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

ACM Reference Format:
Jiasi Shen and Martin C. Rinard. 2019. Using Active Learning to Synthesize Models of Applications That Access Databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3314221.3314591

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

ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314591

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.

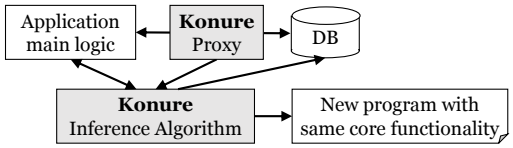


Figure 1. The Konure architecture, including a transparent proxy interposed between the application and the database to observe the generated database traffic.

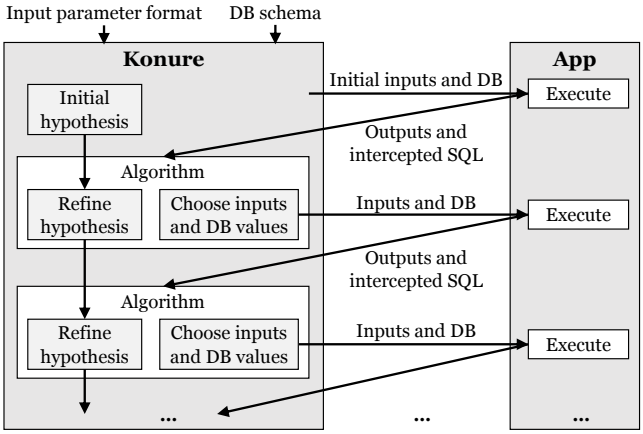


Figure 2. The Konure active learning algorithm iteratively refines its hypothesis to infer the application.

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.

2 Example

We next present an example that illustrates how Konure infers and regenerates a database-backed application. The example is a student registration system adapted from an application written by an independent evaluation team hired by an agency of the United States Government to evaluate techniques for detecting and nullifying SQL injection attacks. The application was written in Java and interacts with a MySQL database [76] via JDBC [60].

Command: The application implements the following command: "`liststudentcourses -s` *s* `-p` *p*", where the input parameter *s* denotes student ID and *p* denotes password. The application first checks whether the student with ID *s* has password *p* in the database. If so, the application displays the list of courses for which this student has registered, along with the teacher for each course.

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 inferrability 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

```
SELECT * FROM student WHERE id = '0'
```

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

```
q1: select student.id, student.password, student.
    firstname, student.lastname
    where student.id = s
```

(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

Prog	:=	ϵ | Seq | If | For
Seq	:=	Query Prog
If	:=	if Query then Prog else Prog
For	:=	for Query do Prog else Prog
Query	:=	$y \leftarrow$ select Col$^+$ where Expr ; print Orig*
Expr	:=	true | Expr $\wedge$ Expr | Col = Col | Col = Orig
Col	:=	$t.c$
Orig	:=	x | y.Col
		$x, y \in$ *Variable*, $t \in$ *Table*, $c \in$ *Column*

Figure 4. Grammar for the Konure DSL

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_1 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

```
SELECT * FROM student WHERE id = '5'
SELECT * FROM student WHERE id = '5' AND
    password = '6'
```

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

```
q1: select student.id, student.password, student.
    firstname, student.lastname
    where student.id = s
q2: select student.id, student.password, student.
    firstname, student.lastname
    where student.id = s ∧ student.password = p
```

(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

```
if y1 ← select student.id,student.password,student
    .firstname,student.lastname
    where student.id = s then P1 else P2
```

Figure 6. Hypothesis after resolving the topmost Prog nonterminal to an If statement.

```
SELECT * FROM student WHERE id = '1'
SELECT * FROM student WHERE id = '1' AND
    password = '2'
SELECT * FROM course c JOIN registration r ON r.
    course_id = c.id WHERE r.student_id = '1'
```

(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.

```
q1: select student.id, student.password, student.
    firstname, student.lastname
    where student.id = s
q2: select student.id, student.password, student.
    firstname, student.lastname
    where student.id = s ∧ student.password = p
q3: select course.id,course.name,course.
    course_number,course.size_limit,course.
    is_offered,course.teacher_id,registration.
    student_id,registration.course_id
    where registration.course_id = course.id ∧
        registration.student_id = s
```

(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 7. Third execution trace

```
if y1 ← select student.id,student.password,student
    .firstname,student.lastname
    where student.id = s then {
 if y2 ← select student.id,student.password,
     student.firstname,student.lastname
     where student.id = s ∧ student.password = p
 then P3 else P4 } else P2
```

Figure 8. Hypothesis after resolving P_1 (Figure 6).

```
SELECT * FROM student WHERE id = '3'
SELECT * FROM student WHERE id = '3' AND
    password = '4'
SELECT * FROM course c JOIN registration r ON r.
    course_id = c.id WHERE r.student_id = '3'
SELECT firstname, lastname FROM teacher WHERE id
     = '16'
SELECT count(*) FROM registration WHERE
    course_id = '12'
SELECT firstname, lastname FROM teacher WHERE id
     = '11'
SELECT count(*) FROM registration WHERE
    course_id = '7'
```

Figure 9. Concrete trace from an execution to resolve P_3 (Figure 8). The third query retrieves two rows. The final four queries are generated by a loop that iterates over the retrieved two rows.

```
if y1 ← select student.id, student.password, student
    .firstname, student.lastname
    where student.id = s then {
 if y2 ← select student.id, student.password,
     student.firstname, student.lastname
     where student.id = s ∧ student.password = p
 then {
  for y3 ← select course.id, course.name, course.
      course_number, course.size_limit, course.
      is_offered, course.teacher_id, registration.
      student_id, registration.course_id
      where registration.course_id = course.id ∧
          registration.student_id = s;
      print y3.course.id, y3.course.teacher_id
  do P5 else P6 } else P4 } else P2
```

Figure 10. Hypothesis after resolving P_3 (Figure 8).

the application iterates over all of the rows retrieved from the third query, and resolves P_3 to a For statement.

For this execution the application also produces the `id` and `teacher_id` columns from the retrieved rows of the `course` table as output. The updated hypothesis (Figure 10) therefore contains a Print statement that prints these values.

Regeneration: Konure proceeds as above, systematically targeting and resolving nonterminals in the hypothesis, until all of the nonterminals are resolved and it has inferred a model of the command. It can then regenerate the command, inserting security/safety checks as desired. Our current Konure implementation regenerates Python code using a standard SQL library to perform the database queries. This regeneration eliminates a seeded SQL injection attack vulnerability present in the original program.

Noisy Specifications: Because the active learning algorithm, guided by the DSL, tends to generate contexts that conform to common use cases, Konure can work productively with programs that contain obscure corner-case bugs not exercised during the inference [51, 61]. The SQL injection attack vulnerability present in the original Student Registration application but discarded in the regeneration is an example of just such an obscure corner case bug. We view such programs as *noisy specifications*. Given the known challenges developers face when attempting to deliver correct programs, we consider the ability of Konure to work successfully with such noisy specifications as a significant advantage of the overall approach.

Developer 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.

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 $P \in \text{Prog}$, $\widetilde{P}$ 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. $\mathcal{D}(P)$ 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:

$$\mathcal{K} = \{\widetilde{P} \mid P \in \text{Prog}, \mathcal{T}(\widetilde{P}) \cap \mathcal{R}(\widetilde{P}) = \emptyset, \mathcal{D}(\widetilde{P}) = \texttt{true}\}$$

The first restriction, $\mathcal{T}(\widetilde{P}) \cap \mathcal{R}(\widetilde{P}) = \emptyset$, 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, $\mathcal{D}(\widetilde{P}) = \texttt{true}$, states that the two branches of any If statement in $\widetilde{P}$ 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 $\widetilde{P}$, and determine if $\widetilde{P} \in \mathcal{K}$. 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 $\sigma = \langle \sigma_I, \sigma_D, \sigma_R \rangle \in \textit{Context}$ (Figure 11) contains value mappings for the input parameters ($\sigma_I \in \textit{Input}$), database contents ($\sigma_D \in \textit{Database}$), and results retrieved by database queries ($\sigma_R \in \textit{Result}$). The input context σ_I maps input parameter variables $x \in \textit{Variable}$ to concrete values. The database context σ_D maps database locations (identified by a table name, a row number, and a column name) to concrete values. The results context σ_R maps each query result variable $y \in \textit{Variable}$ 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 $\sigma \in \textit{Context}$ 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 $\sigma \in \textit{Context}$ produces the concrete trace $\sigma(P)$. Note that Konure does not access the source code of P when it executes $\boxed{P}$.

$$\sigma \in \mathit{Context} = \mathit{Input} \times \mathit{Database} \times \mathit{Result}$$
$$\sigma_I \in \mathit{Input} = \mathit{Variable} \to \mathit{Value}$$
$$\sigma_D \in \mathit{Database} = \mathit{Table} \to \mathbb{Z}_{>0} \to \mathit{Column} \to \mathit{Value}$$
$$\sigma_R \in \mathit{Result} = \mathit{Variable} \to \mathbb{Z}_{>0} \to \mathit{Table} \to \mathit{Column} \to \mathit{Value}$$
$$\mathit{Value} = \mathit{Int} \cup \mathit{String}$$

Figure 11. KONURE contexts.

CTrace	:=	CQuery* CData*; `print` CVal*
CQuery	:=	`SELECT` CCol+ `FROM` CJoin `WHERE` CExpr
CJoin	:=	t | CJoin `JOIN` t `ON` CCol = CCol
CExpr	:=	`true` | CExpr `AND` CExpr | CCol = CCol
		| CCol = CVal | CCol `IN` CVal*
CCol	:=	$t.c$
CVal	:=	i | s
CData	:=	CRow*
CRow	:=	(CCol CVal)+

$t \in \mathit{Table}, \quad c \in \mathit{Column}, \quad i \in \mathit{Int}, \quad s \in \mathit{String}$

(a) Concrete traces.

ATrace	:=	AQuery* r*; `print` AOrig*
AQuery	:=	$y \leftarrow$ `select` ACol+ `where` AExpr
AExpr	:=	`true` | AExpr $\wedge$ AExpr | ACol = ACol
		| ACol $\in$ AOrig+
ACol	:=	$t.c$
AOrig	:=	x | y.ACol

$x, y \in \mathit{Variable}, \quad t \in \mathit{Table}, \quad c \in \mathit{Column}, \quad r \in \mathbb{Z}_{\geq 0}$

(b) Abstract traces.

Figure 12. Grammars for concrete and abstract traces.

Definition 8. An *origin location* $O \in$ Orig in a program $P \in$ Prog is an occurrence of a variable x or a column y.Col in a query result y.

Definition 9. An *abstract trace* is the list of queries, along with their results, that KONURE generates from a concrete trace after replacing concrete values with their origin locations and replacing SQL syntax with the syntax of abstract traces (Figure 12b). An abstract trace contains abstract queries (AQuery*), row counts for each query (r*), and output origin locations (`print` AOrig*). The main modifications from a concrete trace are to replace each concrete value by its origin location and to summarize the retrieved data with row counts.

To infer the origin locations, KONURE maintains a context, which keeps track of the concrete values available at each origin location in the input and result components. One complication is the possibility that two distinct origin locations may hold the same concrete value in an execution. When such ambiguities occur, KONURE adopts a demand-driven approach to obtain an unambiguous origin location (Section 3.4). With the origin locations inferred, it is straightforward to rewrite the trace syntax as an abstract trace.

Definition 10. A *query-result pair* (Q, r) has a query $Q \in$ Query and an integer $r \in \mathbb{Z}_{\geq 0}$ that counts the number of rows retrieved by Q during execution. Converting an abstract trace into a list of query-result pairs is straightforward.

Definition 11. A *loop layout tree* for a program $P \in$ Prog is a tree that represents information about the execution of loops. 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 $\langle Q, r, \lambda \rangle$, 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 λ is a nonnegative integer that indicates the iteration index. If no such loop was found then λ = NotLoop. Each path from the root of the loop layout tree to a leaf generates a corresponding annotated trace.

Definition 13. A *path constraint* $W = (\langle Q_1, r_1, s_1 \rangle, \ldots, \langle Q_n, r_n, s_n \rangle)$, 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 $(= 0)$, (≥ 1), or (≥ 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 $\boxed{P}$, path constraint W, and context σ as parameters. It first invokes EXECUTE, which runs $\boxed{P}$ in context σ to obtain the flat list e of query-result pairs converted from the concrete trace that $\boxed{P}$ 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 $\boxed{P}$ and a context $\sigma = \langle \sigma_I, \sigma_D, \sigma_R \rangle \in \mathit{Context}$. It first populates the database with contents specified in σ_D and then executes $\boxed{P}$ with input parameters specified in σ_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: $\boxed{P}$ is the executable of a program $P \in \mathcal{K}$.
Output: Program equivalent to P.

```
1: procedure Infer(⌈P⌉)
2:     σ ← Database empty, input parameters distinct
3:     t ← GetTrace(⌈P⌉, Nil, σ)
4:     return InferProg(⌈P⌉, Nil, t)
5: end procedure
```

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

Input: $\boxed{P}$ is the executable of a program $P \in \mathcal{K}$.
Input: W is a path constraint.
Input: σ is a context that satisfies W.
Output: Annotated trace t, $t \sim W$, from executing $\boxed{P}$ with σ.

```
1: procedure GetTrace(⌈P⌉, W, σ)
2:     e ← Execute(⌈P⌉, σ)
3:     l ← DetectLoops(e)
4:     return MatchPath(l, W)
5: 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.

InferProg: Algorithm 3 implements the main Konure inference algorithm. This algorithm recursively explores all relevant paths through the program, resolving Prog nonterminals as they are (conceptually) encountered. Algorithm 3 takes as parameters the executable $\boxed{P}$ 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 Prog nonterminal to resolve. Konure therefore determines whether the query Q was generated by a Seq, If, or For statement, then recurses to infer the remaining parts of the program.

Konure makes this determination by examining three deduplicated annotated traces t_0, t_1, 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 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 Algorithm 3). Otherwise, it examines t_0 and t_1

Algorithm 3 Recursively infer a subprogram

Input: $\boxed{P}$ is the executable of a program $P \in \mathcal{K}$.
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: procedure InferProg(⌈P⌉, s1, s2)
2:     if s2 = Nil then return ϵ                          ▷ Prog := ϵ
3:     end if
4:     k ← The length of s1
5:     Q ← The first query in s2
6:     for i = 0, 1, 2 do
7:         W_i ← MakePathConstraint(s1, Q, i)
8:         (f_i, t_i) ← SolveAndGetTrace(⌈P⌉, W_i)
9:         if f_i then                                    ▷ Satisfiable
10:            t_{i,1} ← t_i[1, ..., (k + 1)]             ▷ New trace prefix
11:            t_{i,2} ← t_i[(k + 2), ...]                ▷ New trace suffix
12:        end if
13:    end for
14:    if f_2 and found loop on the last query in t_{2,1} then
15:        b_t ← InferProg(⌈P⌉, t_{2,1}, t_{2,2})
16:        if f_0 then b_f ← InferProg(⌈P⌉, t_{0,1}, t_{0,2})
17:        else b_f ← ϵ
18:        end if
19:        return "for Q do b_t else b_f"                 ▷ Prog := For
20:    else if f_0 and f_1 and ((t_{0,2} = Nil and t_{1,2} ≠ Nil) or
                                (t_{0,2} ≠ Nil and t_{1,2} = Nil) or
                                the first queries in t_{0,2} and t_{1,2}
                                have different skeletons) then
21:        b_t ← InferProg(⌈P⌉, t_{1,1}, t_{1,2})
22:        b_f ← InferProg(⌈P⌉, t_{0,1}, t_{0,2})
23:        return "if Q then b_t else b_f"                ▷ Prog := If
24:    else
25:        if f_0 then b ← InferProg(⌈P⌉, t_{0,1}, t_{0,2})
26:        else b ← InferProg(⌈P⌉, t_{1,1}, t_{1,2})
27:        end if
28:        return "Q b"                                   ▷ Prog := Seq
29:    end if
30: end procedure
```

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

Input: $\boxed{P}$ is the executable of a program $P \in \mathcal{K}$.
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: procedure SolveAndGetTrace(⌈P⌉, W)
2:     σ ← Solve(W)
3:     if σ = Unsat then
4:         return false, Nil
5:     else
6:         t ← GetTrace(⌈P⌉, W, σ)
7:         return true, t
8:     end if
9: 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 σ_0, σ_1, and σ_2 that cause $\boxed{P}$ 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_1, 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_1, 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 $Q \in \text{Query}$ and a context $\sigma = \langle \sigma_I, \sigma_D, \sigma_R \rangle \in \mathit{Context}$, $\sigma(Q)$ denotes the result from evaluating Q in σ. Evaluating Q involves replacing origin locations in Q with their values in σ_I and σ_R, rewriting the query in SQL syntax, then performing the SQL query on σ_D. The query result contains an ordered list of rows. $|\sigma(Q)|$ denotes the number of rows in $\sigma(Q)$. $Q.y$ denotes the variable that stores the retrieved data. $\sigma[Q.y \mapsto z]$ denotes the new context after updating σ_R to map $Q.y$ to z.

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

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

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 σ_n the context after updating σ with W.

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

Like many database test data generation approaches [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 $O \in \text{Orig}$ in a program $P \in \text{Prog}$ 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 σ 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 $\sigma \in \mathit{Context}$ that satisfies W, O_1, O_2 hold the same values in the context after updating σ with W.

Definition 18. For a program $P \in \text{Prog}$ and a context $\sigma \in \mathit{Context}$, $\sigma \vdash P \Downarrow_{\text{exec}} e$ denotes evaluating P in σ to obtain a list of query-result pairs e. $\sigma \vdash P \Downarrow_{\text{loops}} l$ denotes evaluating P in σ to obtain a loop layout tree l.

Definition 19. For programs $P, P' \in \text{Prog}$ and annotated trace t, we use the notation $P \xrightarrow{t} P'$ 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 Solve(W) returns a context $\sigma \in \mathit{Context}$ 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 $\sigma \in \mathit{Context}$, if $\sigma \vdash P \Downarrow_{\text{exec}} e$, Execute($\boxed{P}, \sigma$) = 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 $\sigma \in \mathit{Context}$, if $\sigma \vdash P \Downarrow_{\text{exec}} e$ and $\sigma \vdash P \Downarrow_{\text{loops}} l$ then DetectLoops(e) = l.

Proof. By induction on the derivation of P. □

Theorem 2 (Core Recursion)**.** For any programs $P \in \mathcal{K}$ and $P' \in \text{Prog}$ and annotated traces t_1, t_2, if $P \xrightarrow{t_1} P'$ and $P' \xrightarrow{t_2} \epsilon$ then InferProg($\boxed{P}, t_1, t_2$) 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 $\boxed{P}$ 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 $P \in \mathcal{K}$, Infer($\boxed{P}$) 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 $P \xrightarrow{t} \epsilon$. The rest of the proof follows from Theorem 2. □

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

Each recursive call to InferProg constructs a subprogram for $P \in \mathcal{K}$. 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 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 Execute($\boxed{P}, \sigma$). 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 all articles and a command that retrieves a specific article and its associated comments.
- **Student Registration:** The student registration application discussed in Section 2.

The Fulcrum, Enki, and Blog servers receive HTTP requests, interact with the database accordingly, and respond the client with an HTML page that contains the data retrieved. The Kandan server receives HTTP requests, interacts with the database accordingly, and responds with JSON objects that contain data retrieved and HTML templates to display the JSON data. For these applications, the Konure prototype works with the retrieved database results after they are automatically extracted from the surrounding HTML/JSON code. Student Registration implements a command-line interface that receives text commands, interacts with the database accordingly, and responds with text output.

Based on our understanding and use of the applications, we identified data retrieval commands that these applications execute as part of their standard functionality. In general, these commands step through tables, typically using results from earlier look-ups to access the correct data in current tables. As a command traverses tables, it collects data to return to the user. Fulcrum uses five database tables, Kandan uses four database tables, Enki uses five database tables, Blog uses two database tables, and Student Registration uses five database tables. For Fulcrum, we identified eight of 14 data retrieval commands as potential inference candidates. For Kandan, we identified six of 11, for Enki, four of ten, for Blog, two of two, and for Student Registration, one of one.

We built virtual machines for executing these applications, then configured our Konure prototype to operate properly in this context. Specifically, the Rails framework stores password hashes in the database. Based on the Rails configuration, the Rails framework uses these hashes to perform a password check at the start of specified commands. We configured our Konure prototype to generate databases and parameters that, during inference, always pass the password check. We also support the insertion of boilerplate password checking code into the regenerated code for specified commands. We anticipate that the automated introduction of such boilerplate code will be standard in many usage contexts. We then used Konure to infer and regenerate the commands. The source code for the regenerated commands is available [5].

Table 1 presents statistics from running the Konure prototype on the commands. The first column (**Command**) presents the name of the command. The second (**Params**) presents the number of input parameters for the command. The third (**App**) presents the name of the application.

The next column (**Runs**) presents the number of executions that Konure used to infer the command. Each execution involves a set of generated input values presented to the application working with generated database contents. All commands require fewer than 30 executions to obtain a model for the command as expressed in the Konure DSL. The next column (**Solves**) 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.

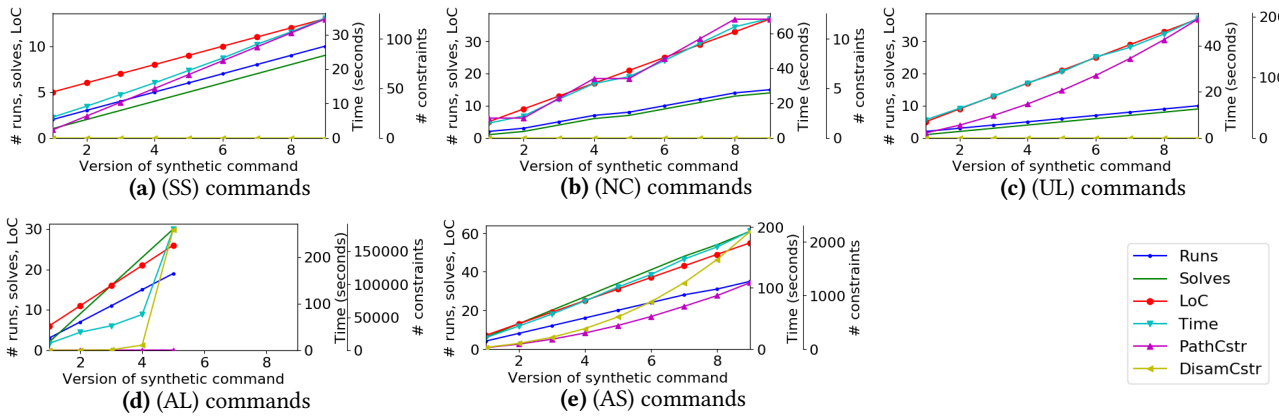
We recruited a software engineer with three years of experience working with Ruby on Rails applications to evaluate the Konure inference and regeneration by comparing the original Ruby on Rails and regenerated Python versions of each command. This manual comparison indicated that the inferred and regenerated 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

Command	Params	App	Runs	Solves	Time	LoC	SQL	If	For	Output
`get_home`	1	Fulcrum	5	43	8 mins	21	5	1	0	9
`get_projects`	1	Fulcrum	5	43	8 mins	21	5	1	0	9
`get_projects_id`	2	Fulcrum	12	124	29 mins	25	8	2	0	8
`get_projects_id_stories`	2	Fulcrum	11	42	7 mins	31	8	3	0	11
`get_projects_id_stories_id`	3	Fulcrum	12	50	8 mins	31	9	3	0	11
`get_projects_id_stories_id_notes`	3	Fulcrum	11	41	8 mins	24	9	3	0	4
`get_projects_id_stories_id_notes_id`	4	Fulcrum	13	46	10 mins	28	10	4	0	4
`get_projects_id_users`	2	Fulcrum	12	124	30 mins	25	8	2	0	8
`get_channels`	1	Kandan	21	125	105 mins	63	16	4	2	27
`get_channels_id_activities`	2	Kandan	23	242	39 mins	49	16	6	0	13
`get_channels_id_activities_id`	3	Kandan	14	18	7 mins	25	11	3	0	3
`get_me`	1	Kandan	11	139	6 mins	44	8	3	0	25
`get_users`	1	Kandan	15	236	9 mins	67	11	3	0	45
`get_users_id`	2	Kandan	11	139	6 mins	44	8	3	0	25
`get_admin_comments_id`	1	Enki	2	5	22 secs	10	1	0	0	5
`get_admin_pages`	0	Enki	2	1	22 secs	13	2	1	0	4
`get_admin_pages_id`	1	Enki	2	5	23 secs	9	1	0	0	4
`get_admin_posts`	0	Enki	3	2	33 secs	16	3	1	1	3
`get_articles`	0	Blog	2	11	21 secs	12	2	0	0	6
`get_article_id`	1	Blog	6	29	42 secs	16	3	1	0	6
`liststudentcourses`	2	Student	6	20	41 secs	24	5	3	1	3



(a) (SS) commands **(b)** (NC) commands **(c)** (UL) commands

(d) (AL) commands **(e)** (AS) commands

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.
- **Ambiguous Short Reference Chains (AS):** Like (NC), 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.

[13] 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).

References

[1] 2018. Enki. https://github.com/xaviershay/enki.

[2] 2018. Fulcrum. https://github.com/fulcrum-agile/fulcrum.

[3] 2018. Getting Started with Rails. http://guides.rubyonrails.org/getting_started.html.

[4] 2018. Kandan – Modern Open Source Chat. https://github.com/kandanapp/kandan.

[5] 2019. PLDI 2019 Konure Code. http://people.csail.mit.edu/jiasi/pldi2019.code/ and https://people.csail.mit.edu/rinard/paper/pldi2019.code/.

[6] F. Aarts, J. De Ruiter, and E. Poll. 2013. Formal Models of Bank Cards for Free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 461–468. https://doi.org/10.1109/ICSTW.2013.60

[7] Fides Aarts and Frits Vaandrager. 2010. *Learning I/O Automata*. Springer Berlin Heidelberg, Berlin, Heidelberg, 71–85. https://doi.org/10.1007/978-3-642-15375-4_6

[8] M. H. Alalfi, J. R. Cordy, and T. R. Dean. 2009. WAFA: Fine-grained dynamic analysis of web applications. In *2009 11th IEEE International Symposium on Web Systems Evolution*. 141–150. https://doi.org/10.1109/WSE.2009.5631226

[9] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 1–8.

[10] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (Nov. 1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[11] Dana Angluin and Carl H. Smith. 1983. Inductive Inference: Theory and Methods. *ACM Comput. Surv.* 15, 3 (Sept. 1983), 237–269. https://doi.org/10.1145/356914.356918

[12] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2007. CANDID: Preventing Sql Injection Attacks Using Dynamic Candidate Evaluations *(CCS '07)*. 13. https://doi.org/10.1145/1315245.1315249

[13] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From Relational Verification to SIMD Loop Synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 123–134. https://doi.org/10.1145/2442516.2442529

[14] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 95–110.

[15] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active Learning of Points-to Specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 678–692. https://doi.org/10.1145/3192366.3192383

[16] Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2015. Recursive Games for Compositional Program Synthesis. In *Verified Software: Theories, Tools, and Experiments - 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers*. 19–39.

[17] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2010. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. *ACM Trans. Inf. Syst. Secur.* 13, 2, Article 14 (March 2010), 39 pages. https://doi.org/10.1145/1698750.1698754

[18] Aaron R Bradley and Zohar Manna. 2007. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media.

[19] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, NY, USA, 322–335. https://doi.org/10.1145/1180405.1180445

[20] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. 2016. Active learning for extended finite state machines. *Formal Aspects of Computing* 28, 2 (2016), 233–263. https://doi.org/10.1007/s00165-016-0355-5

[21] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2491956.2462180

[22] T. S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Softw. Eng.* 4, 3 (May 1978), 178–187. https://doi.org/10.1109/TSE.1978.231496

[23] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

[24] Anthony Cleve, Nesrine Noughi, and Jean-Luc Hainaut. 2013. Dynamic program analysis for database reverse engineering. In *Generative and Transformational Techniques in Software Engineering IV*. Springer, 297–321.

[25] Yossi Cohen and Yishai A. Feldman. 2003. Automatic High-quality Reengineering of Database Programs by Abstraction, Transformation and Reimplementation. *ACM Trans. Softw. Eng. Methodol.* 12, 3 (July 2003), 285–316. https://doi.org/10.1145/958961.958962

[26] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. 2009. A Systematic Survey of Program Comprehension Through Dynamic Analysis. *IEEE Trans. Softw. Eng.* 35, 5 (Sept. 2009), 684–702. https://doi.org/10.1109/TSE.2009.28

[27] Kathi Hogshead Davis and Peter H. Aiken. 2000. Data Reverse Engineering: A Historical Survey. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00) (WCRE '00)*. IEEE Computer Society, Washington, DC, USA, 70–.

[28] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 193–206.

[29] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. 2016. Sampling for Bayesian Program Learning. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. 1289–1297.

[30] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 420–435.

[31] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 422–436.

[32] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 229–239.

[33] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. *Combining Model Learning and Model Checking to Analyze TCP Implementations*. Springer International Publishing, Cham, 454–471. https://doi.org/10.1007/978-3-319-41540-6_25

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

(COMPSAC 2007). https://doi.org/10.1109/COMPSAC.2007.43

[35] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[36] Timon Gehr, Dimitar Dimitrov, and Martin T. Vechev. 2015. Learning Commutativity Specifications. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 307–323.

[37] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[38] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[39] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. 2010. Learning of Event-recording Automata. *Theor. Comput. Sci.* 411, 47 (Oct. 2010), 4029–4054. https://doi.org/10.1016/j.tcs.2010.07.008

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

[41] B. P. Gupta, D. Vira, and S. Sudarshan. 2010. X-data: Generating test data for killing SQL mutants. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 876–879. https://doi.org/10.1109/ICDE.2010.5447862

[42] Raju Halder and Agostino Cortesi. 2010. Obfuscation-based Analysis of SQL Injection Attacks *(ISCC '10)*. 931–938. https://doi.org/10.1109/ISCC.2010.5546750

[43] William G. J. Halfond and Alessandro Orso. 2005. AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks *(ASE '05)*. 174–183. https://doi.org/10.1145/1101908.1101935

[44] Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 710–720.

[45] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. *The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning*. Springer International Publishing, Cham, 307–322. https://doi.org/10.1007/978-3-319-11164-3_26

[46] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. 2016. Synthesizing framework models for symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 156–167.

[47] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: sketching for Java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 934–937.

[48] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

[49] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper) *(SP '06)*. 6. https://doi.org/10.1109/SP.2006.29

[50] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis *(SSYM'05)*. 18–18.

[51] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. 2012. Automatic Input Rectification. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 80–90. http://dl.acm.org/citation.cfm?id=2337223.2337233

[52] Edward F Moore. 1956. Gedanken-experiments on sequential machines. *Automata studies* 34 (1956), 129–153.

[53] Nesrine Noughi, Marco Mori, Loup Meurice, and Anthony Cleve. 2014. Understanding the Database Manipulation Behavior of Programs. In *Proceedings of the 22Nd International Conference on Program Comprehension (ICPC 2014)*. ACM, New York, NY, USA, 64–67. https://doi.org/10.1145/2597008.2597790

[54] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven Synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 408–418. https://doi.org/10.1145/2594291.2594297

[55] Jeff Perkins, Jordan Eikenberry, Alessandro Coglio, Daniel Willenson, Stelios Sidiroglou-Douskos, and Martin Rinard. 2016. AutoRand: Automatic Keyword Randomization to Prevent Injection Attacks. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721 (DIMVA 2016)*. Springer-Verlag New York, Inc., New York, NY, USA, 37–57. https://doi.org/10.1007/978-3-319-40667-1_3

[56] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 522–538.

[57] Yewen Pu, Zachery Miranda, Armando Solar-Lezama, and Leslie Kaelbling. 2018. Selecting Representative Examples for Program Synthesis. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 80. PMLR, 4161–4170. http://proceedings.mlr.press/v80/pu18b.html

[58] Arjun Radhakrishna, Nicholas V. Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Černý. 2018. DroidStar: Callback Typestates for Android Classes. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1160–1170. https://doi.org/10.1145/3180155.3180232

[59] Harald Raffelt, Bernhard Steffen, and Therese Berg. 2005. LearnLib: A Library for Automata Learning and Experimentation. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '05)*. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/1081180.1081189

[60] George Reese. 2000. *Database Programming with JDBC and JAVA*. O'Reilly Media, Inc.

[61] Martin C. Rinard. 2007. Living in the comfort zone. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. 611–622.

[62] Martin C. Rinard, Jiasi Shen, and Varun Mangalick. 2018. Active Learning for Inference and Regeneration of Computer Programs That Store and Retrieve Data. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2018)*. ACM, New York, NY, USA, 12–28. https://doi.org/10.1145/3276954.3276959

[63] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[64] Burr Settles. 2009. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison.

[65] Jiasi Shen and Martin Rinard. 2018. *Using Dynamic Monitoring to Synthesize Models of Applications That Access Databases*. Technical Report. http://hdl.handle.net/1721.1/118184

[66] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. 2015. Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 473–486. https://doi.org/10.1145/2694344.2694389

[67] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[68] H. Tanno, X. Zhang, T. Hoshino, and K. Sen. 2015. TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 717–720. https://doi.org/10.1109/ICSE.2015.231

[69] Frits Vaandrager. 2017. Model Learning. *Commun. ACM* 60, 2 (Jan. 2017), 86–95. https://doi.org/10.1145/2967606

[70] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. 2009. Symbolic Query Exploration. In *Formal Methods and Software Engineering*, Karin Breitman and Ana Cavalcanti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–68.

[71] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL Query Explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 425–446.

[72] Michele Volpato and Jan Tretmans. 2015. Approximate Active Learning of Nondeterministic Input Output Transition Systems. *Electronic Communications of the EASST* 72 (2015).

[73] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

[74] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2018. Speeding Up Symbolic Reasoning for Relational Queries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 157 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276527

[75] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *PACMPL* 2, POPL (2018), 63:1–63:30.

[76] Michael Widenius and Davis Axmark. 2002. *Mysql Reference Manual* (1st ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[77] Wenfei Wu, Ying Zhang, and Sujata Banerjee. 2016. Automatic Synthesis of NF Models by Program Analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*. ACM, New York, NY, USA, 29–35. https://doi.org/10.1145/3005745.3005754

[78] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 508–521.

[79] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-example. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 580–593. https://doi.org/10.1145/3187009.3177735

[80] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 63 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133887

[81] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. 2004. DiscoTect: A System for Discovering Architectures from Running Systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 470–479.