc{Konure} inference algorithm. This algorithm recursively explores all relevant paths through the program, resolving \textsc{Prog} nonterminals as they are (conceptually) encountered. \textbf{Algorithm 3} takes as parameters the executable \( P \) of the program to infer and a split annotated trace consisting of a prefix \( s_1 \) that corresponds to an explored path through the program and a suffix \( s_2 \) from the remaining unexplored part of the program. The first query \( Q \) in \( s_2 \) is generated by the next \textsc{Prog} nonterminal to resolve. \textsc{Konure} therefore determines whether the query \( Q \) was generated by a \textsc{Seq}, If, or For statement, then recurses to infer the remaining parts of the program.

\textsc{Konure} makes this determination by examining three deduplicated annotated traces \( t_0, t_1, \text{and } t_2 \). All of these traces are from executions that follow the same path to \( Q \) as \( s_1 \). In the execution that generated \( t_0 \), \( Q \) retrieves zero rows, in the execution that generated \( t_1 \), \( Q \) retrieves at least one row, and in the execution that generated \( t_2 \), \( Q \) retrieves at least two rows. If \textsc{Konure} detects a loop in \( t_2 \) over the rows that \( Q \) retrieves, it infers that \( Q \) was generated by a For statement (line 14 in \textbf{Algorithm 3}). Otherwise, it examines \( t_0 \) and \( t_1 \)

Algorithm 3 Recursively infer a subprogram

Input: \( P \) is the executable of a program \( P \in X \).
Input: \( s_1 \) is a prefix of an annotated trace.
Input: \( s_2 \) is a suffix of an annotated trace.
Output: Subprogram equivalent to \( P \)’s subprogram after trace \( s_1 \).

1: \textbf{procedure} INFER\( \langle P \rangle (s_1, s_2) \)
2: \hspace{1em} \textbf{if} \( s_2 = \text{Nil} \) \textbf{then} \textbf{return} \( \epsilon \) \textbf{▷ Prog} := \( \epsilon \)
3: \hspace{1em} \textbf{end if}
4: \hspace{1em} \( k \leftarrow \text{The length of } s_1 \)
5: \hspace{1em} \( Q \leftarrow \text{The first query in } s_2 \)
6: \hspace{1em} \textbf{for} \( i = 0, 1, 2 \) \textbf{do}
7: \hspace{1em} \( W_i \leftarrow \text{MAKEPATHCONSTRAINT}(s_1, Q, i) \)
8: \hspace{1em} \( (f_i, t_i) \leftarrow \text{SOLVEANDGETTRACE}(P, W_i) \)
9: \hspace{1em} \textbf{if} \( f_i \) \textbf{then}
10: \hspace{1em} \( t_{i,1} \leftarrow t_i[1, \ldots, (k + 1)] \) \textbf{▷ New trace prefix}
11: \hspace{1em} \( t_{i,2} \leftarrow t_i[(k + 2), \ldots] \) \textbf{▷ New trace suffix}
12: \hspace{1em} \textbf{end if}
13: \hspace{1em} \textbf{end for}
14: \hspace{1em} \textbf{if} \( f_2 \) and found loop on the last query in \( t_{2,1} \) \textbf{then}
15: \hspace{1em} \( b_1 \leftarrow \text{INFER\( \langle P \rangle (t_{2,1}, t_{2,2}) \)
16: \hspace{1em} \textbf{if} \( f_0 \) \textbf{then} \( b_f \leftarrow \text{INFER\( \langle P \rangle (t_{0,1}, t_{0,2}) \)
17: \hspace{1em} \textbf{else} \( b_f \leftarrow \epsilon \)
18: \hspace{1em} \textbf{end if}
19: \hspace{1em} \textbf{return} “for } Q \text{ do } b_t \text{ else } b_f ” \textbf{▷ Prog} := \( \text{For} \)
20: \hspace{1em} \textbf{else if} \( f_0 \) and \( f_1 \) and \((t_{0,2} = \text{Nil and } t_{1,2} \neq \text{Nil}) \) or \((t_{0,2} \neq \text{Nil and } t_{1,2} = \text{Nil}) \) or the first queries in \( t_{0,2} \) and \( t_{1,2} \) have different skeletons) \textbf{then}
21: \hspace{1em} \( b_t \leftarrow \text{INFER\( \langle P \rangle (t_{1,1}, t_{1,2}) \)
22: \hspace{1em} \( b_f \leftarrow \text{INFER\( \langle P \rangle (t_{0,1}, t_{0,2}) \)
23: \hspace{1em} \textbf{return} “if } Q \text{ then } b_t \text{ else } b_f ” \textbf{▷ Prog} := \( \text{If} \)
24: \hspace{1em} \textbf{else}
25: \hspace{1em} \textbf{if} \( f_0 \) \textbf{then} \( b \leftarrow \text{INFER\( \langle P \rangle (t_{0,1}, t_{0,2}) \)
26: \hspace{1em} \textbf{else} \( b \leftarrow \text{INFER\( \langle P \rangle (t_{1,1}, t_{1,2}) \)
27: \hspace{1em} \textbf{end if}
28: \hspace{1em} \textbf{return} “Q } b “ \textbf{▷ Prog} := \( \text{Seq} \)
29: \hspace{1em} \textbf{end if}
30: \hspace{1em} \textbf{end procedure}

Algorithm 4 Obtain a deduplicated annotated trace that satisfies a path constraint

Input: \( P \) is the executable of a program \( P \in X \).
Input: \( W \) is a path constraint.
Output: The first component represents the satisfiability of \( W \). When satisfiable, the second component is an annotated trace \( t \) where \( t \sim W \).

1: \textbf{procedure} \text{SOLVEANDGETTRACE}(P, W) \n2: \hspace{1em} \( \sigma \leftarrow \text{SOLVE}(W) \)
3: \hspace{1em} \textbf{if} \( \sigma = \text{Unsat} \) \textbf{then}
4: \hspace{1em} \textbf{return} false, Nil \n5: \hspace{1em} \textbf{else}
6: \hspace{1em} \( t \leftarrow \text{GETTRACE}(P, W, \sigma) \)
7: \hspace{1em} \textbf{return} true, t \n8: \hspace{1em} \textbf{end if}
9: \hspace{1em} \textbf{end procedure}
to determine if $Q$ was generated by an If statement (line 20 in Algorithm 3) or a Seq statement (line 24 in Algorithm 3) — conceptually, if the queries that follow $Q$ in $t_0$ and $t_1$ differ, then $Q$ is generated by an If statement, otherwise it is generated by a Seq statement.

Konure obtains traces $t_0$, $t_1$, and $t_2$ by using MakePathConstraint to construct three path constraints $W_0$, $W_1$, and $W_2$, then using an SMT solver to obtain contexts $\sigma_0$, $\sigma_1$, and $\sigma_2$ that cause $P$ to produce (deduplicated annotated) traces $t_0$, $t_1$, and $t_2$ (Algorithm 4). If $W_i$ is satisfiable then $t_i \sim W_i$.

MakePathConstraint takes the trace prefix $s_i$, the subsequent query $Q$, and an integer $i$. The procedure constructs a new path constraint, $W_i$, which specifies that any satisfying context must enable the program to execute down the same path as $s_i$, then perform query $Q$ and retrieve a certain number of rows as specified by $i$. In particular, if $i = 0$ then $Q$ is required to retrieve zero rows. If $i = 1$ or $i = 2$ then $Q$ is required to retrieve at least $i$ rows.

As presented, Algorithm 1 (and associated soundness proof Theorem 3) does not work with Print statements. Our implemented Konure prototype infers Print statements by correlating values that appear in the output with values observed in the database traffic. Recall that in the Konure DSL, each Print statement is associated with a query and only prints values retrieved by its query. This restriction enables Konure to associate each Print statement with its corresponding query.

### 3.3 Path Constraint Solver

**Definition 15.** For a query $Q \in Query$ and a context $\sigma = (\sigma_1, \sigma_2, \sigma_3) \in Context$, $\sigma(Q)$ denotes the result from evaluating $Q$ in $\sigma$. Evaluating $Q$ involves replacing origin locations in $Q$ with their values in $\sigma_1$ and $\sigma_3$, rewriting the query in SQL syntax, then performing the SQL query on $\sigma_2$. The query result contains an ordered list of rows. $|\sigma(Q)|$ denotes the number of rows in $\sigma(Q)$. $\sigma|Q, y \rightarrow z|$ denotes the variable that stores the retrieved data. $\sigma|Q, y \rightarrow z|$ denotes the new context after updating $\sigma$ to map $Q, y$ to $z$.

**Definition 16.** A context $\sigma \in Context$ satisfies a path constraint $W = (Q_1, r_1, s_1), \ldots, (Q_n, r_n, s_n)$ if (1) a sequence of contexts $\sigma_1, \ldots, \sigma_n \in Context$ are updated according to the evaluation of the queries $Q_1, \ldots, Q_n$ in $\sigma$ and (2) $|\sigma_i(Q_i)|$ satisfies $r_i$ for all $i = 1, \ldots, n$. Specifically, the context sequence satisfies $\sigma_1 = \gamma$ and for all $i = 1, \ldots, n - 1$,

$$\sigma_{i+1} = \begin{cases} \sigma_i|Q_i, y \rightarrow \sigma_i(Q_i)|, & s_i = \text{false or } |\sigma_i(Q_i)| = 0 \\ \sigma_i|Q_i, y \rightarrow \sigma_i(Q_i)[k_i]|, & s_i = \text{true and } |\sigma_i(Q_i)| \geq 1 \end{cases}$$

for some integer $k_i$ such that if $|\sigma_i(Q_i)| \geq 1$ then $1 \leq k_i \leq |\sigma_i(Q_i)|$. We call $\sigma_1$ the context after updating $\sigma$ with $W$.

Solve takes a path constraint $W$ and uses an SMT solver to solve for a context $\sigma \in Context$ that satisfies $W$. The procedure returns a satisfying $\sigma$ if it exists and returns “Unsat” otherwise.

Like many database test data generation approaches \cite{23, 41, 68, 70, 71, 74}, Solve uses a row-based approach to translate path constraints into SMT formulas. For each query $Q_i$ in $W$ that is required to retrieve at least one or at least two rows, Solve generates variables that model the required number of rows of the relevant tables. It then generates constraints that require the values of these variables to satisfy the selection criteria of $Q_i$. It also generates constraints that require primary keys to be unique.

For each query $Q_i$ that is required to retrieve zero rows, Solve generates constraints that ensure that none of the values in the relevant tables satisfy the selection criteria of $Q_i$. If $Q_i$ occurs in a loop, the constraints only enforce that $Q_i$ retrieves zero rows in at least one iteration of the loop (as opposed to always retrieving zero rows). Here, loop iterations map easily to the rows of unknown variables, because loops in the Konure DSL are designed to iterate over rows of data.

### 3.4 Origin Location Disambiguation

Recall that an origin location $O \in Orig$ in a program $P \in Prog$ is an occurrence of a variable $x$ or a column reference $y.Col$ in $P$. Concrete traces contain intercepted queries executed by the program. In these intercepted queries, the origin locations have been replaced by the corresponding concrete values from the execution. When Konure converts concrete traces into abstract traces, it restores the origin locations by matching concrete values across query results and input parameters to translate the concrete values back into their corresponding origin locations.

Because Konure uses a general SMT solver to obtain contexts $\sigma$ that satisfy specified path constraints $W$, the contexts may introduce ambiguity by coincidentally generating the same value in different input parameters or database locations. This ambiguity shows up as different origin locations $O_1$ and $O_2$ that both contain the same concrete value to translate. Konure resolves the ambiguity as follows:

- Konure first asks the solver if it is possible to reproduce the path to the ambiguous concrete value with the additional constraint that $O_1$ and $O_2$ hold disjoint values. If so, the resulting execution resolves the ambiguity.
- Otherwise, Konure asks the solver if it is possible to reproduce this path with the additional constraint that $O_1$ holds a value not in $O_2$. If not, the values in $O_1$ are a subset of the values in $O_2$. Konure similarly uses the solver to determine if the values in $O_2$ are a subset of the values in $O_1$. If $O_1$ and $O_2$ are subsets of each other, they hold the same values and Konure can use either origin location.
- Otherwise, there exists an execution in which $O_1$ has at least one value $v$ not in $O_2$ (or vice-versa). Konure asks the solver to produce a context that generates this execution. The resulting execution in this context resolves the ambiguity — if the value $v$ ever appears in the same location as the concrete value, then Konure uses $O_1$ as the origin location, otherwise it uses $O_2$. 

3.5 Soundness Proof Outline

We next outline the structure of a soundness proof for the core Konure inference algorithm (Algorithm 1).

**Definition 17.** Origin locations $O_1, O_2 \in \text{Orig}$ are equivalent with respect to path constraint $W$, denoted as $O_1 \equiv_W O_2$, if for any context $\sigma \in \text{Context}$ that satisfies $W$, $O_1, O_2$ hold the same values in the context after updating $\sigma$ with $W$.

**Definition 18.** For a program $P \in \text{Prog}$ and a context $\sigma \in \text{Context}$, $\sigma \vdash P \circ \loopexec e$ denotes evaluating $P$ in $\sigma$ to obtain a list of query-result pairs $e$. $\sigma \vdash P \llint \looploops l$ denotes evaluating $P$ in $\sigma$ to obtain a loop layout tree $l$.

**Definition 19.** For programs $P, P' \in \text{Prog}$ and annotated trace $t$, we use the notation $P \Rightarrow P'$ to denote that traversing the AST of $P$ from top to bottom, by following the row counts in $t$, leads to a subtree $P'$.

**Definition 20.** The size of a program $P \in \text{Prog}$ is denoted as $|P|$ and defined as the number of times that the AST of $P$ applies a production to expand a "Prog" nonterminal.

**Proposition 1 (Solver).** For any path constraint $W$, the procedure $\text{Solve}(W)$ returns a context $\sigma \in \text{Context}$ if and only if $W$ is satisfiable.

**Rationale.** The path constraint solver outlined in Section 3.3 asks the SMT solver a question that is equisatisfiable as the existence of a satisfying context. Since the logical formulas are quantifier-free and involve only equality checks, their satisfiability is efficiently decidable [18].

**Proposition 2 (Disambiguation).** For any program $P \in \mathcal{K}$ and context $\sigma \in \text{Context}$, if $\sigma \vdash P \circ \loopexec e, \text{execute}(P, \sigma) = e'$, and $e' = ((Q_1, r_1), \ldots, (Q_n, r_n))$, then $e' = ((Q'_1, r_1), \ldots, (Q'_n, r_n))$, where $Q_i$ and $Q'_i$ are identical except for the use of different but equivalent origin locations for any $i = 1, \ldots, n$.

**Rationale.** The disambiguation procedure (Section 3.4) asks the SMT solver a question that equivalently encodes the relationship between origin locations. By Proposition 1, we obtain a correct list of query-result pairs after disambiguating the traces obtained from program execution.

**Theorem 1 (Loop Detection).** For any program $P \in \mathcal{K}$ and context $\sigma \in \text{Context}$, if $\sigma \vdash P \circ \loopexec e$ and $\sigma \vdash P \llint \looploops l$ then $\text{DetectLoops}(e) = l$.

**Proof.** By induction on the derivation of $P$.

**Theorem 2 (Core Recursion).** For any programs $P, P' \in \mathcal{K}$ and $P' \in \text{Prog}$ and annotated traces $t_1, t_2$, if $P \vdash t_1 \Rightarrow P'$ and $P' \vdash t_2 \Rightarrow \epsilon$ then $\text{InferProg}(P, t_1, t_2)$ and $P'$ are identical except for the use of different but equivalent origin locations.

**Proof.** The proof first performs case analysis of the relationship between the possible first production in $P'$, properties of the path constraints $W_i$, and values $f_i, t_{i,j}$ from the executions of $P$ in Algorithm 3 to show that Algorithm 3 chooses the correct first production in $P'$. The proof then proceeds by induction on the productions applied to derive $P'$.

**Theorem 3 (Soundness of Inference).** For any program $P \in \mathcal{K}$, $\text{Infer}(P)$ and $P$ are identical except for the use of different but equivalent origin locations.

**Proof.** The proof first shows that the initial trace $t$ at line 3 of Algorithm 1 satisfies $P \rightarrow \epsilon$. The rest of the proof follows from Theorem 2.

**Theorem 4 (Complexity).** For any program $P \in \mathcal{K}$, the execution of $\text{InferProg}(P)$ calls the $\text{InferProg}$ procedure at most $|P|$ times.

Each recursive call to $\text{InferProg}$ constructs a subprogram for $P \in \mathcal{K}$, and the algorithm does not need to backtrack because it never makes an incorrect hypothesis choice. Each step is conclusive — only one nonterminal expansion is possible. The algorithm also does not involve an equivalence check.

The inference algorithm terminates when it has fully constructed the AST of $P$. More concretely, the number of recursive calls to $\text{InferProg}$ is linear in the size of the given program. Critically, this number of executions is bounded by the size of the source code of $P$, not by the number of iterations that any loop executes. It works because any loop’s iterations are independent from each other (Figure 4).

We prove Theorems 2 through 4 only for programs $P \in \mathcal{K}$ (and with no Print statements). However, the proofs rely only on the black box execution of $P$ in $\text{execute}(P, \sigma)$. The soundness properties therefore hold for arbitrary programs written in arbitrary languages as long as the program’s externally observable behavior is equivalent to that of some program $P \in \mathcal{K}$.

4 Experimental Results

We implemented a Konure prototype and acquired five benchmark applications to evaluate this prototype. Each application takes commands as input, translates the commands into SQL queries against the relational database, and returns results extracted from the results of the queries.

4.1 Applications and Commands

Our benchmark applications include:

- **Fulcrum Task Manager:** Fulcrum [2] is an open source project planning tool, built with Ruby on Rails (RoR), with over 1500 stars on GitHub. Fulcrum maintains multiple projects. Each project may contain multiple stories. Each story may contain multiple notes. Fulcrum commands enable users to navigate the contents of projects, stories, and notes, as well as the users who created these contents.

- **Kandan Chat Room:** Kandan [4] is an open source chat room application, built with (RoR), with over 2700 stars on GitHub. Kandan maintains multiple chat rooms (so-called channels) that users can access. Its commands enable users to navigate chat rooms and messages (so-called activities) and display relevant user information.
• **Enki Blogging Application**: Enki [1] is an open source blogging application, built with RoR, with over 800 stars on GitHub. Enki maintains multiple pages and posts, each of which may have comments. Enki commands enable the author of the blog to navigate pages, posts, and comments.

• **Blog**: The Blog application is an example obtained from the RoR website [3]. Blog maintains information about blog articles and blog comments. It implements a command that retrieves the article data associated with the blog. The second column (**Command** type) presents the name of the command. The first column (**Source Code** for the regenerated commands is available [5].

- **Konure code will be standard in many usage contexts.** We then anticipate that the automated introduction of such boilerplate code into the regenerated code for specified commands. We also support the insertion of boilerplate password checking statements, if statements, for statements, and the number of potentially ambiguous origin locations.

  The next column (**Run**s) presents the number of invocations of the Z3 SMT solver that Konure executed to infer the model for the command. Because Konure may invoke the SMT solver multiple times for each inference step, the number of Z3 invocations is larger than the number of application executions. The next column (**Time**) presents the wall-clock time required to infer the model for each command. We measured time on a Ubuntu 16.04 virtual machine with 2 cores and 2 GB memory. The host machine uses a processor with 4 cores (3.4 GHz Intel Core i5) and has 24 GB 1600 MHz DDR3 memory. The times vary from less than a minute to about two hours. In general, the times are positively correlated with the number of solves, the length of the programs, and the number of potentially ambiguous origin locations.

  Most of the inference time was spent on solving for alternative database contents to satisfy various constraints. The inference time also includes the time required to set up, tear down, and execute the applications (and their web servers) in the Konure environment.

  The remaining columns present statistics from the regenerated Python implementations. The **LoC**, SQL, **if**, **for**, and **Output** columns present the number of lines of code, SQL statements, if statements, for statements, and the number of lines that generate output. We note that the regenerated programs are free of SQL injection attack vulnerabilities. These vulnerabilities are present in the original student registration application from the DARPA Red Team.

  We recruited a software engineer with three years of experience working with Ruby on Rails applications to evaluate the Konure inference and regeneration by comparing the original Ruby on Rails and regenerated Python versions of each command. This manual comparison indicated that the inferred and regenerated commands were consistent with the original Ruby on Rails implementations. The evaluation also highlighted how the Rails framework, specifically the ActiveRecord object relational mapping abstraction, implicitly generates substantial database traffic as it assembles the object state (including the state of objects on which it depends) when initially loading the object. This code that generates this database traffic is explicit and therefore directly visible in the regenerated Python code.

### 4.2 Performance on Synthetic Commands

We evaluate the scalability of the inference algorithm with experiments on the following classes of synthetic commands. The source code for these commands is available [5].
Table 1. Inference effort and regenerated code size

<table>
<thead>
<tr>
<th>Command</th>
<th>Params</th>
<th>App</th>
<th>Runs</th>
<th>Solves</th>
<th>Time</th>
<th>LoC</th>
<th>SQL</th>
<th>If</th>
<th>For</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_home</td>
<td>1</td>
<td>Fulcrum</td>
<td>5</td>
<td>43</td>
<td>8 mins</td>
<td>21</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>get_projects</td>
<td>1</td>
<td>Fulcrum</td>
<td>5</td>
<td>43</td>
<td>8 mins</td>
<td>21</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>get_projects_id</td>
<td>2</td>
<td>Fulcrum</td>
<td>12</td>
<td>124</td>
<td>29 mins</td>
<td>25</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>get_projects_id_stories</td>
<td>2</td>
<td>Fulcrum</td>
<td>11</td>
<td>42</td>
<td>7 mins</td>
<td>31</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>get_projects_id_stories_id</td>
<td>3</td>
<td>Fulcrum</td>
<td>12</td>
<td>50</td>
<td>8 mins</td>
<td>31</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>get_projects_id_stories_id_notes</td>
<td>3</td>
<td>Fulcrum</td>
<td>11</td>
<td>41</td>
<td>8 mins</td>
<td>24</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>get_projects_id_stories_id_notes_id</td>
<td>4</td>
<td>Fulcrum</td>
<td>13</td>
<td>46</td>
<td>10 mins</td>
<td>28</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>get_projects_id_users</td>
<td>2</td>
<td>Fulcrum</td>
<td>12</td>
<td>124</td>
<td>30 mins</td>
<td>25</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>get_channels</td>
<td>1</td>
<td>Kandan</td>
<td>21</td>
<td>125</td>
<td>105 mins</td>
<td>63</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>27</td>
</tr>
<tr>
<td>get_channels_id_activities</td>
<td>2</td>
<td>Kandan</td>
<td>23</td>
<td>242</td>
<td>39 mins</td>
<td>49</td>
<td>16</td>
<td>6</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>get_channels_id_activities_id</td>
<td>3</td>
<td>Kandan</td>
<td>14</td>
<td>18</td>
<td>7 mins</td>
<td>25</td>
<td>11</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>get_me</td>
<td>1</td>
<td>Kandan</td>
<td>11</td>
<td>139</td>
<td>6 mins</td>
<td>44</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>get_users</td>
<td>1</td>
<td>Kandan</td>
<td>15</td>
<td>236</td>
<td>9 mins</td>
<td>67</td>
<td>11</td>
<td>3</td>
<td>0</td>
<td>45</td>
</tr>
<tr>
<td>get_users_id</td>
<td>2</td>
<td>Kandan</td>
<td>11</td>
<td>139</td>
<td>6 mins</td>
<td>44</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>get_admin_comments_id</td>
<td>1</td>
<td>Enki</td>
<td>2</td>
<td>5</td>
<td>22 secs</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>get_admin_pages</td>
<td>0</td>
<td>Enki</td>
<td>2</td>
<td>1</td>
<td>22 secs</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>get_admin_pages_id</td>
<td>1</td>
<td>Enki</td>
<td>2</td>
<td>5</td>
<td>23 secs</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>get_admin_posts</td>
<td>0</td>
<td>Blog</td>
<td>2</td>
<td>11</td>
<td>21 secs</td>
<td>12</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>get_article_id</td>
<td>1</td>
<td>Blog</td>
<td>6</td>
<td>29</td>
<td>42 secs</td>
<td>16</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>liststudentcourses</td>
<td>2</td>
<td>Student</td>
<td>6</td>
<td>20</td>
<td>41 secs</td>
<td>24</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 13. Performance on synthetic commands

- **Simple Sequences (SS):** A sequence of different queries, without any conditional or loop statements. Each query does not reference any previously retrieved data.

- **Nested Conditionals (NC):** A series of nested conditional statements. Each except the innermost If statement has a nested If statement in the then branch. The innermost If statement has a query in the then branch. None of the queries reference previously retrieved data.

- **Unambiguous Long Reference Chains (UL):** Like (NC), but each query references data retrieved by the previous query when the data is nonempty.

- **Ambiguous Long Reference Chains (AL):** Like (UL), but each then block has an additional query before the nested If statement. This additional query retrieves a superset of the data that will be retrieved by the next query, which prints the retrieved data.

We expect the current Konure implementation to (1) scale well for (SS) and (NC) commands — the fact that the queries are independent makes it straightforward to translate path constraints to a small number of logical formulas, (2) scale well for (UL) commands, because disambiguation is unnecessary, (3) scale poorly for (AL) commands, because the number of disambiguation constraints grows rapidly as the length of the query reference chain increases, and (4) scale well for (AS) commands, because the reference chains are short.

For each class above, we built representative commands with varying code sizes. We then used Konure to infer each command. Figure 13 presents statistics from running Konure on these synthetic commands. For SS commands (Figure 13a),
the horizontal axis presents the number of queries in the command. For the remaining commands (Figures 13b-13e), the horizontal axis presents the number of conditionals in the command plus one. The left vertical axis presents the number of runs, solves, or lines of code. The lines Runs (executions of the command), Solves (invocations of Z3), and LoC (lines of code in the command) use this axis. The first right vertical axis presents the inference time in seconds. The line Time (wall-clock time for inference) uses this axis. The second right vertical axis presents the number of constraints that Konure sends to the SMT solver during inference. The lines PathCstr (constraints to enforce an execution path) and DisamCstr (constraints to disambiguate origin locations) use this axis. In Figure 13d, Konure ran out of memory after the version with five conditionals.

Discussion: Konure scales well for (SS), (NC), (UL), and (AS) commands, which is consistent with results in Section 4.1. Konure does not scale well for (AL) commands, where the major performance bottleneck is sending the solver disambiguation constraints (Section 3.4). We did not optimize Konure to generate a small number of disambiguation constraints, so the communication dominates the inference time.

After Z3 receives constraints, it solves them quickly. We anticipate that commands with ambiguous long reference chains will occur rarely in practice, as the structure of database tables typically supports the application functionality well enough to access the desired data by navigating through only several tables. The four commands from Table 1 with the longest inference times (get_projects_id, get_projects_id_users, get_channels, and get_channels_id_activities) all infer in feasible times. We therefore anticipate the inference algorithm will scale to handle real applications.

Since we expect ambiguous long reference chains to occur rarely, we did not optimize Konure for this case. If this issue becomes important in practice, a way to mitigate it would be to develop a solver that returns maximally distinct values. This solver would ensure that unrelated origin locations hold disjoint values.

Because Konure analyzes each command separately, it scales linearly with the number of commands. Therefore, it easily scales to handle applications with many commands, which is often the primary source of complexity.

5 Related Work

Active Learning: Active learning is a classical topic in machine learning [64]. Our approach is characterized by its extensive exploitation of structure present in the program inference task: (1) learning outcomes specified by a DSL, (2) hypotheses as sentential forms in the DSL, and (3) learning by resolving nonterminals in the current hypothesis.

Program Synthesis: The vast majority of program synthesis research works with a given set of input/output examples [9, 16, 29–32, 40, 47, 56, 57, 67, 73, 75, 78–80]. Because the examples typically underspecify the program behavior, there are often many programs that satisfy the examples. The synthesized program is therefore typically selected according to either the choices the solver makes [47] or a heuristic that ranks synthesized programs (for example, ranking shorter programs above longer programs) [29, 32, 40]. Konure, in contrast, uses active learning to choose inputs and database contents that eliminate uncertainty and obtain a model that completely captures the core application functionality.

SyGuS identifies a range of program synthesis problems for which it is productive to structure the search space as a DSL [9]. Unlike SyGuS, Konure deploys a top-down inference algorithm that progressively refines a working hypothesis represented as a sentential form of the DSL grammar. Unlike the vast majority of solver-driven synthesis algorithms (which require finite search spaces), Konure works effectively with an unbounded space of models.

LaSy works with a sequence of user-provided input/output pairs to iteratively generalize an overspecialized program [54]. Konure, in contrast, (1) automatically generates a sequence of inputs and database contents that uniquely identify the program within the DSL, (2) observes not just inputs and outputs, but also the traffic between the database and the application, and (3) uses a top-down approach that iteratively resolves DSL grammar nonterminals as opposed to a bottom-up approach that replaces overspecialized code fragments.

[15] presents a static technique that rewrites source code to optimize the execution of loops. Konure, in contrast, does not work with the source code and uses active learning over program executions to infer the program behavior.

To better evaluate the value of active learning in our context, we implemented a system that observes inputs, outputs, and database traffic generated during normal use to infer models of programs that access databases [65]. The results show that this approach often fails to infer the full functionality of the application because it often misses infrequent corner cases. Wrapping a standard CEGIS-style loop [67] around this system would require finite programs (whose input is bounded and terminate on all inputs after a bounded number of operations) as specifications. In contrast, Konure (1) uses active learning to find inputs, as opposed to asking the user for examples or specifications, (2) adopts a syntax-guided approach, as opposed to using a flat solver-based approach, (3) works with database-backed programs where the size of input data is unbounded, and (4) infers models within a countably infinite space of models defined by a DSL.

Model Inference: Our previous research produced an active learning technique for black-box inference of programs that manipulate key/value maps [62]. Konure, in contrast, also observes database traffic, works with a broader and more expressive class of applications, and deploys a top-down, syntax-guided inference algorithm (as opposed enumerating store/retrieve pairs as in [62]).
Brahma implements oracle-guided synthesis for loop-free programs that compute functions of finite-precision bit-vector inputs [48]. Brahma finitizes the synthesis problem by working with a finite set of components, with each component used exactly once in the synthesized model. Konure, in contrast, works with an infinite space of models.

Mimic traces memory accesses to synthesize a model of a traced function [44]. It uses a random generate-and-test search over a space of programs generated by code mutation operators. There is no guarantee that the generated model is correct or that the search will find a model if one exists.

Other related techniques include an active learning technique for learning commutativity specifications of data structures [36], a technique for learning program input grammars [14], a technique for learning points-to specifications [15], and a technique for learning models of the design patterns that Java computations implement [46]. Unlike Konure, all of these techniques focus on characterizing specific aspects of program behavior and do not aspire to capture the complete behavior of the application.

State Machine Model Learning: State machine learning algorithms [7, 10, 20, 22, 33, 39, 45, 52, 59, 69, 72] construct partial representations of program functionality in the form of finite automata with states and transition rules. State fuzzing tools [6, 28, 58] hypothesize state machines for programs. Network function state model extraction [77] uses program slicing and models the sliced partial programs as packet-processing automata. Konure, in contrast, infers complete application functionality (as opposed to a partial model of the application) and can support application regeneration.

Dynamic Analysis for Program Comprehension: There is a large body of research on dynamic analysis for program comprehension, but (due to complicated logic of Web technologies) relatively little of this research targets Web application servers [26]. WAFA [8] analyzes Web applications, focusing on interactions between Web components, using source code annotations. In contrast, Konure infers applications without analyzing, modifying, or requiring access to source code. Konure works for applications written in any language and can infer both Web and non-Web applications that interact with an external relational database.

DAVIS [53] visualizes the data-manipulation behavior of an execution of a data-intensive program. DAVIS detects loops whose body contains only one query. DiscoTect [81] summarizes the software architecture of a running object-oriented system as a state machine. They both analyze program behavior when processing certain user-specified inputs. In contrast, Konure actively explores the execution paths of the program by solving for inputs and database contents that enable it to infer the full application behavior.

Database Reverse Engineering/Reengineering: Database reverse engineering analyzes a program’s data access patterns, often to reconstruct implicit assumptions of the database schema [24, 27]. Konure infers programs that interact with databases (and not the structure of the database).

Database program reengineering often involves analyzing the source code to produce more efficient database queries [21, 25]. In contrast, Konure (1) does not require dynamic program instrumentation or static analysis, (2) does not require the program to be written in specific languages or patterns, and (3) regenerates a new executable program (instead of transforming database queries).

Input Generation for Discovering Defects: Concolic testing [19, 37, 38, 63] generates inputs that systematically explore all execution paths in the program. The goal is to find inputs that expose software defects. BuzzFuzz [35] generates inputs that target defects that occur because of coding oversights at the boundary between application and library code. DIODE [66] generates inputs that target integer overflow errors. All of these techniques target programs written in general-purpose languages such as C. Given the complexity and generality of computations as expressed in this form, completely exploring and characterizing application behavior is infeasible in this context. Our approach, in contrast, (1) works with applications whose behavior can be productively modeled with programs in our DSL and (2) infers a model that captures the complete functionality of the program.

6 Conclusion

Applications that read relational databases are pervasive in modern computing environments. We present new active learning techniques that automatically infer and regenerate these applications. Key aspects of these techniques include (1) the formulation of an inferrable DSL that supports the range of computational patterns that these applications exhibit and (2) the inference algorithm, which progressively synthesizes inputs and database contents that productively resolve uncertainty in the current working hypothesis. Results from our implementation highlight the ability of this approach to infer and regenerate applications that access relational databases.

Acknowledgments

We thank Jürgen Cito, Shivam Handa, Osbert Bastani, Cong Yan, Sara Achour, Deokhwan Kim, James Koppel, our shepherd Rahul Sharma, and the anonymous reviewers for their insightful and helpful comments. This research was supported by DARPA (Grant FA8650-15-C-7564).
Using Active Learning to Synthesize Models of Applications

PLDI ’19, June 22–26, 2019, Phoenix, AZ, USA

References

[34] X. Fu, X. Lu, B. Pei, S. Shen, K. Qian, and L. Tao. 2007. A Static Analysis Framework For Detecting SQL Injection Vulnerabilities


