Using Active Learning to Synthesize Models of Applications That Access Databases

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

We present a new technique that uses active learning to infer models of applications that manipulate relational databases. This technique comprises a domain-specific language for modeling applications that access databases (each model is a program in this language) and an associated inference algorithm that infers models of applications whose behavior can be expressed in this language. The inference algorithm generates test inputs and database configurations, runs the application, then observes the resulting database traffic and outputs to progressively refine its current model hypothesis. The end result is a model that completely captures the behavior of the application. Because the technique works only with the externally observable inputs, outputs, and databases, 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).

We also present a technique for automatically regenerating an implementation from the inferred model. The regenerator can produce a translated implementation in a different language and systematically include relevant security and error checks.

Additional Key Words and Phrases: Active learning, synthesis, reverse engineering, black box, path constraints

1 INTRODUCTION

Applications that access databases are ubiquitous in computing systems. Such applications typically 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 or modify 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 [Bandhakavi et al. 2007; Bisht et al. 2010; Fu et al. 2007; Halder and Cortesi 2010; Halfond and Orso 2005; Jovanovic et al. 2006; Livshits and Lam 2005; Perkins et al. 2016].

1.1 Konure

We present Konure, a new system that interacts with the application and its database to infer a model of the application behavior. The inference algorithm systematically constructs test database configurations and input commands, runs the application with the test database configurations and inputs, then observes the resulting outputs and database traffic to iteratively construct the model.

To make the inference problem tractable, Konure works with a domain-specific language that (1) captures common application behavior and (2) supports a hierarchical inference algorithm that progressively explores the behavior of the application to infer a model of application behavior. The inference algorithm operates in a top-down manner. It maintains a current hypothesis as a sentential form of the grammar that defines the domain-specific language. At each step it selects a nonterminal in this sentential form, constructs inputs and database configurations 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 on this configured database with the

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

constructed inputs, then observes the resulting database traffic and outputs to refine the hypothesis by applying the inferred production to the nonterminal.

If the application conforms to one of the models defined by the domain-specific language, then the algorithm is guaranteed to (1) terminate and (2) produce an inferred model that correctly captures the complete semantics of the application. Because Konure interacts with the application only via its input, output, and observed interactions with databases that Konure configures, it can infer and regenerate applications written in any language or in any coding style or methodology.

Because the model captures the complete semantics of the application, it can help developers explore and better understand the behavior of the application. Konure can also *regenerate* the application, translating the application into a potentially different language and systematically applying coding patterns and additional checks that are known to be safe. Potential benefits include successfully reverse engineering application functionality, supporting application migration to different languages or computing platforms, and the elimination of security vulnerabilities.

1.2 Experimental Results

We present case studies applying Konure to three applications: Kandan Chat Room [kan 2018], Blog [rai 2018], 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.

1.3 Previous Work In Program Synthesis

Program synthesis is currently an active research area [Alur et al. 2013; Beyene et al. 2015; Ellis et al. 2016; Feng et al. 2018, 2017; Feser et al. 2015; Gulwani et al. 2017; Jeon et al. 2015; Polikarpova et al. 2016; Wang et al. 2018; Yaghmazadeh et al. 2016]. The vast majority of this research works with a given set of input/output examples to synthesize a program that satisfies the given examples. Because the examples typically underspecify the program behavior, there are typically many programs that satisfy the input/output examples. The synthesized program is therefore typically selected according either to the choices the solver makes [Jeon et al. 2015] or according to a heuristic that ranks synthesized programs (for example, ranking shorter programs above longer programs) [Ellis et al. 2016; Feser et al. 2015; Gulwani et al. 2017].

Konure, in contrast, uses active learning to reverse engineer an existing application (in effect using the application itself as a specification). Because it is not constrained by a given set of input/output pairs, it can select the inputs and database configurations to purposefully refine its current application behavior hypothesis to eliminate uncertainty and synthesize a model that completely captures the behavior of the application.

1.4 Previous Work In Inferring Models for Programs

We identify oracle-guided synthesis as implemented in Bramha [Jha et al. 2010] as the closest previous research. Like Konure, Brahma interacts with a program to infer a model that completely captures the behavior of the program. Konure deploys a top-down, syntax-guided inference algorithm to infer models within a countably infinite space of models defined by a domain-specific language. Brahma, on the other hand, finitizes the synthesis problem by working with a finite set of components, with each component in the set used exactly once in the synthesized model. Konure maintains a sentential form that captures all remaining possible models and refines the hypothesis by generating inputs and database configurations that enable Konure to determine which production to apply to the current nonterminal in the sentential form. Brahma, in contrast, adopts a flat, solver-based approach that repeatedly 1) generates two programs that both satisfy the current set of input/output pairs, 2) generates a new input that distinguishes the two programs, 3) queries an oracle to find the correct output for the new input, and 4) adds the resulting input/output pair to the current set of input/output pairs. Brahma terminates when there is only one program that satisfies the set of input/output pairs. These differences reflect the different characteristics of the target programs to infer: Konure infers models for an unbounded space of programs that may contain loops that iterate over the results of database queries; Brahma focuses on loop-free programs that compute functions of finite-precision bit-vector inputs, with the input-output behavior of the components defined by logical formulas.

Mimic [Heule et al. 2015] traces the memory accesses of an opaque function to synthesize a model of the traced function. It uses a random generate-and-test search over a space of programs generated by code mutation operators, with a carefully designed fitness function measuring the degree to which the current model matches the observed memory traces. Input generation heuristics are used to find inputs that work well with the mutation operators and fitness function to find suitable code models. There is no guarantee that the generated model is correct or that the search will find a model if one exists. Mimic was applied to infer models for the Java Arrays.prototype computations, successfully inferring models for 12 of these computations. Konure targets a different class of computations, which enables it to deploy an algorithm that is guaranteed to infer a model if the application conforms to one of the models defined by the domain-specific language.

[Gehr et al. 2015] present an active learning technique for learning commutativity specifications of data structures. [Bastani et al. 2017c] present a technique for learning program input grammars. [Bastani et al. 2017b] present a technique for learning points-to specifications. [Jeon et al. 2016] present a technique for learning models of the design patterns that Java computations implement. 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.

1.5 Contributions

This paper makes the following contributions:

- **Inference Algorithm:** It presents a new algorithm for inferring the behavior of database-backed applications. Using active learning, the algorithm repeatedly constructs test database configurations and input commands, configures the database, runs the application on the constructed inputs, and observes the resulting outputs and database interactions to build up a model of the application behavior. The algorithm exploits the formulation of the model space as a domain-specific language to represent its current hypothesis as a sentential form in the underlying grammar of the language. It selects database configurations and inputs to structure the search as the repeated application of productions applied to nonterminals in the sentential form. This approach enables Konure to work effectively with unbounded model spaces and infer a model that captures the complete semantics of the application.
- **Computational Patterns:** It presents a domain-specific language for capturing specific computational patterns typically implemented by database-backed applications. The inference algorithm and domain-specific language are designed together to enable an effective active learning algorithm that leverages the structure of the domain-specific language to learn models that completely capture application behavior.
- **Experimental Results:** We present experimental results using Konure to infer and regenerate applications written in Ruby on Rails and Java. The results highlight the ability of our techniques to infer and regenerate robust, safe Python implementations of applications originally coded in other languages.

$$\begin{array}{lcl}
\text{Prog} & := & in^{*}\ \text{Block} \\
\text{Block} & := & \epsilon \mid \text{Query Block} \mid \text{If} \mid \text{For} \\
\text{If} & := & \texttt{if}\ \text{Query}\ \texttt{then}\ \text{Block}\ \texttt{else}\ \text{Block} \\
\text{For} & := & \texttt{for}\ \text{Query}\ \texttt{do}\ \text{Block}\ \texttt{else}\ \text{Block} \\
\text{Query} & := & d \leftarrow \texttt{select}\ (\text{Col}\ out)^{*}\ \texttt{where}\ \text{Expr} \\
\text{Expr} & := & \text{True} \mid \text{Col} = \text{Col} \mid \text{Col} \in d\ \text{Col} \mid \text{Col} = in \mid \text{Expr} \wedge \text{Expr} \\
\text{Col} & := & t.c
\end{array}$$

$$in, d \in \mathit{Variable}, \quad out \in \{\text{True}, \text{False}\}, \quad t \in \mathit{Table}, \quad c \in \mathit{Column}$$

Fig. 1. Abstract syntax for inferrable programs (the Konure DSL)

2 EXAMPLE

We next present an example that illustrates how Konure infers and regenerates database-backed applications. The example is a student registration system that allows student users to check current registration. The application was written in Java and interacts with a MySQL database [Widenius and Axmark 2002] via JDBC [Reese 2000].

Command Interface: The student registration application provides the following command-line interface: "`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 Schema: The application interacts with a MySQL database that contains the following tables and columns. (1) The *student* table, which contains student ID (primary key), first name, last name, and password. (2) The *teacher* table, which contains teacher ID (primary key), first name, and last name. (3) The *course* table, which contains course ID (primary key), name, course number, teacher ID (foreign key referencing the *teacher* table), etc. (4) The *registration* table, which contains student ID (foreign key referencing the *student* table) and course ID (foreign key referencing the *course* table).

2.1 Konure Domain-Specific Language

Konure infers application functionality that can be expressed with the Konure domain-specific language (DSL).[1] This DSL models the computation of the application component that handles the main logic of data interaction and supports a wide range of applications that mainly display data according to inputs.

Figure 1 presents the abstract syntax. The data interaction component (Prog) starts with a list of inputs, in^{*}, and has a block of code (Block). The block of code contains a list of database queries along with two types of control logic: conditional statements (If) or loops (For).

Each database query (Query) performs an SQL `select` operation to retrieve all data that satisfy an expression (Expr). The query expression (Expr) may contain a conjunction of clauses. Each clause may specify that (1) a column must have equal value as another column (Col = Col), (2) a column must have value that belongs to certain results retrieved earlier (Col $\in d$ Col), (3) a column must have equal value as an input parameter (Col = *in*). Each query may select certain columns (Col) from the retrieved data and may specify whether the data for each column is included in the final output (*out*). The query may also implement an SQL join operation by selecting from multiple tables. The retrieved data is stored in a variable *d* for use in later parts of the program.

[1]The application itself can be implemented in any language.

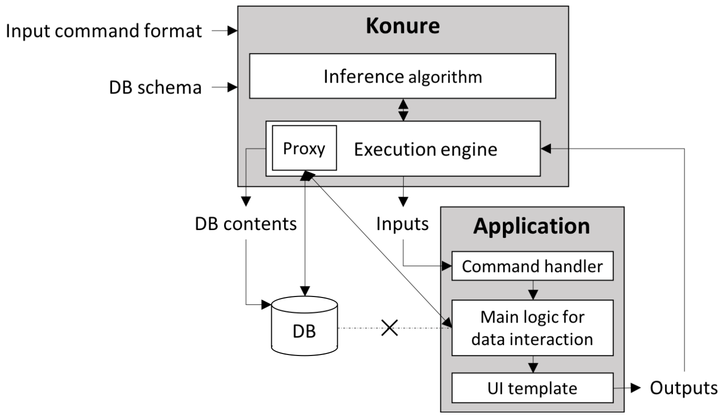

Fig. 2. Konure architecture

A conditional statement (If) is performed on the data retrieved from the previous query. When the query retrieves nonempty data, the program enters the `then` branch. When the query returns empty, the program enters the `else` branch. More sophisticated logical checks, such as conjunctions, may be implemented in the query expressions (Expr). The code in the two branches (Block) must begin with different queries.

A loop (For) iterates over data retrieved from the previous query. When the query retrieves nonempty data, the program enters the do block and repeats this loop body once for each retrieved row. When the query returns empty, the program enters the `else` branch. The loop body must not be empty, and its first query must use the data that this loop iterates over. This computation model captures a wide range of database-backed applications and allows Konure to infer the program functionality efficiently.

2.2 Inference

Figure 2 outlines the process when Konure infers an application. Konure works with applications that process sequences of commands. [2] For each command, the application first enters a command handler which preprocesses the inputs. The application then enters the main logic that decides what interactions to execute against the database. After performing SQL queries and retrieving data, the application uses the data to render an output which is then displayed to the user.

Konure starts with a specification of the application's input command interface and database schema. The inference algorithm chooses input values and a database configuration. It then populates the database with the specified contents and executes the application with the specified input command and parameters. When the application runs, it interacts with the database, transparently through the Konure proxy. After several database interactions, the application renders an output using the retrieved data.

The Konure proxy observes the database traffic and collects the SQL queries that the application sends to the database, along with the data that each query retrieves. We call these SQL queries a *concrete trace* (more discussion later in Section 3.1.1). Recall that the Konure DSL (Figure 1) allows each query to use values copied from results of an earlier query or from inputs. Konure abstracts away these observed concrete values and replaces them with the *source locations* of each value

[2] Command interfaces include the command-line interface, HTTP requests, and graphical interfaces that allow users to specify command parameters and execute commands.

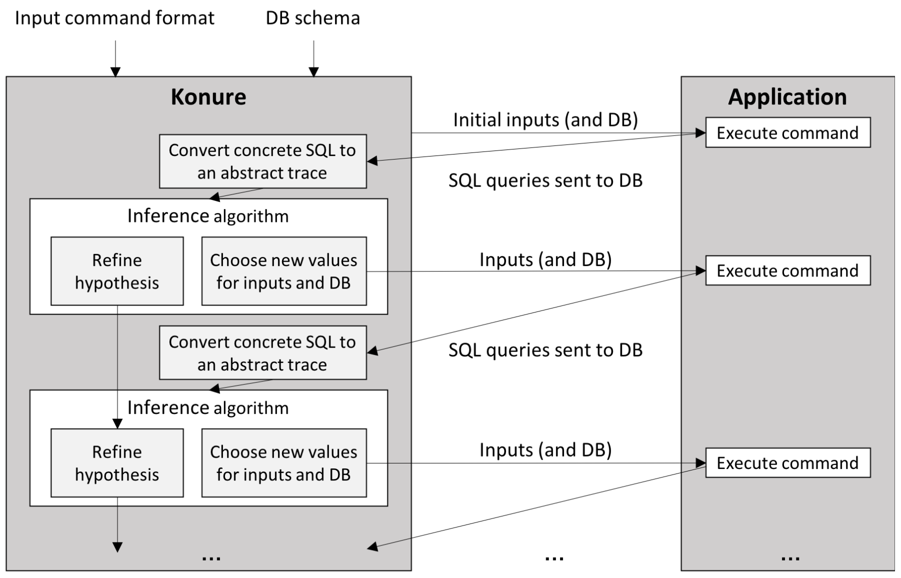

Fig. 3. Konure communicates with the application multiple times to progressively refine the hypothesis about the application's functionality

(Section 3.1.2). The source locations describe where a concrete value may be copied from, in terms of input parameters or columns from results of earlier queries. Konure determines potential source locations by matching equal values. Konure resolves ambiguity by carefully choosing values for the inputs and the database. Besides rewriting the queries, Konure also rewrites the application outputs by replacing concrete values with their source locations. We call this rewritten trace an *abstract trace* of the execution (Section 3.1.3). The inference algorithm then analyzes the abstract trace to update its hypothesis of the application's functionality. The process repeats until the algorithm has identified a model of application functionality.

Figure 3 summarizes the communications between Konure and the application. The SQL queries in this figure are the concrete trace that Konure captures by the proxy. Konure executes the application multiple times using different inputs and database contents (Section 3.1.4) until Konure terminates with a hypothesis, in the form of a model in the DSL, that does not have any unexplored hidden path (Section 3.3). Here each unexplored hidden path corresponds to a nonterminal in the DSL grammar that Konure will resolve by applying an appropriate production.

A Simple Execution Trace: When inferring the student registration system, Konure first creates an empty database and executes the application with command "`liststudentcourses -s 0 -p 1`." The input parameters s and p are set to 0 and 1, respectively, and can be arbitrary integers. After the application executes, it produces an output with general usage information. Konure collects the concrete trace in Figure 4. This trace contains only one query. The query uses a constant `'0'`, which comes from the input parameter s. No data was retrieved from the database. The application outputs did not contain either input parameters. Based on this information, Konure produces the abstract trace in Figure 5. 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, referenced by variable `in_s`.

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

Fig. 4. Concrete trace of the application's first execution. These SQL queries are what the application sent to the database. KONURE captures these SQL queries through its proxy in the execution engine.

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

Fig. 5. Abstract trace of the application's first execution. This abstract trace is converted from the concrete trace in Figure 4 by replacing the constant `'0'` with its source location `in_s`, which denotes the input parameter *s*. This abstract trace is the internal representation that KONURE inference algorithm uses to model the application behavior.

```
d1 <- select student.id, student.password, student.firstname, student.lastname
      where student.id = in_s
if d1 { ?? } else {}
```

Fig. 6. KONURE's hypothesis of the functionality of the application's data interaction component, after the application's first execution. The question marks "??" represent a hidden path in the application that KONURE needs to explore.

KONURE then analyzes this abstract trace and forms a hypothesis for the application in the form of the KONURE DSL, presented in Figure 6. The query result is stored in variable `d1`. KONURE uses the empty `else` branch to denote that the application did not issue any other queries when `d1` was empty.

Based on the current hypothesis, KONURE asks a question about the hidden path in the `if` branch: What will happen if the application enters the `if` branch? To answer this question, KONURE looks for appropriate values for inputs and database contents so that `d1` would contain nonempty data. KONURE provides the following logical formula to an SMT solver:

$$x^1_{student,\,id} = x^1_{\text{in}}$$

Variable $x^1_{student,\,id}$ denotes the value for a student ID. Variable x^1_{in} denotes the value for the first input parameter, *s*. Unconstrained variables include: x^2_{in} (the second input parameter, *p*), $x^1_{student,\,firstname}$ (the value for a student firstname), $x^1_{course,\,name}$ (the value for a course name), etc.

The solver returns with the following assignment:

$$x^1_{student,\,id} = 5, \quad x^1_{student,\,password} = 0, \quad \ldots, \quad x^1_{\text{in}} = 5, \quad x^2_{\text{in}} = 6, \quad \ldots$$

which assigns the same value to student ID and input *s*. It also assigns values to other columns and inputs even if their corresponding variables are unconstrained.

An Execution Trace with Two Queries: KONURE populates the database with the above values, inserting a row into the *student* table where *id* = 5. KONURE runs the application with command "`liststudentcourses -s 5 -p 6`." The application produces the concrete trace in Figure 7. The first query returns the row that KONURE has inserted. The second query returns no data. These queries contain constant `'5'`, which can come from the input parameter *s* or from the result of the first query. The constant `'6'` comes from the second input parameter, *p*. The output from the application is again general usage information that does not contain input parameters or any retrieved data. Based on this information, KONURE produces the abstract trace in Figure 8. In this

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

Fig. 7. Concrete trace of the application's second execution. These SQL queries are what the application sent to the database. KONURE captures these SQL queries through its proxy in the execution engine.

```
q1: select student.id, student.password, student.firstname, student.lastname
    where student.id = in_s
q2: select student.id, student.password, student.firstname, student.lastname
    where student.id = [in_s, q1.student.id] AND student.password = in_p
```

Fig. 8. Abstract trace of the application's second execution. This abstract trace is converted from the concrete trace in Figure 7. The conversion replaces the constant '5' with its two potential source locations in_s and q1.student.id, which denote the input parameter *s* and the column *id* of the result from query q1, respectively. The conversion also replaces the constant '6' with its source location in_p, which denotes the input parameter *p*. This abstract trace is the internal representation that KONURE inference algorithm uses to model the application behavior.

```
d1 <- select student.id, student.password, student.firstname, student.lastname
        where student.id = in_s
if d1 {
  d2 <- select student.id, student.password, student.firstname, student.
        lastname
          where student.id = [in_s, d1.student.id] AND student.password = in_p
  if d2 { ?? } else {}
} else {}
```

Fig. 9. KONURE's hypothesis of the functionality of the application's data interaction component, after the application's second execution. The question marks "??" represent a hidden path in the application that KONURE needs to explore.

abstract trace, the first query is the same as in the first execution of the program in Figure 5. The second query is new. For the student ID, the second query uses either input *s* or the student ID returned from the first query, because these two values were equal during execution. For the student password, the second query uses input *p*.

To refine the hypothesis, KONURE first needs to disambiguate the two potential source locations that the second query used to check the student ID. KONURE asks this question: Is it possible to assign values to the inputs and the database such that the application still executes these two queries but values in_s and q1.student.id differ? KONURE encodes this question with the following logical formulas:

$$x^1_{student,id} = x^1_{\text{in}} \wedge (x^1_{student,id} = x^1_{\text{in}} \rightarrow x^1_{student,id} \neq x^1_{\text{in}})$$

The first clause enforces that query q1 returns a row. The second clause enforces that any data returned by q2 (the first check in the parentheses) must have different values to distinguish in_s and q1.student.id (the check after implication). The SMT solver returns "unsat" which means it is impossible to assign inputs and database contents such that values in_s and q1.student.id differ, in other words, these two values are equivalent and can be used interchangeably. Konure concludes that this abstract trace is accurate enough and is ready for further analysis.[3]

Konure then updates its hypothesis for the application, as in Figure 9. Note that Konure has filled in the `if` branch for the case when `d1` is nonempty. Konure uses an empty `else` branch to denote that the application did not issue more queries when `d2` was empty.

To refine the hypothesis further, Konure asks the question about the unexplored, hidden path: What will happen if the application enters the `if` branch after `d2`? To answer this question, Konure looks for appropriate values for inputs and database contents so that `d1` and `d2` both contain nonempty data. Konure provides the following logical formula to the SMT solver:

$$
\begin{aligned}
x^1_{student,\, id} = x^1_{\text{in}} \wedge (x^2_{student,\, id} = x^1_{\text{in}} \wedge x^2_{student,\, password} = x^2_{\text{in}})\\
\wedge(x^1_{student,\, id} = x^2_{student,\, id} \rightarrow (x^1_{student,\, password} = x^2_{student,\, password}\\
\wedge\, x^1_{student,\, firstname} = x^2_{student,\, firstname} \wedge x^1_{student,\, lasname} = x^2_{student,\, lasname}))
\end{aligned}
$$

In this formula, Konure uses one variable for each input parameter (x^1_{in} and x^2_{in}) and uses two variables for each column of each table ($x^1_{student,id}, x^2_{student,id}, x^1_{student,password}, x^2_{student,password}$, etc). The two variables for each column denotes two potential rows that need to be populated into the database table. The first clause of the formula enforces that `d1` is nonempty and returns at least the row represented by $x^1_{student,*}$. The second clause enforces that `d2` is nonempty and returns at least the row represented by $x^2_{student,*}$. The third clause enforces that table *student* has primary key, *id*. In other words, if any two rows of this table have the same *id*, they must represent the same student. The solver returns a satisfying assignment, which has only one unique row in the *student* table. Konure then uses these new values to execute the application again.

An Execution Trace with a Loop: After several rounds of application executions, one execution of the application issued nine SQL queries to the database. Konure converted these SQL queries into the abstract trace in Figure 10. Query `q3` denotes an inner join on tables *course* and *registration* using the course ID as the key. Queries `q4` through `q9` are produced from a loop in the application. When Konure analyzes this trace, it detects that queries `q4` and `q5` were repeated as a group by three times. Konure also notes that three equals the number of rows retrieved by query `q3`.[4] Based on the Konure DSL in Figure 1, Konure hypothesizes that queries `q4` and `q5` are part of a loop. Konure updates its hypothesis about the application functionality, presented in Figure 11. Note that Konure has formed a `for` loop. In the loop, Konure has placeholders for the case where `d4` contains nonempty data and for the case where `d5` is empty. To explore these hidden paths, Konure asks these questions: (1) What will happen if the application executes a loop iteration that enters the `if` branch after `d4`? (2) What will happen if the application executes a loop iteration that enters the `else` branch after `d5`? Konure encodes these questions into logical formulas and uses the new values returned from the SMT solver to execute the application again.

Konure repeatedly asks questions about hidden paths of the application, executes the application with carefully synthesized values, observes the execution trace, and refines the hypothesis about the application. The inference algorithm terminates when there are no more hidden paths to explore. The completed hypothesis for the student registration example is available in Appendix E.1.

[3]Under a different scenario, if Konure does find out that two ambiguous source locations are not equivalent, Konure will use carefully chosen values to execute the application again. The new execution trace would allow Konure to eliminate the incorrect source location.

[4] Because Konure controls database contents and observes traffic, Konure knows how many rows are retrieved by each query.

```
q1: select student.id, student.password, student.firstname, student.lastname
    where student.id = in_1
q2: select student.id, student.password, student.firstname, student.lastname
    where student.id = [in_1, q1.student.id] AND student.password = in_2
q3: select course.id (output: True), course.name, course.course_number, course
    .size_limit, course.is_offered, course.teacher_id (output: True),
    registration.student_id, registration.course_id (output: True)
    where registration.course_id = course.id AND registration.student_id = [
        in_1, q1.student.id, q2.student.id]
q4: select teacher.firstname, teacher.lastname
    where teacher.id = q3.teacher_id
q5: select count(registration)
    where registration.course_id = [q3.course.id, q3.registratoin.course_id]
q6: select teacher.firstname, teacher.lastname
    where teacher.id = q3.teacher_id
q7: select count(registration)
    where registration.course_id = [q3.course.id, q3.registratoin.course_id]
q8: select teacher.firstname, teacher.lastname
    where teacher.id = q3.teacher_id
q9: select count(registration)
    where registration.course_id = [q3.course.id, q3.registratoin.course_id]
```

Fig. 10. Abstract trace of an execution of the application. This abstract trace is converted from a concrete trace by replacing constants with their source locations `in_1`, `q1.student.id`, `in_2`, `q3.teacher_id`, `q3.course.id`, and so on. This abstract trace is the internal representation that Konure inference algorithm uses to model the application behavior.

After Konure completes the hypothesis of the application, it regenerates an executable program that implements this functionality. Our current Konure implementation regenerates Python code using a standard SQL library to perform the database queries. The regenerated program is available in Appendix D.1.

As shown in this example, the Konure inference algorithm involves resolving ambiguity in abstract traces, detecting and handling loops, refining hypotheses, identifying hidden paths, encoding hidden paths into logical formulas, and solving these formulas. We present the inference algorithm in detail in Section 3.

3 DESIGN

We next present the Konure active learning algorithm.

3.1 Definitions

We first present the definitions of concrete traces, source locations, abstract traces, value assignments, and deduplicated traces.

3.1.1 Concrete Trace. A *concrete trace* is the list of SQL queries, along with the results retrieved from the database, that Konure collects from an execution of the application. The application sends these SQL queries to the database, which responds to the application with data that satisfiy the queries. These SQL queries may contain concrete values, such as integers and strings, that are originally copied from the input parameters and the database contents.

```
d1 <- select student.id, student.password, student.firstname, student.lastname
      where student.id = in_s
if d1 {
  d2 <- select student.id, student.password, student.firstname, student.
      lastname
        where student.id = [in_s, d1.student.id] AND student.password = in_p
  if d2 {
    d3 <- select course.id (output: True), course.name, course.course_number,
        course.size_limit, course.is_offered, course.teacher_id (output: True)
        , registration.student_id, registration.course_id (output: True)
          where registration.course_id = course.id AND registration.student_id
               = [in_1, d1.student.id, d2.student.id]
    for each row3 in d3 {
      d4 <- select teacher.firstname, teacher.lastname
            where teacher.id = row3.teacher_id
      if d4 { ?? } else {
        d5 <- select count(registration)
              where registration.course_id = [row3.course.id, row3.
                  registratoin.course_id]
        if d5 {} else { ?? }
      }
    } else {}
  } else {}
} else {}
```

Fig. 11. Konure's hypothesis of the functionality of the application's data interaction component, after several executions. The question marks "??" represent a hidden path in the application that Konure needs to explore.

CTrace	:=	(CQuery CRes)* CVal*
CQuery	:=	`SELECT` CCol* `FROM` CJoin CWhere
CJoin	:=	t | CJoin `JOIN` t `ON` CCol = CCol
CWhere	:=	ϵ | `WHERE` CExpr
CExpr	:=	CCol = CCol | CCol = CVal | CCol `IN` CVal$^+$ | CExpr `AND` CExpr
CCol	:=	$t.c$
CVal	:=	i | s
CRes	:=	CRow*
CRow	:=	(CCol CVal)$^+$

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

Fig. 12. Abstract syntax for concrete traces

The concrete trace is an instance of the syntax in Figure 12. Each "CQuery" in the concrete trace represents an SQL query that the application sends to the database, collected by Konure as it intercepts the database traffic. The query may select columns ("CCol*") from the join of one or more tables ("CJoin"). The query may select all the rows ("CWhere := ϵ") or select the rows that satisfy certain selection criteria ("CWhere := `WHERE` CExpr"). The selection criteria may be conjunctions of expressions that perform equality or membership checks ("CExpr"). Each selection criterion may use concrete values ("CVal") that are provided by the inputs or produced by earlier queries. Each "CRes" in the concrete trace represents the data that the corresponding query retrieved from the database. The "CRes" consists of a list of tuples that each represent a row ("CRow") of data. Each row contains a list of column-value pairs. The mechanism for abstracting away the concrete values in "CVal" and "CRes" will be discussed later in Section 3.1.3.

Apart from the list of query-result pairs, the concrete trace also contains a list of values (CVal^*) collected from the outputs of the application. Konure collects the interesting values in the outputs that originally came from either the input parameters or the database content.

3.1.2 Source Location. Recall that a Konure DSL program may use the production "Expr := Col = *in*" to refer to the value of an input parameter. The program may also use the production "Expr := Col ∈ *d* Col" to refer to a column "Col" in the results retrieved by an earlier query, "*d*". The program will use these referenced concrete values to populate the SQL queries it sends to the database. These SQL queries will then be captured in the concrete trace.

A *source location* for a concrete value in the concrete trace is a location from which the original value can be copied. When the application refers to an input parameter, the database queries during execution will replace the symbol "*in*" with the concrete value of the corresponding input parameter in that execution. In this case, we call the input parameter "*in*" as a source location for this concrete value. When the application refers to the data retrieved from an earlier query, the database queries during execution will replace the the symbols "*d* Col" with the concrete values of these retrieved results in that execution. In this case, we call the column "Col" of the query "*d*" the source location for these concrete values.

Konure uses the concrete values observed in the database query traffic to reconstruct the source locations for the values that appear in the database queries. To find the source location for each concrete value in an observed database query, Konure matches this concrete value with equal values in the input parameters and results from previously issued database queries. Algorithm 8 (in the Appendix) presents the algorithm that finds the source location for each concrete value. Informally, the source location for a concrete value "CVal" is the variable in the original application that produced this value.

One complication is the possibility that two distinct source locations may have the same concrete value. When Konure encounters such ambiguities, it generates additional constraints that force ambiguous source locations to have different values. If the resulting constraints are not satisfiable, then the source locations must always have the same value. If the resulting constraints are satisfiable, then Konure uses the resulting value assignment to rerun the application and observe an unambiguous and accurate source location in the new trace. Konure thus iteratively learns the relations between various input parameters and various rows/columns of query results. Konure uses this information throughout the inference algorithm to ensure that only unambiguous traces are used for analysis. Note that when two source locations are equivalent, they can be used interchangeably in the program. In this case, Konure uses any one of them when generating the inferred program.

3.1.3 Abstract Trace. 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 source locations and replacing SQL syntax with the syntax of abstract traces.

The abstract trace is an instance of the syntax in Figure 13. Each "AQuery" in the abstract trace is generalized from a "CQuery" in the concrete trace. The main modifications are to replace each concrete value ("CVal") by its source locations ("ASrc^+"), and to summarize the retrieved data ("CRes") with the number of rows ("ARes"). Each source location ("ASrc") refers to either an input

$$
\begin{array}{lcl}
\text{ATrace} & := & (\text{AQuery ARes})^* \ \text{ASrc}^* \\
\text{AQuery} & := & d \leftarrow \texttt{select}\ \text{ACol}^*\ \texttt{where}\ \text{AExpr} \\
\text{AExpr} & := & \text{True} \mid \text{ACol} = \text{ACol} \mid \text{ACol} \in \text{ASrc}^+ \mid \text{AExpr} \land \text{AExpr} \\
\text{ASrc} & := & in \mid d\ r\ \text{ACol} \\
\text{ACol} & := & t.c \\
\text{ARes} & := & r
\end{array}
$$

$$in, d \in \mathit{Variable}, \quad t \in \mathit{Table}, \quad c \in \mathit{Column}, \quad r \in \mathit{Int}$$

Fig. 13. Abstract syntax for abstract traces

Algorithm 1

Input: D is a pair of functions that each maps an accessible location (an input parameter or an earlier query) to the data available at the location. D represents all the data available before the current query and is constructed by the GeneralizeTrace procedure.

Input: e_c is the selection criteria for the current query in the concrete trace and is an instance of the nonterminal "CExpr" in the syntax for concrete traces.

Output: Abstract expression for e_c, as an instance of the nonterminal "AExpr" in the syntax for abstract traces, where the concrete values are replace with variables that refer to data available in D.

```
1:  procedure GeneralizeExpr($D, e_c$)
2:      if $e_c$ is of the form "CCol = CCol" then
3:          return $e_c$
4:      else if $e_c$ is of the form "CCol = CVal" then
5:          "$c = v_c$" ← $e_c$
6:          $L$ ← SourceLocations($D, v_c$)
7:          return "$c \in L$"
8:      else if $e_c$ is of the form "CCol IN CVal$^+$" then
9:          "$c$ IN $v_c$" ← $e_c$
10:         $L \leftarrow \bigcap_{v \in v_c}$ SourceLocations($D, v$)
11:         return "$c \in L$"
12:     else if $e_c$ is of the form "CExpr AND CExpr" then
13:         "$e_{c1}$ AND $e_{c2}$" ← $e_c$
14:         $e_{a1}$ ← GeneralizeExpr($D, e_{c1}$)
15:         $e_{a2}$ ← GeneralizeExpr($D, e_{c2}$)
16:         return "$e_{a1} \land e_{a2}$"
17:     end if
18: end procedure
```

parameter ("*in*") or an earlier query result. The earlier query is identified by the variable d that the query assigns to ("AQuery"), a row number, and a column ("$d\ r$ ACol").

Apart from rewriting concrete values in the queries, Konure also replaces the concrete values in the outputs with source locations ("ASrc$^+$").

Algorithm 9 (in the Appendix) presents the procedure that generalizes a concrete trace to an abstract trace. This procedure calls the procedures in Algorithms 10 (in the Appendix) and 1. The

Algorithm 2

Input: A is a value assignment for the input parameters and the database contents.
Output: T is a deduplicated trace collected from an execution of the application. If any loop is detected during execution, the returned trace contains only one iteration of the loop.
Output: l is a list of locations in T that are originally found to be the starting location of loops in the abstract trace.

```
1:  procedure DeduplicatedTrace($A$)
2:      $\langle \sigma_i, \sigma_d \rangle \leftarrow A$
3:      Populate the database with contents as specified in $\sigma_d$
4:      Execute the application with input parameters as specified in $\sigma_i$
5:      $T_c \leftarrow$ The concrete trace collected from the execution
6:      $T_a \leftarrow$ GeneralizeTrace($T_c$)                    ▹ Abstract trace
7:      $l_a \leftarrow$ DetectLoops($T_a$)
8:      $\langle T, l \rangle \leftarrow$ KeepOneIteration($T_a, l_a$)          ▹ Deduplicated trace
9:      return $\langle T, l \rangle$
10: end procedure
```

critical step is performed in the procedure GeneralizeExpr in Algorithm 1, which converts an expression in the concrete trace into a corresponding expression in the abstract trace.

3.1.4 Value Assignment. A *value assignment* specifies the values to be assigned to the input parameters and to the database contents, using the following domain:

$$\begin{aligned} \mathit{Assignment} &= \mathit{Input} \times \mathit{Database} \\ \mathit{Input} &= \mathit{Variable} \longrightarrow \mathit{Value} \\ \mathit{Database} &= \mathit{Table} \longrightarrow \mathit{Int} \longrightarrow \mathit{Column} \longrightarrow \mathit{Value} \\ \mathit{Value} &= \mathit{Int} \cup \mathit{Str} \end{aligned}$$

A value assignment $A = \langle \sigma_i, \sigma_d \rangle \in \mathit{Assignment}$ contains information about the values to assign to the inputs ($\sigma_i \in \mathit{Input}$) and the database ($\sigma_d \in \mathit{Database}$). The input information σ_i maps an input parameter $\mathit{in} \in \mathit{Variable}$ to a concrete value. The database information σ_d maps a database location (identified by a table, a row number, and a column) to a concrete value.

3.1.5 Deduplicated Trace. When the Konure DSL program has a loop that iterates over a set of data, the application during execution will repeat the loop body once for each row of the data. The more rows there are during an execution, the longer the concrete/abstract traces will be. To ensure termination of the inference algorithm, we preprocess each trace so that the trace contains only one iteration of each loop.

A *deduplicated trace* is a trace that is an instance of the syntax of abstract traces but contains at most one iteration of each loop. Algorithm 2 presents how Konure collects a deduplicated trace from an execution of the application.

The procedure DetectLoops finds loops in the abstract trace as follows. Consider the scenario when there is a "For" loop in the application and when the data that the loop iterates over indeed contains multiple rows. This application will execute certain queries multiple times, each time with different concrete data filled in. Recall that the Konure DSL requires each "For" loop to start with a query that uses the data that the loop iterates over. This query will appear in the abstract trace multiple times, each time using a slightly different source location ("ASrc"): Each time the source

Algorithm 3

Output: The inferred application expressed in the Konure DSL.

```
1: procedure InferProg
2:     $A \leftarrow$ Value assignment for empty database and distinct input parameters
3:     $\langle T, l \rangle \leftarrow$ DeduplicatedTrace($A$)
4:     return InferBlock($T, l, 1$)
5: end procedure
```

location refers to the same source query ("*d*") and the same column ("ACol") but a different row ("*r*"). This query is repeated once for each row of the data that the loop iterates over. The procedure DetectLoops identifies all groups of repetitive queries that share this pattern. When there are multiple groups of such queries, the procedure finds the group that starts earliest in the abstract trace and uses the first query of the group as the start of the first loop iteration. For each iteration of the loop, the procedure continues recursively to find nested loops in the abstract trace. For each loop found, DetectLoops returns the starting location of each loop iteration.

The procedure KeepOneIteration takes the abstract trace along with the information of loops found by DetectLoops. For each loop found, the procedure discards all but one iteration in the abstract trace, resulting in a deduplicated trace. This procedure also keeps track of the locations in the deduplicated trace that were originally the starting locations of loops. The KeepOneIteration returns the deduplicated trace and the corresponding information of loop locations.

3.2 Syntax-guided active learning algorithm

The Konure inference algorithm is constructive [Angluin and Smith 1983], that is, instead of enumerating candidate solutions in a large search space, the algorithm constructs the solution progressively every time Konure finds an interesting behavior of the application.

At a high level, the algorithm starts with an initial abstract syntax tree (AST) as the initial hypothesis and progressively expands the hypothesis until all details are fully inferred. More concretely, the algorithm maintains an AST, in the Konure DSL, with nonterminals denoting hidden parts that are left to infer. Konure fills in this tree from top to bottom. The inference proceeds by expanding nonterminals until the AST is complete, resulting a complete program. Each time Konure executes the application, Konure chooses values for inputs and database contents to disambiguate the nonterminals in turn. These values are chosen so that the use of each potential production would exhibit a different behavior when the application is executed. The algorithm eventually visits all the hidden parts of the tree to terminate with a correctly inferred model.

The entry point of the inference algorithm is the procedure InferProg in Algorithm 3. This procedure first executes the application with an initial value assignment, which produces a deduplicated trace. This trace serves as an initial observation for Konure to start constructing the hypothesis program. This trace is passed into the procedure InferBlock in Algorithm 4.

The InferBlock procedure recursively constructs deeper parts of the hypothesis program. Intuitively, the algorithm tries to flip the result for each query in the deduplicated trace, each time forcing the application to reveal a hidden path in the next execution if such a hidden path exists.

To flip a query's result, the algorithm calls FlipTrace in Algorithm 5. This procedure first alternates the given trace to produce a desired trace. The desired trace is obtained by first truncating the given trace up to the specified location and then forcing the last query to return a specified result. The procedure then asks the solver whether this desired trace is possible for a program to

Algorithm 4

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

```
1: procedure InferBlock($T, l, i$)
2:     if $T$ is Nil or $i >$ length of $T$ then
3:         For each output value in $T$, mark the source query that generated it
4:         return “Nil”                                          ▷ Production “Block := $\epsilon$”
5:     end if
6:     $\langle A_0, T_0, l_0\rangle \leftarrow$ FlipTrace($T, i$, “$= 0$”)
7:     $\langle A_1, T_1, l_1\rangle \leftarrow$ FlipTrace($T, i$, “$\geq 1$”)
8:     $\langle A_2, T_2, l_2\rangle \leftarrow$ FlipTrace($T, i$, “$\geq 2$”)
9:     if $A_0$ satisfiable then
10:        $\langle\langle q_1, r^0_1\rangle, \ldots, \langle q_i, r^0_i\rangle, \langle q^0_{i+1}, r^0_{i+1}\rangle, \ldots, \langle q^0_{k_0}, r^0_{k_0}\rangle\rangle \leftarrow T_0$
11:    end if
12:    if $A_1$ satisfiable then
13:        $\langle\langle q_1, r^1_1\rangle, \ldots, \langle q_i, r^1_i\rangle, \langle q^1_{i+1}, r^1_{i+1}\rangle, \ldots, \langle q^1_{k_1}, r^1_{k_1}\rangle\rangle \leftarrow T_1$
14:    end if
15:    if $A_2$ satisfiable and $l_2$ says $T_2$ has loop starting at $(i+1)$ then ▷ Production “Block := For”
16:        $b_t \leftarrow$ InferBlock($T_2, l_2, i+1$)
17:        $b_f \leftarrow$ InferBlock($T_0, l_0, i+1$)
18:        return “for $q_i$ do $b_t$ else $b_f$”
19:    else if $A_0$ satisfiable and $A_1$ satisfiable and $q^0_{i+1} \neq q^1_{i+1}$ then   ▷ Production “Block := If”
20:        $b_t \leftarrow$ InferBlock($T_1, l_1, i+1$)
21:        $b_f \leftarrow$ InferBlock($T_0, l_0, i+1$)
22:        return “if $q_i$ then $b_t$ else $b_f$”
23:    else                                          ▷ Production “Block := Query Block”
24:        $b \leftarrow$ InferBlock($T, l, i+1$)
25:        return “$q_i$ $b$”
26:    end if
27: end procedure
```

produce. If this desired trace is possible to produce, the solver returns a satisfying value assignment, which is then used by the inference algorithm to execute the given application again and observe more information. If the desired trace is impossible to produce, the solver returns “Unsat”, which is then used by the inference algorithm to learn that the specified hidden path does not exist.

To ask the solver whether a desired trace is satisfiable, the algorithm calls EncodePathConstraints in Algorithm 6 to generate a logical formula from the desired trace. The logical formula

Algorithm 5

Input: T is a trace collected from the DeduplicatedTrace procedure, which contains a list of query-result pairs ($\langle q, r\rangle$) collected from an execution of the application.
Input: i is a positive integer.
Input: m describes the desired result for the i-th query.
Output: When it is possible to reproduce trace T up to the i-th query with only the i-th result changed to m, return a satisfying value assignment and a new trace (along with loop information) collected from using this assignment to execute the application. When it is not possible, return Unsat.

```
1: procedure FlipTrace($T, i, m$)
2:     $\langle\langle q_1, r_1\rangle, \ldots, \langle q_i, r_i\rangle, \ldots, \langle q_k, r_k\rangle\rangle \leftarrow T$
3:     $f \leftarrow$ “Result for $q_i$ should have $m$ rows”
4:     $F \leftarrow \langle\langle q_1, r_1\rangle, \ldots, \langle q_{i-1}, r_{i-1}\rangle, \langle q_i, f\rangle\rangle$
5:     $C \leftarrow$ EncodePathConstraints($F$)
6:     Ask solver to find a value assignment that satisfies constraint $C$
7:     if satisfiable then
8:         $A \leftarrow$ The satisfying assignment
9:         $\langle T', l'\rangle \leftarrow$ DeduplicatedTrace($A$)
10:        while $T'$ has unresolved ambiguity do
11:            Ask solver for a value assignment that satisfies $C$ but with less ambiguity
12:            $A \leftarrow$ Satisfying assignment with less ambiguity
13:            $\langle T', l'\rangle \leftarrow$ DeduplicatedTrace($A$)
14:        end while
15:        return $\langle A, T', l'\rangle$
16:    else
17:        return $\langle$Unsat, Nil, Nil$\rangle$
18:    end if
19: end procedure
```

$$\mathrm{E} \;:=\; \text{True} \mid \text{False} \mid x = x \mid \neg\,\mathrm{E} \mid \mathrm{E} \wedge \mathrm{E} \mid \mathrm{E} \vee \mathrm{E}$$
$$x \in \mathit{Variable}$$

Fig. 14. Syntax for the constraint language

uses the constraint language presented in Figure 14. The procedure first handles the trivial contradiction when certain tables are required to be empty and nonempty at the same time. The procedure then handles the main case as follows. It first declares variables for all input parameters and nonempty tables. These variables are then used to describe the desired path with a logical formula. The logical formula contains two parts: C_p enforces the database restriction that the primary key of a table must be unique. C_q enfoces the restrictions from the desired path F by forcing the results for each query.

When constructing C_q, the algorithm encodes each query one at a time, by calling procedure EncodeQueryConstraints. We present a simplified version of this procedure in Algorithm 7,

Algorithm 6

Input: F is a desired trace constructed by the FlipTrace procedure, which contains a list of query-result pairs ($\langle q, r \rangle$). In each query-result pair, the query (q) is an instance of the nonterminal symbol "Query" in the Konure DSl that represents an SQL query that the application sent to the database. The result (r) describes the *desired* number of rows that should be retrieved by the corresponding query. Each query has the following attributes: The q.expr attribute is an instance of the nonterminal symbol "AExpr" in the sytnax for abstract traces that represents the criteria for selecting rows. The q.tables attribute is the set of tables that are used by the query. Each table t has the following attributes: The t.columns attribute is the set of all columns of table t. The t.key attribute is t's primary key column if defined in the database schema, or Nil if t does not have a primary key.

Output: A logical formula using the constraint language in Figure 14. This formula specifies the constraints that need to be satisfied by the input parameters and the database contents such that executing the queries in F will retrieve the desired results.

```
1:  procedure EncodePathConstraints($F$)
2:      $\langle\langle q_1, r_1\rangle, \ldots, \langle q_k, r_k\rangle\rangle \leftarrow F$
3:      $I_e \leftarrow \{i \in \{1, \ldots, k\} \mid r_i = 0 \wedge q_i.\text{expr is "True"}\}$
4:      $E \leftarrow \bigcup_{i \in I_e} q_i.\text{tables}$                                  ▷ Tables that must be empty
5:      $I_n \leftarrow \{i \in \{1, \ldots, k\} \mid r_i > 0\}$
6:      $N \leftarrow \bigcup_{i \in I_n} ((\bigcup_{s \in \text{SourceQueries}(F,\, i)} q_s.\text{tables}) \cup q_i.\text{tables})$   ▷ Tables that must be nonempty
7:      if $N \cap E \neq \emptyset$ then
8:          return "False"                                                  ▷ Trivially unsatisfiable
9:      else
10:         $T \leftarrow$ All tables
11:         for table $t \in T - E$, column $c \in t$.columns, $j \in \{1, \ldots, k\}$ do
12:             Declare variables $x^{j}_{t,c}$ and $x^{2j}_{t,c}$              ▷ Declare $2k$ variables for each column
13:         end for
14:         for the $j$-th input parameter of the application do
15:             Declare variable $x^{j}_{\text{in}}$                          ▷ Declare a variable for each input
16:         end for
17:         $T_k \leftarrow \{t \in T - E \mid t.\text{key is not Nil}\}$
18:         $C_p \leftarrow \bigwedge_{t \in T_k} \bigwedge_{i_1, i_2 \in \{1,\ldots,k\}} ((\neg(x^{i_1}_{t,t.\text{key}} = x^{i_2}_{t,t.\text{key}})) \vee (\bigwedge_{c \in t.\text{columns}} x^{i_1}_{t,c} = x^{i_2}_{t,c}))$ ▷ Primary keys
19:         $I_{nt} \leftarrow \{i \in \{1, \ldots, k\} \mid q_i.\text{expr is not "True"}\}$
20:         $C_q \leftarrow \bigwedge_{i \in I_{nt}}$ EncodeQueryConstraints($F$, $i$)              ▷ Queries
21:         return "$C_p \wedge C_q$"
22:     end if
23: end procedure
```

which works for queries that are not in a loop.[5] This procedure first collects all the source queries that are referred to, directly or indirectly, by the current query (procedure SourceQueries). The procedure then uses all the source queries to encode a formula that enforces the desired result. If the current query is desired to return nonempty data, the procedure enforces this result by adding

[5]Our full implementation encodes constraints appropriately for queries in loops.

Algorithm 7

Input: F is a desired trace constructed by the FlipTrace procedure, which contains a list of query-result pairs ($\langle q, r\rangle$). In each query-result pair, the query (q) is an instance of the nonterminal symbol "Query" in the Konure DSl that represents an SQL query that the application sent to the database. The result (r) describes the *desired* number of rows that should be retrieved by the corresponding query.
Input: i is a positive integer.
Output: A logical formula using the constraint language in Figure 14. This formula specifies the constraints that need to be satisfied by the input parameters and the database contents such that executing the i-th query in F will retrieve the desired results.

```
1:  procedure EncodeQueryConstraints($F, i$)
2:      $\langle\langle q_1, r_1\rangle, \ldots, \langle q_k, r_k\rangle\rangle \leftarrow F$
3:      $S \leftarrow$ SourceQueries($F, i$)
4:      $T \leftarrow (\bigcup_{s \in S} q_s.\text{tables}) \cup q_i.\text{tables}$
5:      if $r_i$ is not "= 0" then
6:          $J_1 \leftarrow \{\langle t, i\rangle \mid t \in T\}$        ▷ $J_1$ is a function that maps a table $t$ to integer $i$
7:          $J_2 \leftarrow \{\langle t, (k+i)\rangle \mid t \in T\}$        ▷ $J_2$ is a function that maps a table $t$ to integer $(k+i)$
8:          $C_1 \leftarrow$ EncodeExpr($(\bigwedge_{s \in S} q_s.\text{expr}) \wedge q_i.\text{expr}, J_1$)        ▷ Query $q_i$ returns at least the row $i$
9:          if $r_i$ is "$\geq 1$" then
10:             return $C_1$
11:         else        ▷ $r_i$ is "$\geq 2$"
12:             $C_2 \leftarrow$ EncodeExpr($(\bigwedge_{s \in S} q_s.\text{expr}) \wedge q_i.\text{expr}, J_2$)
                    ▷ Query $q_i$ returns at least the row $(k+i)$
13:             $C_d \leftarrow \bigvee_{t \in q_i.\text{tables}} (\bigvee_{c \in t.\text{columns}} \neg(x^i_{t,c} = x^{k+i}_{t,c}))$        ▷ Row $i$ and row $(k+i)$ differ
14:             return $C_1 \wedge C_2 \wedge C_d$
15:         end if
16:     else        ▷ $r_i$ is "= 0"
17:         $R \leftarrow \{J : T \rightarrow \{1, \ldots, 2k\}\}$        ▷ $R$ is the set of functions that map tables to integers
18:         if $S = \emptyset$ then
19:             return $\bigwedge_{J \in R} \neg$ EncodeExpr($q_i.\text{expr}, J$)
                    ▷ No combination of tables/rows may satisfy $q_i$
20:         else
21:             return $\bigwedge_{J \in R} \neg$EncodeExpr($(\bigwedge_{s \in S} q_s.\text{expr}) \wedge q_i.\text{expr}, J$)
                    ▷ If source queries are satisfied, no combination of tables/rows may satisfy $q_i$
22:         end if
23:     end if
24: end procedure
```

a qualifying row. If the current query is desired to return empty data, the procedure enforces this result by requiring that no row satisfy the selection criteria.

To encode the selection criteria for each query, the algorithm calls procedure EncodeExpr in Algorithm 11 (in the Appendix). This procedure transforms each nonterminal symbol "AExpr" from the deduplicated trace into a logical formula using the variables declared earlier in EncodePathConstraints and using the constraint language in Figure 14.

3.3 Correctness

We present proof outlines for the termination, the complexity, and the completeness of the Konure inference algorithm. A precondition for all of these propositions is that there exists a model P expressed in the Konure DSL that completely captures the semantics of the application.

Notation: Let P be a model of the application in the Konure DSL. Let P' be a Konure DSL program that is equivalent to P but omits unreachable branches. Specifically, it (1) uses straight-line sequential code to replace any unreachable conditional branches in P and (2) uses an `if` statement to replace any `for` loop that can be repeated at most once in P.

Let Q be the inferred program produced by procedure InferProg. Let N be the number of times that the AST of program P' expands a production for the "Block" nonterminal symbol. Let B_p be the program that corresponds to the nonterminal "Block" in P'. Let B_q be the outcome of procedure InferBlock when inferring P.

Lemma 1. *For any deduplicated trace T produced by executing program P, there exists a path in the AST of program P' from root to leaf where each "Query" nonterminal symbol corresponds to a query in T in the same order.*

Proof outline. We perform an induction on the length of T and the structure of B_p.

(1) If B_p is of the form "ϵ": Program P' is an empty program has no "Query" symbol in B_p. Any trace produced by executing P is empty. Hence, T contains no query, which consistent with B_p.

(2) If B_p is of the form "Query Block": We denote these two nonterminals as "Q_p B_{p1}". Any trace produced by executing P starts with an instance of Q_p as the first query. Thus, the first query in T corresponds to the first "Query" nonterminal symbol in B_p. By the inductive hypothesis, the remainder of T corresponds to the "Query" nonterminals in B_{p1}.

(3) If B_p is of the form "If": B_p expands to "`if` Query `then` Block `else` Block", denoted by "`if` Q_p `then` B_{p1} `else` B_{p2}". Any trace produced by executing P starts with an instance of Q_p as the first query. Thus, the first query in T corresponds to the first "Query" nonterminal symbol in B_p. Depending on the result for Q_p, the execution continues into either B_{p1} or B_{p2}. If execution continues into B_{p1}, by the deductive hypothesis, the remainder of T corresponds to the "Query" nonterminals in B_{p1}. If execution continues into B_{p2}, by the deductive hypothesis, the remainder of T corresponds to the "Query" nonterminals in B_{p2}. Either case, the original trace T corresponds to the "Query" nonterminals in B_p.

(4) If B_p is of the form "For": B_p expands to "`for` Query `do` Block `else` Block", denoted by "`for` Q_p `do` B_{p1} `else` B_{p2}". Any trace produced by executing P starts with an instance of Q_p as the first query. Thus, the first query in T corresponds to the first "Query" nonterminal symbol in B_p. Depending on the result for Q_p, the execution continues into either B_{p1} or B_{p2}. If execution continues into B_{p1}, the loop body may be executed multiple times, but the deduplicated trace T contains only one iteration. By the deductive hypothesis, the remainder of T corresponds to the "Query" nonterminals in B_{p1}. If execution continues into B_{p2}, by the deductive hypothesis, the remainder of T corresponds to the "Query" nonterminals in B_{p2}. Either case, the original trace T corresponds to the "Query" nonterminals in B_p.

□

Lemma 2. *B_p and B_q use the same productions.*

Proof outline. We perform a structural induction on B_p.

(1) If B_p is of the form "ϵ": Program P' is an empty program. Any trace produced by executing P is empty. Hence, the trace T collected by InferProg on line 3 is empty. This empty trace is passed to procedure InferBlock, which on line 4 directly returns an empty program as the inference result B_q. In this case, both B_p and B_q use the production "Block := ϵ".

(2) If B_p is of the form "Query Block": We denote these two nonterminals as "Q_p B_{p1}". The inference in procedure InferBlock will enter the branch on line 23 for the following reasons: (a) No matter how many rows query Q_p returns (after flipping results on lines 6–8), the new trace will always contain query Q_p followed by the same first query in B_{p1}. (b) The new trace will not have a loop following immediately after Q_p.

Let B_{q1} be outcome of the recursive call to InferBlock on line 24. Because the recursive call increments the argument i by 1, the callee uses the fragment of the input trace excluding the first query. According to Lemma 1, this fragment corresponds to the subtree of B_p excluding the root, which is B_{p1}. By the inductive hypothesis, B_{p1} and B_{q1} use the same productions. Because InferBlock then returns B_q = "Q_p B_{q1}" as the inference result, it uses the production "Block := Query Block". This production is same as the production used by B_p.

(3) If B_p is of the form "If": B_p expands to "`if` Query `then` Block `else` Block", denoted by "`if` Q_p `then` B_{p1} `else` B_{p2}". The inference in procedure InferBlock will enter the branch on line 19, for the following reasons: (a) Both the `if` and the `else` branches are reachable (as P' does not have unreachable branches). Thus, both A_0 and A_1 are satisfiable. (b) As required by the Konure DSL, B_{p1} and B_{p2} must have different queries as their first queries. Thus, the condition $q^0_{i+1} \neq q^1_{i+1}$ holds. (c) No matter how many rows query Q_p returns (after flipping results on lines 6–8), the new trace from executing P does not have a loop that immediately follows Q_p.

After entering the branch on line 19, the inference recursively calls InferBlock twice to infer the two different branches B_{p1} and B_{p2}, respectively. The input traces to these two recursive calls are as follows. The first call receives trace T_1, which was forced to return at least one row from query Q_p. Hence, the query that follows Q_p in T_1 will be the first query in the `if` branch, B_{p1}. The second call receives trace T_0, which was forced to return empty from query Q_p. Hence, the query that follows Q_p in T_0 will be the first query in the `else` branch, B_{p2}.

Let B_{q1} and B_{q2} be the outcomes of these two recursive calls. Because each recursive call increments the argument i by 1, each callee uses the fragment of the input trace excluding the first query. According to Lemma 1, these fragments correspond to the subtrees B_{p1} and B_{p2}, respectively. By the inductive hypothesis, B_{p1} and B_{q1} use the same productions and B_{p2} and B_{q2} use the same productions. Because InferBlock then returns B_q = "`if` Q_p `then` B_{q1} `else` B_{q2}" as the inference result, it uses the production "Block := If". This production is same as the production used by B_p.

(4) If B_p is of the form "For": B_p expands to "`for` Query `do` Block `else` Block", denoted by "`for` Q_p `do` B_{p1} `else` B_{p2}". The inference in procedure InferBlock will enter the branch on line 15, for the following reasons: (1) The `for` is possible to execute more than one iteration (as P' does not have unreachable branches). Thus, both A_2 is satisfiable. (2) The trace produced by executing P with A_2 results in a trace with at least two repetitions of the loop body B_{p1}. Thus, this loop will be detected by DeduplicatedTrace.

After entering the branch on line 15, the inference recursively calls InferBlock twice to infer the two different branches B_{p1} and B_{p2}, respectively. The input traces to these two recursive calls are as follows. The first call receives trace T_2, which was forced to return at least two rows from query Q_p. Hence, the query that follows Q_p in T_2 will be the first query in the `for` branch, B_{p1}. Furthermore, the loop body will appear exactly once in T_2, as the loop will be successfully detected and deduplicated. The second call receives trace T_0, which was forced to return empty from query Q_p. Hence, the query that follows Q_p in T_0 will be the first query in the `else` branch, B_{p2}.

Let B_{q1} and B_{q2} be the outcomes of these two recursive calls. Because each recursive call increments the argument i by 1, each callee uses the fragment of the input trace excluding the first query. According to Lemma 1, these fragments correspond to the subtrees B_{p1} and B_{p2}, respectively. By the inductive hypothesis, B_{p1} and B_{q1} use the same productions and B_{p2} and B_{q2} use the same productions. Because InferBlock then returns B_q = "`for` Q_p `do` B_{q1} `else` B_{q2}" as the inference result, it uses the production "Block := For". This production is same as the production used by B_p. □

Proposition 1 (Complexity). *The number of times that procedure InferBlock is called is at most N.*

Proof outline. This result follows directly from Lemma 2. Note that this number of executions is not affected by the number of times that any loop executed. □

Proposition 2 (Termination). *Assuming that the given application terminates, the Konure inference algorithm will terminate.*

Proof outline. Assuming that the given application terminates, each call to DeduplicatedTrace will terminate. The rest of the proof follows directly from Proposition 1. □

Lemma 3. *B_q is equivalent to B_p.*

Proof outline. By Lemma 2, these two "Block" nonterminal symbols use the same productions of the Konure DSL syntax. The remaining part of the proof needs to show that these two programs use equivalent value references for the queries ("Expr := Col ∈ d Col") and the outputs ("Col *out*"). Because Konure disambiguates all semantically different source locations in each deduplicated trace (see Section 3.1.2 and Algorithm 5), the inferred program always uses the correct source locations or their equivalences. □

Proposition 3 (Completeness). *Q is equivalent to P.*

Proof outline. This result follows directly from Lemma 3. □

4 EXPERIMENTAL RESULTS

We obtained three applications that work with an external relational database. 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 applications include:

- **Kandan Chat Room:** Kandan [kan 2018] is an open source chat room application, built with Ruby on Rails (RoR), with over 2700 stars on GitHub. The Kandan server receives HTTP requests, interacts with the database accordingly, and responds with JSON objects that contain data retrieved from the database and HTML templates to display the JSON data.
- **Blog:** The Blog application is an example obtained from the Ruby on Rails website [rai 2018]. The Blog server receives HTTP requests, interacts with the database accordingly, and responds the client with an HTML page that contains the data retrieved from the database.
- **Student Registration:** The student registration application is a Java application developed by a hostile DARPA Red Team to test SQL injection attack detection and nullification techniques. It implements a command-line interface that receives text commands, interacts with the database accordingly, and responds with text output.

Table 1. Application detail

App	Command	Parameters	Tabs	Cols	PL
Kandan	`get_channels`	`username`	4	50	RoR
Kandan	`get_channels_id_activities`	`username, channel_id`	4	50	RoR
Kandan	`get_channels_id_activities_id`	`username, channel_id, activity_id`	4	50	RoR
Kandan	`get_channels_id_attachments`	`username, channel_id`	4	50	RoR
Kandan	`get_me`	`username`	4	50	RoR
Kandan	`get_users`	`username`	4	50	RoR
Kandan	`get_users_id`	`username, user_id`	4	50	RoR
Blog	`get_articles`	None	2	13	RoR
Blog	`get_article_id`	`article_id`	2	13	RoR
Student reg	`liststudentcourses`	`id, password`	5	20	Java

Table 2. Inference effort

Command	Runs	Solves	Time
`get_channels`	21	133	130 mins
`get_channels_id_activities`	25	193	42 mins
`get_channels_id_activities_id`	14	18	6 mins
`get_channels_id_attachments`	18	62	9 mins
`get_me`	12	162	6 mins
`get_users`	15	312	8 mins
`get_users_id`	12	162	6 mins
`get_articles`	2	11	15 secs
`get_article_id`	6	29	45 secs
`liststudentcourses`	6	20	40 secs

Table 1 presents the applications and commands (along with command parameters) that we infer. The first column (App) presents the name of the application. The second column (Command) presents the name of the inferred command. The third column (Parameters) presents the parameters that the command takes. The next two columns (Tabs and Cols) present the number of tables and total number of columns for the database schema. The last column (PL) presents the language in which the application is implemented.

Table 2 presents statistics from the Konure command inference algorithm on the different commands. The first column (Command) presents the name of the command. The second column (Runs) presents the number of application executions that Konure required to derive the model (in the domain-specific language) of the command. Each execution involves a chosen input presented to the application working with chosen database contents. All commands require fewer than 30 executions to obtain the representation for that command functionality as expressed in the domain-specific language.

The third column (Solves) presents the number of invocations of the Z3 SMT solver that Konure executed to derive the model for the command. Note that because Konure may invoke the SMT solver multiple times (depending on the structure of the application) for each step in the inference, the number of Z3 invocations is larger than the number of application executions.

The fourth column (Time) presents the wall-clock time required to infer the model for each command. We measured the Ubuntu 16.04 virtual machine that uses 2 cores and has 2 GB memory. The host machine uses a processor with 4 cores (2.6 GHz Intel Core i5) and has 8 GB 1600 MHz DDR3 memory. The times vary from less than a minute to about two hours. In general, the times

Table 3. Regenerated code size

Command	Regeneration	LOC	SQL	If	For	Output
get_channels	Appendix B.1	69	16	4	2	34
get_channels_id_activities	Appendix B.2	52	16	6	0	17
get_channels_id_activities_id	Appendix B.3	24	11	3	0	3
get_channels_id_attachments	Appendix B.4	43	13	5	0	14
get_me	Appendix B.5	49	8	3	0	31
get_users	Appendix B.6	76	11	3	0	55
get_users_id	Appendix B.7	49	8	3	0	31
get_articles	Appendix C.1	11	2	0	0	6
get_article_id	Appendix C.2	15	3	1	0	6
liststudentcourses	Appendix D.1	23	5	3	1	3

are positively correlated with the number of solves, the length of the programs, and the number of potentially ambiguous data fields. Most of the inference time was spent on solving for alternative database contents to satisfy various constraints. The inference time also includes the time to set database contents, time to set up the proxy, and time to launch, execute, and tear down the applications (and their web servers).

Table 3 presents statistics summarizing aspects of the regenerated Python code. The first column (Command) presents the name of the command. The second column (Regeneration) presents the Appendix that contains the regenerated Python implementation. The next column (LOC) presents the number of lines of code in the regenerated Python implementation. The next columns (SQL, If, For, and Output) present the number of SQL statements, if statements, for statements, and the number of statements that return database fields that appear in the command output.

We next discuss aspects of each application in turn.

Kandan: Kandan maintains multiple chat rooms (so-called channels) that users can access. It implements commands that enable users to navigate the chat rooms and retrieve chat messages (so-called activities) from different channels. Kandan also maintains information about individuals who use the chat rooms and provides commands that retrieve this information. In general, the commands step through the tables, typically using results returned from earlier look-ups to access the correct data in the current table. As it traverses the tables, it collects information to return to the user.

Blog: The Blog application maintains information about blog articles and blog comments. It implements a command that retrieves all of the articles and another command that retrieves a specific article and its associated comments. In comparison with the regenerated Kandan commands, the regenerated Blog commands are relatively simple - the first command simply accesses the table that contains article information; the second accesses the specific row in the article table that contains the desired article, then looks up the comments for that article in the comments table.

Student Registration: We discuss this application in Section 2. We note that the regenerated student registration application is free of SQL injection attack vulnerabilities present in the original application from the DARPA Red Team.

4.2 Discussion

The ability of Konure to infer and regenerate the commands from these applications reflects the ability of the domain-specific language to represent the kinds of computations that these applications implement. We note that the generated code involves the heavy use of strings and string formatting operations. Because Konure systematically generates these strings from a single algorithm, it is able to include correct formatting and implement correct usage patterns. Konure is also able to recognize and infer loops that iterate over rows returned from previous database queries.

5 RELATED WORK

State Machine Model Learning: State machine learning algorithms [Aarts and Vaandrager 2010; Angluin 1987; Cassel et al. 2016; Chow 1978; Fiterău-Broştean et al. 2016; Grinchtein et al. 2010; Isberner et al. 2014; Moore 1956; Raffelt et al. 2005; Vaandrager 2017; Volpato and Tretmans 2015] construct partial representations of program functionality in the form of finite automata with states and transition rules. State fuzzing tools [Aarts et al. 2013; De Ruiter and Poll 2015] hypothesize state machines for given program implementations. One goal is to aid developers in discovering bugs such as spurious state transitions. Network function state model extraction [Wu et al. 2016] performs program slicing and models the sliced partial programs as packet-processing automata. Konure, in contrast, models application behavior with a domain-specific language whose structure supports an effective active inference algorithm that infers complete application functionality (as opposed to a partial model of the application). Because Konure is designed to infer the complete application functionality, it also supports application regeneration.
Syntax-Guided Synthesis: [Alur et al. 2013] identifies a range of program synthesis problems for which it is productive structure the search space as a domain-specific language and presents a framework for this approach. Konure similarly uses a domain-specific language to structure the search space. Unlike the examples presented in [Alur et al. 2013], Konure exploits the structure of the domain-specific language to obtain a top-down inference algorithm that uses active learning to progressively select productions that refine a working hypothesis represented as a sentential form of the grammar of the domain-specific language. Unlike the vast majority of solver-driven synthesis algorithms (which require finite search spaces), this approach enables Konure to work effectively with an unbounded space of models.
Model Extraction For Machine Learning Models: Model extraction algorithms use queries to construct representations for given programs, where the representations are stateless functions such as decision trees [Bastani et al. 2017a; Craven and Shavlik 1995; Tramèr et al. 2016] or symbolic rules [Towell and Shavlik 1993]. Model compression algorithms [Buciluǎ et al. 2006; Hinton et al. 2015] use machine learning models, such as neural networks, to mimic a given machine learning model, typically by generating inputs (training data) and observing the outputs from the given model. In contrast to inferring machine learning models, Konure targets program components that interact with a database.
Dynamic Analysis for Program Comprehension: There is a large body of research on dynamic analysis for program comprehension, but (because of the complicated logic of Web technologies) relatively little of this research targets Web application servers [Cornelissen et al. 2009]. WAFA [Alalfi et al. 2009] analyzes Web applications, focusing on the interactions between Web components, using source code annotations. In contrast, Konure infers applications without analyzing, modifying, or requiring access to the source code. Konure therefore works for applications written in any language and can infer both Web and non-Web applications that interact with an external relational database.

DAViS [Noughi et al. 2014] visualizes the data-manipulation behavior of an execution of a data-intensive program. DiscoTect [Yan et al. 2004] summarizes the software architecture of a running object-oriented system as a state machine. These techniques 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 configurations that enable it to infer the full application behavior.

Database reverse engineering involves analyzing the program's data access patterns, often for reconstructing implicit assumptions of the database schema [Cleve et al. 2013; Davis and Aiken 2000]. Konure infers programs that interact with databases (and not the structure of the database that the program interacts with).

Database Program Reengineering: Reengineering database-backed programs often involves extracting data-access patterns from the source code and transforming these access patterns into more efficient database queries [Cheung et al. 2013; Cohen and Feldman 2003]. 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, rather than only transforming database queries, in potentially different languages and platforms. Note that while Konure works with applications whose behavior can be expressed in the domain-specific language, the application can implement this behavior using any language or coding patterns.

Concolic testing: Concolic testing [Cadar et al. 2006; Godefroid et al. 2005, 2012; Sen et al. 2005] generates inputs that systematically explore all execution paths in the program. The goal is to find inputs that expose software defects. BuzzFuzz [Ganesh et al. 2009] generates inputs that target defects that occur because of coding oversights at the boundary between application and library code. DIODE [Sidiroglou-Douskos et al. 2015] generates inputs that target integer overflow errors. All of these techniques target with programs written in general-purpose languages such as C, dynamically analyze the execution of the program, and use the resulting information to guide the input generation. 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 domain-specific language, (2) interacts with the application by constructing inputs and database configurations, then observing the resulting database interactions and resulting outputs (as opposed to observing the internal operation of the running program), (3) infers a model that captures the complete functionality of the program, and (4) makes it possible to regenerate a new program in a different language or for a different computation platform.

6 CONCLUSION

Applications that translate commands into database operations 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 domain-specific language that supports the range of computational patterns that these applications exhibit and 2) the inference algorithm itself, which progressively synthesizes inputs and database configurations that productively resolve uncertainty in the current working application behavior hypothesis. Results from our implementation highlight the ability of this approach to reverse engineer and regenerate our benchmark applications.

REFERENCES

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

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

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

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

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

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.

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

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

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

Osbert Bastani, Carolyn Kim, and Hamsa Bastani. 2017a. Interpreting Blackbox Models via Model Extraction. *CoRR* abs/1705.08504 (2017). arXiv:1705.08504 http://arxiv.org/abs/1705.08504

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017b. Active Learning of Points-To Specifications. *CoRR* abs/1711.03239 (2017).

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017c. 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.

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.

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

Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model Compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06)*. ACM, New York, NY, USA, 535–541. https://doi.org/10.1145/1150402.1150464

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

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

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

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

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.

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

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

Mark W. Craven and Jude W. Shavlik. 1995. Extracting Tree-structured Representations of Trained Networks. In *Proceedings of the 8th International Conference on Neural Information Processing Systems (NIPS'95)*. MIT Press, Cambridge, MA, USA, 24–30.

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–.

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.

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.

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.

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.

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.

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

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

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

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.

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

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

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

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

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

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

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.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

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

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.

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.

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 2010, Cape Town, South Africa, 1-8 May 2010*. 215–224.

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

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

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

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

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

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.

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

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

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

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

Geoffrey G. Towell and Jude W. Shavlik. 1993. Extracting Refined Rules from Knowledge-Based Neural Networks. *Mach. Learn.* 13, 1 (Oct. 1993), 71–101. https://doi.org/10.1023/A:1022683529158

Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 601–618.

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

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

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

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

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

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.

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.

A APPENDIX: MORE ALGORITHMS FOR KONURE

Algorithm 8

Input: D is a pair of functions, $\langle D_i, D_q \rangle$, that each maps an accessible location to the data available at the location. Function D_i maps an input variable to its value. Function D_q maps a query variable to the data retrieved by this query.

Input: v is a concrete value collected from an execution of the application and is an instance of the nonterminal "CVal" in the syntax for concrete traces.

Output: The set of source locations for v.

```
1: procedure SourceLocations($D, v$)
2:     $\langle D_i, D_q \rangle \leftarrow D$
3:     $L_i \leftarrow \{$"$in$" $\mid v = D_i(in),\ in \in Variable\}$        ▷ Input parameters with value $v$
4:     $L_q \leftarrow \{$"$d\ r\ t.c$" $\mid v = D_q(d)(r)(t.c),\ d \in Variable,\ r \in Int,\ t \in Table,\ c \in t.$columns$\}$
                                                  ▷ Query result rows/columns with value $v$
5:     return $L_i \cup L_q$
6: end procedure
```

Algorithm 9

Input: T_c is a concrete trace, which contains a list of query-result pairs ($\langle q, r \rangle$) collected from an execution of the application. Each query q is an instance of the nonterminal symbol "CQuery" in the sytnax for concrete traces. Each result r is an instance of the nonterminal "CRes".

Output: Abstract trace for T_c, which contains a list of query-result pairs ($\langle q', r' \rangle$). Each query q' is an instance of the nonterminal symbol "AQuery" in the syntax for abstract traces. Each result r' is an integer and is an instance of the nonterminal "ARes".

```
1: procedure GeneralizeTrace($T_c$)
2:     $\langle \langle q_1, r_1 \rangle, \ldots, \langle q_k, r_k \rangle \rangle \leftarrow T_c$
3:     $T_a \leftarrow$ Empty list
4:     for $i = 1, \ldots, k$ do
5:         $D_i \leftarrow \{\langle$variable "$in$", $v\rangle \mid$ input parameter $in$ has value $v\}$     ▷ Data from inputs
6:         $D_q \leftarrow \{\langle$variable "$d_j$", $r_j\rangle \mid j = 1, \ldots, (i-1)\}$     ▷ Data retrieved from earlier queries
7:         $D \leftarrow \langle D_i, D_q \rangle$                              ▷ All data available before $i$-th query
8:         $q' \leftarrow$ GeneralizeQuery($D, i, q_i$)
9:         $r' \leftarrow$ The number of rows in $r_i$
10:        Append $\langle q', r' \rangle$ to $T_a$
11:    end for
12:    return $T_a$
13: end procedure
```

Algorithm 10

Input: D is a pair of functions that each maps an accessible location (an input parameter or an earlier query) to the data available at the location. D represents all the data available before the current query and is constructed by the GeneralizeTrace procedure.

Input: i is a positive integer that represents the location of a query in the concrete trace.

Input: q_c is the i-th query in the concrete trace and is an instance of the nonterminal "CQuery" in the syntax for concrete traces.

Output: Abstract query for q_c, as an instance of the nonterminal "AQuery" in the syntax for abstract traces, where the concrete values are replaced with variables that refer to data available in D.

```
1: procedure GeneralizeQuery($D, i, q_c$)
2:     $d_a \leftarrow$ Variable named "$d_i$", where "$d$" denotes a variable and $i$ is the given integer
3:     "SELECT $c$ FROM $j_c$ $w_c$" $\leftarrow q_c$
4:     $o_c \leftarrow$ Set of checks "CCol = CCol" after each keyword "ON" in $j_c$
5:     if $o_c = \emptyset$ and $w_c = \epsilon$ then        ▷ No selection criterion (retrieve all rows)
6:         return "$d_a \leftarrow$ select $c$ where True"
7:     end if
8:     if $o_c = \emptyset$ and $w_c \neq \epsilon$ then
9:         $e_c \leftarrow$ The check "CExpr" after the keyword "WHERE" in $w_c$
10:    else if $o_c \neq \emptyset$ and $w_c = \epsilon$ then
11:        $e_c \leftarrow$ Connect the elements in $o_c$ with "AND"
12:    else
13:        $w'_c \leftarrow$ The check "CExpr" after the keyword "WHERE" in $w_c$
14:        $o'_c \leftarrow$ Connect the elements in $o_c$ with "AND"
15:        $e_c \leftarrow$ "$w'_c$ AND $o'_c$"
16:    end if
17:    $e_a \leftarrow$ GeneralizeExpr($D, e_c$)
18:    return "$d_a \leftarrow$ select $c$ where $e_a$"
19: end procedure
```

Algorithm 11

Input: e is an instance of the nonterminal symbol "AExpr" in the sytnax for abstract traces.
Input: J is a function that maps each table to an integer.
Output: A logical formula using the constraint language in Figure 14 that uses variables declared in the EncodePathConstraints procedure.

```
1:  procedure EncodeExpr($e, J$)
2:      for nonterminal symbol "Col" in $e$ do
3:          This nonterminal symbol is of the form "$t.c$" where $t$ is a table and $c$ is a column of $t$
4:          Replace symbol "Col" with variable $x_{t,c}^{J(t)}$
5:      end for
6:      for terminal symbol "$in$" in $e$ do
7:          $j \leftarrow$ the index of this input argument as declared in the application interface
8:          Replace symbol "$in$" with variable $x_{\text{in}}^{j}$
9:      end for
10:     Replace each terminal symbol "$\in$" with the equal sign "$=$"
11:     Remove each terminal symbol "$d$" and "$r$"
12:     return the converted expression
13: end procedure
```

B APPENDIX: REGENERATED CODE FOR KANDAN CHAT ROOM

B.1 Kandan Chat Room Command get_channels

```
def get_channels (conn, inputs):
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`username` = :x0 LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        outputs.extend(util.get_data(s3, 'channels', 'id'))
        outputs.extend(util.get_data(s3, 'channels', 'name'))
        outputs.extend(util.get_data(s3, 'channels', 'user_id'))
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities` WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(s3, 'channels', 'id')})
            outputs.extend(util.get_data(s4, 'activities', 'id'))
            outputs.extend(util.get_data(s4, 'activities', 'content'))
            outputs.extend(util.get_data(s4, 'activities', 'channel_id'))
            outputs.extend(util.get_data(s4, 'activities', 'user_id'))
            outputs.extend(util.get_data(s4, 'activities', 'action'))
            if util.has_rows(s4):
                s83 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE `users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities', 'user_id')})
                outputs.extend(util.get_data(s83, 'users', 'id'))
                outputs.extend(util.get_data(s83, 'users', 'email'))
                outputs.extend(util.get_data(s83, 'users', 'first_name'))
                outputs.extend(util.get_data(s83, 'users', 'last_name'))
                outputs.extend(util.get_data(s83, 'users', 'gravatar_hash'))
                outputs.extend(util.get_data(s83, 'users', 'username'))
                outputs.extend(util.get_data(s83, 'users', 'registration_status'))
                s84 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
                outputs.extend(util.get_data(s84, 'users', 'registration_status'))
                s85 = util.do_sql(conn, "SELECT `channels`.* FROM `channels` WHERE (1=1)", {})
                outputs.extend(util.get_data(s85, 'channels', 'id'))
                outputs.extend(util.get_data(s85, 'channels', 'name'))
                outputs.extend(util.get_data(s85, 'channels', 'user_id'))
                s85_all = s85
                for s85 in s85_all:
                    s86 = util.do_sql(conn, "SELECT COUNT(*) FROM `activities` WHERE `activities`.`channel_id` = :x0", {'x0': util.get_one_data(s85, 'channels', 'id')})
                    s87 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                        activities`  WHERE `activities`.`channel_id` = :x0
                        ORDER BY id DESC LIMIT 30 OFFSET 0", {'x0': util.
                        get_one_data(s85, 'channels', 'id')})
                    outputs.extend(util.get_data(s87, 'activities', 'id'))
                    outputs.extend(util.get_data(s87, 'activities', 'content')
                        )
                    outputs.extend(util.get_data(s87, 'activities', '
                        channel_id'))
                    outputs.extend(util.get_data(s87, 'activities', 'user_id')
                        )
                    outputs.extend(util.get_data(s87, 'activities', 'action'))
                    if util.has_rows(s87):
                        s88 = util.do_sql(conn, "SELECT `users`.* FROM `users`
                              WHERE `users`.`id` IN (:x0)", {'x0': util.
                            get_one_data(s87, 'activities', 'user_id')})
                        outputs.extend(util.get_data(s88, 'users', 'id'))
                        outputs.extend(util.get_data(s88, 'users', 'email'))
                        outputs.extend(util.get_data(s88, 'users', 'first_name
                            '))
                        outputs.extend(util.get_data(s88, 'users', 'last_name'
                            ))
                        outputs.extend(util.get_data(s88, 'users', '
                            gravatar_hash'))
                        outputs.extend(util.get_data(s88, 'users', 'username')
                            )
                        outputs.extend(util.get_data(s88, 'users', '
                            registration_status'))
                    else:
                        pass
                s85 = s85_all
            else:
                s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE
                    `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2,
                    'users', 'id')})
                s6 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`
                    WHERE (1=1)", {})
                outputs.extend(util.get_data(s6, 'channels', 'id'))
                outputs.extend(util.get_data(s6, 'channels', 'name'))
                outputs.extend(util.get_data(s6, 'channels', 'user_id'))
                s6_all = s6
                for s6 in s6_all:
                    s7 = util.do_sql(conn, "SELECT COUNT(*) FROM `activities`
                         WHERE `activities`.`channel_id` = :x0", {'x0': util.
                        get_one_data(s6, 'channels', 'id')})
                    s8 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                        activities`  WHERE `activities`.`channel_id` = :x0
                        ORDER BY id DESC LIMIT 30 OFFSET 0", {'x0': util.
                        get_one_data(s6, 'channels', 'id')})
                s6 = s6_all
        else:
            s36 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
            s37 = util.do_sql(conn, "SELECT `channels`.* FROM `channels` WHERE (1=1)", {})
    else:
        pass
    return outputs
```

B.2 Kandan Chat Room Command get_channels_id_activities

```
def get_channels_id_activities (conn, inputs):
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`username` = :x0 LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities` WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(s3, 'channels', 'id')})
            if util.has_rows(s4):
                s40 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE `users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities', 'user_id')})
                s41 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
                s42 = util.do_sql(conn, "SELECT  `channels`.* FROM `channels` WHERE `channels`.`id` = :x0 LIMIT 1", {'x0': inputs[1]})
                if util.has_rows(s42):
                    s138 = util.do_sql(conn, "SELECT  `activities`.* FROM `activities`  WHERE `activities`.`channel_id` = :x0 ORDER BY id LIMIT 1", {'x0': util.get_one_data(s42, 'channels', 'id')})
                    outputs.extend(util.get_data(s138, 'activities', 'id'))
                    outputs.extend(util.get_data(s138, 'activities', 'content'))
                    outputs.extend(util.get_data(s138, 'activities', 'channel_id'))
                    outputs.extend(util.get_data(s138, 'activities', 'user_id'))
                    outputs.extend(util.get_data(s138, 'activities', 'action'))
                    s139 = util.do_sql(conn, "SELECT  `activities`.* FROM `activities`  WHERE `activities`.`channel_id` = :x0 ORDER BY id DESC LIMIT 30", {'x0': util.get_one_data(s42, 'channels', 'id')})
                    outputs.extend(util.get_data(s139, 'activities', 'id'))
                    outputs.extend(util.get_data(s139, 'activities', 'content'
                        ))
                    outputs.extend(util.get_data(s139, 'activities', '
                        channel_id'))
                    outputs.extend(util.get_data(s139, 'activities', 'user_id'
                        ))
                    outputs.extend(util.get_data(s139, 'activities', 'action')
                        )
                    if util.has_rows(s139):
                        s173 = util.do_sql(conn, "SELECT `users`.* FROM `users
                            `  WHERE `users`.`id` IN :x0", {'x0': util.
                            get_data(s139, 'activities', 'user_id')})
                        outputs.extend(util.get_data(s173, 'users', 'id'))
                        outputs.extend(util.get_data(s173, 'users', 'email'))
                        outputs.extend(util.get_data(s173, 'users', '
                            first_name'))
                        outputs.extend(util.get_data(s173, 'users', 'last_name
                            '))
                        outputs.extend(util.get_data(s173, 'users', '
                            gravatar_hash'))
                        outputs.extend(util.get_data(s173, 'users', 'username'
                            ))
                        outputs.extend(util.get_data(s173, 'users', '
                            registration_status'))
                    else:
                        pass
                else:
                    pass
            else:
                s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE
                    `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2,
                    'users', 'id')})
                s6 = util.do_sql(conn, "SELECT  `channels`.* FROM `channels`
                    WHERE `channels`.`id` = :x0 LIMIT 1", {'x0': inputs[1]})
                if util.has_rows(s6):
                    s69 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                        activities`  WHERE `activities`.`channel_id` = :x0
                        ORDER BY id LIMIT 1", {'x0': util.get_one_data(s6, '
                        channels', 'id')})
                    s70 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                        activities`  WHERE `activities`.`channel_id` = :x0
                        ORDER BY id DESC LIMIT 30", {'x0': util.get_one_data(
                        s6, 'channels', 'id')})
                else:
                    pass
        else:
            s25 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                users', 'id')})
            s26 = util.do_sql(conn, "SELECT  `channels`.* FROM `channels`
                WHERE `channels`.`id` = :x0 LIMIT 1", {'x0': inputs[1]})
    else:
            pass
    return outputs
```

B.3 Kandan Chat Room Command get_channels_id_activities_id

```
def get_channels_id_activities_id (conn, inputs):
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
        username` = :x0 LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users
            `.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')
            })
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities`
                WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(
                s3, 'channels', 'id')})
            if util.has_rows(s4):
                s47 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE
                    `users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities
                    ', 'user_id')})
                s48 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE
                     `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2,
                     'users', 'id')})
                s49 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                    activities`  WHERE `activities`.`id` = :x0 LIMIT 1", {'x0'
                    : inputs[2]})
                outputs.extend(util.get_data(s49, 'activities', 'content'))
            else:
                s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE
                    `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2,
                    'users', 'id')})
                s6 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                    activities`  WHERE `activities`.`id` = :x0 LIMIT 1", {'x0'
                    : inputs[2]})
                outputs.extend(util.get_data(s6, 'activities', 'content'))
        else:
            s25 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                users', 'id')})
            s26 = util.do_sql(conn, "SELECT  `activities`.* FROM `activities`
                 WHERE `activities`.`id` = :x0 LIMIT 1", {'x0': inputs[2]})
            outputs.extend(util.get_data(s26, 'activities', 'content'))
    else:
        pass
    return outputs
```

B.4 Kandan Chat Room Command get_channels_id_attachments

```
def get_channels_id_attachments (conn, inputs):
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
        username` = :x0 LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities` WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(s3, 'channels', 'id')})
            if util.has_rows(s4):
                s40 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE `users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities', 'user_id')})
                s41 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
                s42 = util.do_sql(conn, "SELECT  `channels`.* FROM `channels` WHERE `channels`.`id` = :x0 LIMIT 1", {'x0': inputs[1]})
                if util.has_rows(s42):
                    s126 = util.do_sql(conn, "SELECT `attachments`.* FROM `attachments`  WHERE `attachments`.`channel_id` = :x0 ORDER BY created_at DESC", {'x0': util.get_one_data(s42, 'channels', 'id')})
                    outputs.extend(util.get_data(s126, 'attachments', 'id'))
                    outputs.extend(util.get_data(s126, 'attachments', 'user_id'))
                    outputs.extend(util.get_data(s126, 'attachments', 'channel_id'))
                    outputs.extend(util.get_data(s126, 'attachments', 'message_id'))
                    outputs.extend(util.get_data(s126, 'attachments', 'file_file_name'))
                    outputs.extend(util.get_data(s126, 'attachments', 'file_content_type'))
                    outputs.extend(util.get_data(s126, 'attachments', 'file_file_size'))
                else:
                    pass
            else:
                s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
                s6 = util.do_sql(conn, "SELECT  `channels`.* FROM `channels` WHERE `channels`.`id` = :x0 LIMIT 1", {'x0': inputs[1]})
                if util.has_rows(s6):
                    s68 = util.do_sql(conn, "SELECT `attachments`.* FROM `attachments`  WHERE `attachments`.`channel_id` = :x0 ORDER BY created_at DESC", {'x0': util.get_one_data(s6, 'channels', 'id')})
                    outputs.extend(util.get_data(s68, 'attachments', 'id'))
                    outputs.extend(util.get_data(s68, 'attachments', 'user_id'))
                    outputs.extend(util.get_data(s68, 'attachments', '
                        channel_id'))
                    outputs.extend(util.get_data(s68, 'attachments', '
                        message_id'))
                    outputs.extend(util.get_data(s68, 'attachments', '
                        file_file_name'))
                    outputs.extend(util.get_data(s68, 'attachments', '
                        file_content_type'))
                    outputs.extend(util.get_data(s68, 'attachments', '
                        file_file_size'))
                else:
                    pass
        else:
            s25 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                users', 'id')})
            s26 = util.do_sql(conn, "SELECT  `channels`.* FROM `channels`
                WHERE `channels`.`id` = :x0 LIMIT 1", {'x0': inputs[1]})
    else:
        pass
    return outputs
```

B.5 Kandan Chat Room Command get_me

```
def get_me (conn, inputs):
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
        username` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'users', 'id'))
    outputs.extend(util.get_data(s0, 'users', 'email'))
    outputs.extend(util.get_data(s0, 'users', 'first_name'))
    outputs.extend(util.get_data(s0, 'users', 'last_name'))
    outputs.extend(util.get_data(s0, 'users', 'username'))
    outputs.extend(util.get_data(s0, 'users', 'registration_status'))
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users
            `.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')
            })
        outputs.extend(util.get_data(s2, 'users', 'id'))
        outputs.extend(util.get_data(s2, 'users', 'email'))
        outputs.extend(util.get_data(s2, 'users', 'first_name'))
        outputs.extend(util.get_data(s2, 'users', 'last_name'))
        outputs.extend(util.get_data(s2, 'users', 'username'))
        outputs.extend(util.get_data(s2, 'users', 'registration_status'))
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities`
                WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(
                s3, 'channels', 'id')})
            if util.has_rows(s4):
                s35 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE
                    `users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities
                    ', 'user_id')})
                outputs.extend(util.get_data(s35, 'users', 'registration_status'))
                s36 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
                outputs.extend(util.get_data(s36, 'users', 'id'))
                outputs.extend(util.get_data(s36, 'users', 'email'))
                outputs.extend(util.get_data(s36, 'users', 'first_name'))
                outputs.extend(util.get_data(s36, 'users', 'last_name'))
                outputs.extend(util.get_data(s36, 'users', 'username'))
                outputs.extend(util.get_data(s36, 'users', 'registration_status'))
            else:
                s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
                outputs.extend(util.get_data(s5, 'users', 'id'))
                outputs.extend(util.get_data(s5, 'users', 'email'))
                outputs.extend(util.get_data(s5, 'users', 'first_name'))
                outputs.extend(util.get_data(s5, 'users', 'last_name'))
                outputs.extend(util.get_data(s5, 'users', 'username'))
                outputs.extend(util.get_data(s5, 'users', 'registration_status'))
        else:
            s22 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
            outputs.extend(util.get_data(s22, 'users', 'id'))
            outputs.extend(util.get_data(s22, 'users', 'email'))
            outputs.extend(util.get_data(s22, 'users', 'first_name'))
            outputs.extend(util.get_data(s22, 'users', 'last_name'))
            outputs.extend(util.get_data(s22, 'users', 'username'))
            outputs.extend(util.get_data(s22, 'users', 'registration_status'))
    else:
        pass
    return outputs
```

B.6 Kandan Chat Room Command get_users

```
def get_users (conn, inputs):
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`username` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'users', 'id'))
    outputs.extend(util.get_data(s0, 'users', 'email'))
    outputs.extend(util.get_data(s0, 'users', 'first_name'))
    outputs.extend(util.get_data(s0, 'users', 'last_name'))
    outputs.extend(util.get_data(s0, 'users', 'username'))
    outputs.extend(util.get_data(s0, 'users', 'registration_status'))
    if util.has_rows(s0):
        s8 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        outputs.extend(util.get_data(s8, 'users', 'id'))
        outputs.extend(util.get_data(s8, 'users', 'email'))
        outputs.extend(util.get_data(s8, 'users', 'first_name'))
        outputs.extend(util.get_data(s8, 'users', 'last_name'))
        outputs.extend(util.get_data(s8, 'users', 'username'))
        outputs.extend(util.get_data(s8, 'users', 'registration_status'))
        s9 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s9):
            s10 = util.do_sql(conn, "SELECT `activities`.* FROM `activities`
                WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(
                s9, 'channels', 'id')})
            if util.has_rows(s10):
                s63 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE
                    `users`.`id` IN (:x0)", {'x0': util.get_one_data(s10, '
                    activities', 'user_id')})
                outputs.extend(util.get_data(s63, 'users', 'id'))
                outputs.extend(util.get_data(s63, 'users', 'email'))
                outputs.extend(util.get_data(s63, 'users', 'first_name'))
                outputs.extend(util.get_data(s63, 'users', 'last_name'))
                outputs.extend(util.get_data(s63, 'users', 'gravatar_hash'))
                outputs.extend(util.get_data(s63, 'users', 'username'))
                outputs.extend(util.get_data(s63, 'users', '
                    registration_status'))
                s64 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE
                     `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s8,
                     'users', 'id')})
                outputs.extend(util.get_data(s64, 'users', 'id'))
                outputs.extend(util.get_data(s64, 'users', 'email'))
                outputs.extend(util.get_data(s64, 'users', 'first_name'))
                outputs.extend(util.get_data(s64, 'users', 'last_name'))
                outputs.extend(util.get_data(s64, 'users', 'username'))
                outputs.extend(util.get_data(s64, 'users', '
                    registration_status'))
                s65 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE
                    (1=1)", {})
                outputs.extend(util.get_data(s65, 'users', 'id'))
                outputs.extend(util.get_data(s65, 'users', 'email'))
                outputs.extend(util.get_data(s65, 'users', 'first_name'))
                outputs.extend(util.get_data(s65, 'users', 'last_name'))
                outputs.extend(util.get_data(s65, 'users', 'username'))
                outputs.extend(util.get_data(s65, 'users', '
                    registration_status'))
            else:
                s11 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE
                     `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s8,
                     'users', 'id')})
                outputs.extend(util.get_data(s11, 'users', 'id'))
                outputs.extend(util.get_data(s11, 'users', 'email'))
                outputs.extend(util.get_data(s11, 'users', 'first_name'))
                outputs.extend(util.get_data(s11, 'users', 'last_name'))
                outputs.extend(util.get_data(s11, 'users', 'username'))
                outputs.extend(util.get_data(s11, 'users', 'registration_status'))
                s12 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE (1=1)", {})
                outputs.extend(util.get_data(s12, 'users', 'id'))
                outputs.extend(util.get_data(s12, 'users', 'email'))
                outputs.extend(util.get_data(s12, 'users', 'first_name'))
                outputs.extend(util.get_data(s12, 'users', 'last_name'))
                outputs.extend(util.get_data(s12, 'users', 'username'))
                outputs.extend(util.get_data(s12, 'users', 'registration_status'))
        else:
            s36 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s8, 'users', 'id')})
            outputs.extend(util.get_data(s36, 'users', 'id'))
            outputs.extend(util.get_data(s36, 'users', 'email'))
            outputs.extend(util.get_data(s36, 'users', 'first_name'))
            outputs.extend(util.get_data(s36, 'users', 'last_name'))
            outputs.extend(util.get_data(s36, 'users', 'username'))
            outputs.extend(util.get_data(s36, 'users', 'registration_status'))
            s37 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE (1=1)", {})
            outputs.extend(util.get_data(s37, 'users', 'id'))
            outputs.extend(util.get_data(s37, 'users', 'email'))
            outputs.extend(util.get_data(s37, 'users', 'first_name'))
            outputs.extend(util.get_data(s37, 'users', 'last_name'))
            outputs.extend(util.get_data(s37, 'users', 'username'))
            outputs.extend(util.get_data(s37, 'users', 'registration_status'))
    else:
        pass
    return outputs
```

B.7 Kandan Chat Room Command get_users_id

```
def get_users_id (conn, inputs):
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`username` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'users', 'id'))
    outputs.extend(util.get_data(s0, 'users', 'email'))
    outputs.extend(util.get_data(s0, 'users', 'first_name'))
    outputs.extend(util.get_data(s0, 'users', 'last_name'))
    outputs.extend(util.get_data(s0, 'users', 'username'))
    outputs.extend(util.get_data(s0, 'users', 'registration_status'))
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        outputs.extend(util.get_data(s2, 'users', 'id'))
        outputs.extend(util.get_data(s2, 'users', 'email'))
        outputs.extend(util.get_data(s2, 'users', 'first_name'))
        outputs.extend(util.get_data(s2, 'users', 'last_name'))
        outputs.extend(util.get_data(s2, 'users', 'username'))
        outputs.extend(util.get_data(s2, 'users', 'registration_status'))
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities` WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(s3, 'channels', 'id')})
            if util.has_rows(s4):
                s35 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE `users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities', 'user_id')})
                outputs.extend(util.get_data(s35, 'users', 'registration_status'))
                s36 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE  `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
                outputs.extend(util.get_data(s36, 'users', 'id'))
                outputs.extend(util.get_data(s36, 'users', 'email'))
                outputs.extend(util.get_data(s36, 'users', 'first_name'))
                outputs.extend(util.get_data(s36, 'users', 'last_name'))
                outputs.extend(util.get_data(s36, 'users', 'username'))
                outputs.extend(util.get_data(s36, 'users', 'registration_status'))
            else:
                s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
                outputs.extend(util.get_data(s5, 'users', 'id'))
                outputs.extend(util.get_data(s5, 'users', 'email'))
                outputs.extend(util.get_data(s5, 'users', 'first_name'))
                outputs.extend(util.get_data(s5, 'users', 'last_name'))
                outputs.extend(util.get_data(s5, 'users', 'username'))
                outputs.extend(util.get_data(s5, 'users', 'registration_status'))
        else:
            s22 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users', 'id')})
            outputs.extend(util.get_data(s22, 'users', 'id'))
            outputs.extend(util.get_data(s22, 'users', 'email'))
            outputs.extend(util.get_data(s22, 'users', 'first_name'))
            outputs.extend(util.get_data(s22, 'users', 'last_name'))
            outputs.extend(util.get_data(s22, 'users', 'username'))
            outputs.extend(util.get_data(s22, 'users', 'registration_status'))
    else:
        pass
    return outputs
```

C APPENDIX: REGENERATED CODE FOR BLOG APPLICATION

C.1 Blog Application Command get_articles

```
def get_articles (conn, inputs):
    outputs = []
    s0 = util.do_sql(conn, "SELECT `articles`.* FROM `articles`", {})
    outputs.extend(util.get_data(s0, 'articles', 'id'))
    outputs.extend(util.get_data(s0, 'articles', 'title'))
    outputs.extend(util.get_data(s0, 'articles', 'text'))
    s1 = util.do_sql(conn, "SELECT `articles`.* FROM `articles`", {})
    outputs.extend(util.get_data(s1, 'articles', 'id'))
    outputs.extend(util.get_data(s1, 'articles', 'title'))
    outputs.extend(util.get_data(s1, 'articles', 'text'))
    return outputs
```

C.2 Blog Application Command get_article_id

```
def get_article_id (conn, inputs):
    outputs = []
    s0 = util.do_sql(conn, "SELECT `articles`.* FROM `articles`", {})
    s1 = util.do_sql(conn, "SELECT  `articles`.* FROM `articles` WHERE `
        articles`.`id` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s1, 'articles', 'id'))
    outputs.extend(util.get_data(s1, 'articles', 'title'))
    outputs.extend(util.get_data(s1, 'articles', 'text'))
    if util.has_rows(s1):
        s9 = util.do_sql(conn, "SELECT `comments`.* FROM `comments` WHERE `
            comments`.`article_id` = :x0", {'x0': util.get_one_data(s1, '
            articles', 'id')})
        outputs.extend(util.get_data(s9, 'comments', 'commenter'))
        outputs.extend(util.get_data(s9, 'comments', 'body'))
        outputs.extend(util.get_data(s9, 'comments', 'article_id'))
    else:
        pass
    return outputs
```

D APPENDIX: REGENERATED CODE FOR STUDENT REGISTRATION SYSTEM

D.1 Student Registration System Command liststudentcourses

```
def liststudentcourses (conn, inputs):
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM student WHERE id = :x0", {'x0':
        inputs[0]})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT * FROM student WHERE id=:x0 AND
            password=:x1", {'x0': util.get_one_data(s0, 'student', 'id'), 'x1'
            : inputs[1]})
        if util.has_rows(s2):
            s6 = util.do_sql(conn, "SELECT * FROM course c JOIN registration r
                 on r.course_id = c.id WHERE r.student_id = :x0", {'x0': util.
                get_one_data(s2, 'student', 'id')})
            outputs.extend(util.get_data(s6, 'course', 'id'))
            outputs.extend(util.get_data(s6, 'course', 'teacher_id'))
            outputs.extend(util.get_data(s6, 'registration', 'course_id'))
            if util.has_rows(s6):
                s6_all = s6
                for s6 in s6_all:
                    s19 = util.do_sql(conn, "Select firstname, lastname from teacher where id = :x0", {'x0': util.get_one_data(s6, 'course', 'teacher_id')})
                    s20 = util.do_sql(conn, "SELECT count(*) FROM registration WHERE course_id = :x0", {'x0': util.get_one_data(s6, 'registration', 'course_id')})
                s6 = s6_all
            else:
                pass
        else:
            pass
    else:
        pass
    return outputs
```

E APPENDIX: INFERRED DSL FOR STUDENT REGISTRATION SYSTEM

E.1 Student Registration System Command `liststudentcourses`

```
d1 <- select student.id, student.password, student.firstname, student.lastname
      where student.id = in_s
if d1 {
  d2 <- select student.id, student.password, student.firstname, student.lastname
        where student.id = [in_s, d1.student.id] AND student.password = in_p
  if d2 {
    d3 <- select course.id (output: True), course.name, course.course_number, course.size_limit, course.is_offered, course.teacher_id (output: True), registration.student_id, registration.course_id (output: True)
          where registration.course_id = course.id AND registration.student_id = [in_1, d1.student.id, d2.student.id]
    if d3 {
      for each row3 in d3 {
        d4 <- select teacher.firstname, teacher.lastname
              where teacher.id = row3.teacher_id
        d5 <- select count(registration)
              where registration.course_id = [row3.course.id, row3.registratoin.course_id]
      }
    } else {}
  } else {}
} else {}
```