Active Learning for Software Engineering

José P. Cambronero
CSAIL MIT
USA
jcamsan@mit.edu

Thurston H. Y. Dang
CSAIL MIT
USA
tdang@mit.edu

Nikos Vasilakis∗
CSAIL MIT
USA
nikos@vasilaki.is

Jiasi Shen
CSAIL MIT
USA
jiiasi@csail.mit.edu

Jerry Wu†
Google
USA
werryju@mit.edu

Martin Rinard
CSAIL MIT
USA
rinard@csail.mit.edu

Abstract
Software applications have grown increasingly complex to deliver the features desired by users. Software modularity has been used as a way to mitigate the costs of developing such complex software. Active learning-based program inference provides an elegant framework that exploits this modularity to tackle development correctness, performance and cost in large applications. Inferred programs can be used for many purposes, including generation of secure code, code re-use through automatic encapsulation, adaptation to new platforms or languages, and optimization. We show through detailed examples how our approach can infer three modules in a representative application. Finally, we outline the broader paradigm and open research questions.

CCS Concepts • Software and its engineering → Software development techniques.

Keywords program inference, program modeling, active learning

1 Introduction
Software engineering as practiced today is a laborious and error-prone process. Programmers are faced with large, poorly documented (and hard to inspect) systems. This makes working with such codebases an unwieldy exercise, negatively impacting correctness, runtime performance, and development cost.

We propose an alternative paradigm: using active learning to automatically infer program functionality. By feeding a program carefully crafted inputs and observing its behavior, we can capture the program’s core semantics in a form amenable to analysis, augmentation, and engineering.

1.1 Dimensions
Our work aims to improve three key dimensions of software engineering artifacts: correctness, runtime performance, and development cost.

Correctness: Inferring the semantics of an application from its execution facilitates creating an inferred program that satisfies the de facto specification [8], which reflects the behavior experienced and expected by its end-users. Depending on the application and our inference process, the inferred program will not identically replicate the original application. However, this behavior can be an advantage. For example, an application could be hard-coded to only process images of a fixed width and height, while the inferred application can be generalized to varying sizes. Security can be considered a special case of correctness. Vulnerabilities such as buffer overflows can be exploited to corrupt the data or control-flow of the program, causing unintended behavior. Our framework can use a subset of inputs to reduce the attack surface and the regenerated code can be designed to be safe-by-construction.

Runtime Performance: Our systems infer the functionality of a program, divorced from its underlying implementation or language. This frees our techniques from the constraints of legacy codebases, allowing generation of new code that can be highly optimized by the latest compilers. Moreover, in principle, our inference systems can perform more fundamental optimizations at the algorithmic level. For
example, consider a legacy binary that performs bubble sort. An inference system that only observes the active behavior of the program could determine that a sorting operation was performed, and emit code that uses a more efficient sorting algorithm, such as mergesort.

**Development Cost:** Software engineering with active learning can help bridge the gap between the intent of the programmer and the concrete instantiation of their vision. By viewing the active behavior as the specification, our paradigm allows the programmer to code their system in any language and then have its semantics extracted and automatically re-expressed in any other target language.

### 1.3 Structure of the Paper

We begin by introducing key background concepts in active learning and program inference (Section 2). We then outline our paradigm by considering a social network application (Section 3), and show how to infer a small, but diverse, set of its components using three different techniques and systems (Section 4–6):

- Section 4 presents a technique to infer and regenerate binary data parsers, similar to the one needed by the social network application.
- Section 5 discusses Shen and Rinard’s [73] technique for inferring programs that use databases, and how this can be applied to the account management back-end of our social network application.
- Section 6 presents a technique for upgrading a program that uses in-memory data structures such as maps and lists to one that uses a database. This upgrade is done without changing the original program to use an external framework, such as one for Object-Relational Mapping (ORM).

These example components illustrate several benefits of our proposed approach. For example, parsers are a significant source of security vulnerabilities in programs [53, 61]; and database interfaces are known sources of development and maintenance cost. Our regenerated programs can be safe-by-construction—automatically augmented with security checks—while significantly simplifying the development process.

We present the general paradigm of active learning for software engineering (Section 7), including design choices for the observation, inference and generation models, discuss research challenges and threats to validity (Section 8), and compare with prior work (Section 9).

### 2 Background

We introduce background concepts employed throughout the remainder of the paper. We outline program inference, active learning broadly and its application to program inference.

#### 2.1 Program Inference

We define program inference as the task of modeling a computation by repeatedly interacting with, and observing the behavior of, an existing implementation of that computation. The inference technique interacts with the existing implementation by executing a command, i.e., a procedure or function, that has been exposed by the implementation’s API (e.g., the main function of a program written in C).

A hypothesis corresponds to a possible model of that computation (e.g., a partial program), which is said to be consistent with an execution if the execution’s behavior under the model matches the behavior in the original program. We say there is inference ambiguity when there are two or more hypotheses that are consistent with the implementation’s observed executions and may not be equivalent to each other. Absent any additional information, either hypothesis could represent the target computation. To resolve this ambiguity we may attempt to induce an execution that can rule out competing hypotheses.

A crucial difference between program inference and other automated programming techniques, such as inductive program synthesis [62], is that program inference assumes an implementation of the desired computation already exists and is available during inference. This reference implementation eliminates the need for examples or a formal specification.

#### 2.2 Active Learning

Active learning refers to a setting in which a learning algorithm is free to query an oracle or environment to obtain additional information for the learning task [69]; often this information comes in the form of ground-truth labels associated with examples. For example, a system may request labels for examples with low-confidence predictions [46] or
we use a modern social network application to illustrate the challenges of active learning at the scale of a production-grade application (Section 3.1), explain how module-level decomposition can make such active learning tractable (Section 3.2), and showcase an end-to-end example of learning and regeneration (Section 3.3).

### 2.3 Program Inference and Active Learning

As discussed previously, a core challenge in program inference is to resolve the ambiguity between possible hypotheses. Rather than depending on a fixed corpus of inputs, program inference has the advantage of having access to a “cheap” oracle: the existing implementation. This oracle can be used to actively “label” different inputs with their corresponding ground-truth behavior. Furthermore, the inference technique can actively construct inputs that are devised to increase the likelihood that the hypothesized model can be refined. Active learning, thus, provides a grounded approach to exploring and refining the inferred program.

Figure 1 provides a schematic of this high-level algorithm, which motivates the rest of our paper. The program inference technique consists of a core inference algorithm, which incorporates execution feedback from the existing implementation, and a method for acquiring/generating inputs to improve the hypothesis. In the following sections we discuss three different techniques that instantiate this higher-level algorithm to infer different types of programs.

For reference purposes, Figure 2 presents a summary of key terms introduced.

### 3 Active Learning at Scale

We use a modern social network application to illustrate the challenges of active learning at the scale of a production-grade application (Section 3.1), explain how module-level decomposition can make such active learning tractable (Section 3.2), and showcase an end-to-end example of learning and regeneration (Section 3.3).

![Fig. 1. Active learning in program inference. At a high-level, the program inference technique incorporates execution feedback from the existing program implementation and then actively picks new inputs to refine the hypothesized model of the underlying computation.](image)

**Fig. 1.** Active learning in program inference. At a high-level, the program inference technique incorporates execution feedback from the existing program implementation and then actively picks new inputs to refine the hypothesized model of the underlying computation.

**Fig. 2.** Glossary. A summary of key background terms.

**Figure 2 provides a schematic of this high-level algorithm, which motivates the rest of our paper.** The program inference technique consists of a core inference algorithm, which incorporates execution feedback from the existing implementation, and a method for acquiring/generating inputs to improve the hypothesis. In the following sections we discuss three different techniques that instantiate this higher-level algorithm to infer different types of programs.

For reference purposes, Figure 2 presents a summary of key terms introduced.

### 3.1 Challenges and Opportunities

Consider a modern social network application like Facebook. Such an application must provide a dizzying array of functionality—e.g., user accounts, news feeds, photos, group events, search queries—accounting for millions of lines of code. For example, Facebook’s source code in 2011 totaled more than nine million lines [67], spanning different logical subsystems and semantic levels, and supported by thousands of engineers (and open-source backing). How could active learning infer and regenerate such an application?

A key insight is that modern applications, even when seemingly monolithic, are highly modular—just the front-end code of the aforementioned Facebook example was over 10K modules in 2011 [67]. Software modules are development-time constructs that abstract away inessential details and present well-encapsulated interfaces to their consumers. They provide implicit, fine-grained guidelines for breaking down the complexity of active learning.

Pervasive modularization is a relatively recent phenomenon. Recent research [80, 89] shows that today’s software is primarily composed of third-party modules, with large applications numbering in the thousands to tens of thousands of modules. Even more impressive are the statistics of language-specific package repositories. JavaScript hosts more than 1M packages from over 100K authors and serves billions of package downloads per day [68]. Such extensive modularity, the use of encapsulation, and pervasive reuse of code in today’s applications translates to several opportunities for active learning:

- **Divide and Conquer:** Recursive module imports simplify active learning and regeneration, as complex modules are
composed by many simpler ones—to the point that leaf-level, elementary modules can be inferred directly.

- **Interface Reuse**: Module re-use within a single application translates to observing active program behavior in multiple settings, significantly aiding active learning.

- **Fine Granularity**: Multi-thousand-module dependency trees enable visibility into an application’s behavior at a high resolution in time and space: (i) at every function call entering a module, and (ii) on thousands of modules across an application.

- **Clear Boundaries**: Modules today encapsulate state behind small and tight interfaces, with clear self-contained components. This structure simplifies capturing every observable behavior, and lowers the cost of interacting with the target program.

- **Parallel Learning**: Module state at the same level in a dependency tree is independent, allowing parallel inference and regeneration.

The next section returns to the social-network application, illustrating one possible way of putting these ideas into practice.

### 3.2 Breaking Down into Modules

We start by unfolding a complex application or module into smaller modules (top-down), so as to leverage recursive active learning and regeneration. Inferred modules are then composed into larger modules, leading to regeneration of the entire application.

Figure 3 presents an overview of this process using the (simplified) module structure of a social network application. Vertices represent modules. Edges depict their interdependencies: parent (caller) modules importing child (callee, consumer) modules. Dotted edges imply multiple levels of depth, and thicker vertices highlight modules used as detailed examples of active learning in the rest of the paper. Features such as user accounts are implemented as separate modules that draw their functionality from other modules further down in the dependency tree. For example, Photos imports functionality to parse and compress images.

A first step towards module-level active learning is to interpose at the module boundaries. Such interposition can be achieved automatically, for example, by detecting and transforming module interfaces. The specifics of these transformations depend on the language setting, with compiled and dynamic languages providing appropriate mechanisms (e.g., Template Haskell, Rust’s macro system, name (re-)binding, and dynamic code evaluation).

In both language settings, a module’s return value is transformed to automatically capture active program behavior, in this case in the form of input-output pairs. This transformation is structured similar to Figure 4: the module interface is augmented to pass call arguments and return values to the active learning infrastructure (alr). (The complexity of transformation stems from the fact that a module return value can be an object of arbitrary nesting.) The active learning infrastructure could (i) generate a preliminary template for the module, and (ii) instantiate the correct subsystem—e.g., Nero (Section 6), Konure (Section 5), etc.

Further transformations (not detailed) aim to detect module accesses to their surrounding environment—including accesses to global variables, top-level objects, etc.—and capture a module’s full observable behavior. One way to achieve this is by shadowing and transforming free variables in the module environment during the module-loading process [79, 81].

Once the learning and regeneration completes, the decomposed structure can be used to replace the module at runtime. To achieve this, the active learning subsystem sends a notification to the transformation system that a module has been generated. The process could overwrite the pointer to the wrapper-internal module so that (i) the consumer modules remain intact, and (ii) the system keeps feeding the active learning subsystem with data from subsequent calls.

### 3.3 A Teensy Module: String Padding

To further understand these transformations, we zoom into the Format module within the larger social networking application. The interaction between Format and one of its
The wrapper observes that the function is always provided with two arguments of strings, and the second argument alters the padding character. Replacing padding with a third argument modifies the behavior, in this case captured through input-output pairs, the inference algorithm is able to detect that the first argument is always the suffix of the result, the second argument denotes the spaces prefixing the string, and that the third argument alters the padding character. Replacing Padding with, say, the recently ES7-standardized `padEnd` method would improve both the performance and the security of the larger application.

The `Format`–Padding duo is a trivial case—their implementation code amounts to less than 50 lines—used to illustrate the application of transformations and tangible benefits of regeneration. The next three sections walk through the active learning process of modules that are significantly more complex. These are highlighted in Figure 3: a binary data parser for photos (Section 4), a module that accesses the database of users’ posts (Section 5), and a stateful `seed` for a new module with lists and maps (Section 6).

### 4 Inferring Binary Data Parsers

The social networking application, introduced in Section 3, provides image sharing functionality. These images are stored as binary data, with parsing details that vary by image type and their end-use. The social network application must validate input and populate key data structures, defined across different functions, with values from the input.

Writing parsers and data processors is a notoriously difficult software engineering task [11, 12, 29, 72]. Often this logic is scattered throughout a program, yielding what is commonly termed a “shotgun” parser, a known source of security vulnerabilities [53]. Other times, the input validation performed is insufficient to remove vulnerabilities [61].

Our active learning approach provides a framework to infer the original parsing functionality from the application and a collection of inputs. In particular, by exploring the data structure definitions produced by different inputs, we can build up a generalization for these definitions, which represents the desired parsing functionality.

#### 4.1 Image Sharing Service

The image service uses a library, `stb`, to load bitmap (.BMP) images. The application calls the `stbi__bmp_load` function defined in `stb_image.h`. `stbi__bmp_load` parses the image file header using `stbi__bmp_parse_header` and then parses the remainder of the file using `stbi__bmp_load_cont`. `stbi__bmp_load_cont` populates two struct pointers and three integer pointers, and returns a pointer to an array containing the image pixel data.

Figure 6 shows a (simplified) excerpt of code from the `stbi__bmp_load_cont` function that highlights some of the parsing challenges. The code reads the image’s pixel data and stores it in an output array. To do so, the code must branch between different parsing definitions, extract the appropriate number of bytes (and keep track of the file pointer) based on the image’s rows and columns, cast these bytes to the correct type, store them in the correct position in the output array, and keep track of certain image properties (in this case, the use of the alpha channel).

We now walk through the steps that an active learning technique can use to infer this parsing functionality. We implemented these steps in a prototype system called `BDP` (short for Binary Data Parser).

#### 4.2 Approach

Figure 7 presents a block diagram outlining our technique. We begin with a set of example images. The original image parsing module, depicted in gray with bold dotted edges, takes input files (images), reads their data and populates the data structures used later on by the compression function. Our technique, depicted in blue boxes with dashed outlines, layers on additional components to extract behavioral information from this execution and recover patterns that yield the parsing function. We walk through each step in the pipeline and its use in the broader technique.

**Dynamic Analysis:** Each input file is executed through the parsing function. The application has been modified with dynamic instrumentation tools to produce execution traces. These traces contain concrete expressions relating the input and output data structure bytes. After executing each input file, we have a collection of such traces.

Our technique relies on observing a large space of input examples and their corresponding execution traces. However, rather than randomly acquiring new examples, the inference procedure guides the choice of examples that provide new information by choosing input files that cannot be parsed.
for (j=0; j < (int) s->img_y; ++j) {
    // Branch between parsing cases.
    if (easy) { // No bitpacking
        for (i=0; i < (int) s->img_x; ++i) {
            out[z+2] = read8Bits(s); // blue
            out[z+1] = read8Bits(s); // green
            out[z+0] = read8Bits(s); // red
            z += 3;
            // Optional alpha channel
            unsigned char alpha = (easy == 2 ? out[2] : 255);
            all_alpha |= alpha;
            if (out_bytes_per_pixel == 4) out[z++] = alpha;
        }
    } else { // Bitpacking e.g., 16-bit images.
        // Use header information
        int in_bytes_per_pixel = info->bpp;
        for (i=0; i < (int) s->img_x; ++i) {
            // Decode appropriate number of bytes
            uint32 pixel = (in_bytes_per_pixel == 16
            ? read16Bits(s)
            : read32Bits(s));
            unsigned int a;
            // shiftsigned extracts the bits corresponding to the red/green/blue
            // channels, and scales it to between [0,255]
            out[z++] = shiftsigned(pixel & mask_red,
                red_shift, red_count);
            out[z++] = shiftsigned(pixel & mask_green,
                green_shift, green_count);
            out[z++] = shiftsigned(pixel & mask_blue,
                blue_shift, blue_count);
            alpha = (mask_alpha
                ? shiftsigned(v & mask_alpha,
                    alpha_shift, alpha_count)
                : 255);
            // Track properties of input read so far
            all_alpha |= alpha;
            if (out_bytes_per_pixel == 4) out[z++] = alpha;
        }
        stbi__skip(s, pad);
    }
    return (out);
}

Fig. 6. Simplified excerpt from stbi_bmp_load_cont, which reads in
the image’s pixel data into an array. This requires that the developer
distinguish between different parsing cases, read at the appropriate file offset,
cast sequences of bytes into the appropriate type, and store these in the
correct order in the output array.

with the current inferred model, and input files that match
conditions for which we have few existing examples.

Domain-Specific Language: Effective inference, across
the techniques we present in this paper, relies on having a
productive representation of the input and the application
behavior we want to capture. As shown in this and other
techniques here, carefully designing a domain-specific lan-
guage (DSL) that enables expression of important program
features and appropriately constrained generalization is a
key technical contribution.

BDP uses such a DSL, which describes sequences of mem-
ory and maps input/output bytes, to describe the data
structures built up by the target application. This DSL abstracts
away implementation details such as field names and allows
us to implement a simple inference procedure. We refer to
programs in our DSL as Data structure Definition Programs
/DDPs/.

For example, the following DDP can describe a struct with
two integer fields that are read directly from the input file
out[0] = ZeroExtend(64, Read(0, 4))
out[1] = ZeroExtend(64, Read(4, 8))
where the word at index 0 is mapped to an expression that
reads an int stored in bytes 0 through 3 (inclusive) and ex-
tends it to a 64-bit word, similarly for index 1.

Single Input Inference: The DDPs produced by executing
the application and analyzing its traces can then be used to
infer common properties of the input files and how they are parsed
by the original module.

We distinguish between properties that can be inferred
on the basis of each individual DDP and properties that can
only be resolved when multiple DDPs are considered.

For this parsing function, we can infer repetition in the
way groups of bytes are defined, which must account for the
strides in the data structure that is being populated and the
strides in the original input file. Figure 8 shows the layout of
pixel data in the array populated by the parsing function and
compares this with the layout for that data in the original
input file. Bitmaps are commonly stored "bottom-up" (upside-
down), with the file starting with the bottom row of the
image. The output array, however, is "top-down", so it is
necessary to iterate in the zig-zag pattern shown with dashed
dlines. Additionally, while the input file has color channel
information laid out as blue–green–red–alpha (BGRA), this
must be laid out as "RGBA" in the output array. Our inference
Choosing input files for different ways, such as the smallest file or the file associated with a high-error partition.

**Code Generation:** Our model DSL can define the contents of data structures in terms of input bytes, and provides a limited set of control flow operations, such as bounded loops (with bounds determined by concrete values or values based on input file bytes) and branching on an equality expression over the values of particular input file bytes (i.e., parsing flags). This language is used to define a set of parsers that can handle different input types, which are then grouped into a single parser by defining branching conditions that lead to each sub-parser.

During the code generation phase, we compile this DSL program into a C program using a simple translation approach, as every construct in the DSL maps in a straightforward fashion to constructs in popular imperative languages.

Figure 9 shows an excerpt of the C code generated to populate the pixel data array for a 16×16 32-bit bottom-up bitmap.

The generated program can now replace the original parsing function in the application. The code generated has multiple advantages over the original parsing function: it is more compact, the mapping between input bytes and data structure contents is more explicit, and it uses simple array expressions instead of pointer arithmetic.

**Evaluation:** We implemented the technique presented here in a system: BDP. We have successfully generated parsing functions using BDP for two popular image applications: cating and ptrace.

**User Manual Input:** Currently, to use BDP, the user must provide a corpus of inputs that can be used to exercise the application. The active learning process selects files to run from this corpus during inference. Additionally, the user must indicate the function (or functions) in the target application that build the data structures of interest to facilitate their instrumentation. Finally, the user must provide BDP with the command used to execute the application (and which BDP will repeatedly call with different input files).

## 5 Inheriting Modules that Access Databases

Many applications access a database as an integral part of their execution. Databases provide advantages such as optimized data retrieval and storage, and abstraction of the underlying data structures and their layout. However, developing a database-backed module presents challenges: the core functionality must be mapped to database operations, database queries and inputs must be appropriately sanitized to reduce security vulnerabilities [70], and the developer must determine the appropriate database system to back their application.

Shen and Rinard [73] addressed this challenge with their Konure system, which observes the active behavior of the
```c
int64_t expr_0 (int inputStartIndices [1], long file) {
    int8_t INPUT_0;
    readBytes (&INPUT_0, file, inputStartIndices [0], 1);
    // (assert (let ((?x0 ((_ sign_extend 56) INPUT_0)))
    // (= __dfsan_top_level ?x0)))
    int8_t var3003 = INPUT_0;
    int64_t var3002 = ((int8_t) var3003); // sign_extend
    int64_t INTERNAL_VAR0 = var3002; // let clause
    return (INTERNAL_VAR0);
}
```
Monitor Communications: Konure takes a greybox approach — monitoring a database proxy — to observe the application’s execution. For each command execution, Konure observes the inputs, the outputs, and the application’s communications with its database. These interactions are transparent to the original application.

Konure initially executes the command with parameters: listuserposts -u 0 -p 1. Future executions contain values that allow Konure to explore the application. The command is executed on an initially empty database. When it executes, Konure’s database proxy records the SQL query SELECT * FROM users WHERE id = ‘0’, as well as the fact that the query retrieves no data. This information constitutes the application’s concrete trace for this command execution.

Domain-Specific Language: Konure uses a carefully designed domain-specific language to represent applications in a constrained search space and enable efficient inference. The DSL can capture sequences of queries (which may reference prior queries’ results), conditional statements which execute based on a test condition for non-empty query results, and iteration over query results in the form of bounded loops.

During inference, Konure maintains a hypothesized program, expressed in this DSL, which is derived to be consistent with all observed traces. In this case, having observed the query SELECT * FROM users WHERE id = ‘0’, Konure hypothesizes that this query could either be a query followed by a yet-to-be derived subprogram of additional statements or be part of a conditional statement’s guard. The latter would correspond to the following program hypothesis:

\[
\text{if } y1 \text{ <- select users.id, users.password, users.firstname, users.lastname where users.id = u} \text{ then } \{ ?? \} \text{ else } \{ ?? \}
\]

where each “??” is a yet-to-be-explored subprogram, or a DSL non-terminal. Note that the concrete value ‘0’ has been replaced with a data origin (the input parameter u), and the columns for users are explicitly included in the select statement, rather than using the * wildcard. Konure identifies such rewrites using a trace abstraction procedure.

Trace Rewriting: In addition to abstracting out concrete values of a trace with their data origins, Konure also rewrites the trace to summarize loop information. For example, if a concrete trace contains the same query, repeated multiple times, then Konure may hypothesize that the query is executed as part of a loop body.

Inference Algorithm: Konure’s hypotheses contain unexplored subprograms (DSL non-terminals), marked ??, After executing the command a few times, Konure expands a ?? with the appropriate DSL production that is consistent with the execution traces. If this expansion produces more ??s, Konure executes the command more times and expands each ?? recursively. This recursive, hierarchical expansion continues until all non-terminals have been expanded and the final program has been identified.

For any program that conforms to the DSL, the algorithm terminates and produces a program that correctly models the original functionality.

New Inputs: To exercise unexplored subprograms and resolve inference ambiguity, Konure generates input values and database contents that induce different executions. In our user posting example, Konure produces values such that the select query from the users table does not produce empty results. In this case, that corresponds to providing an input parameter for user ID (u) that has a corresponding entry in the users table. In the hypothesis that contains the conditional statement, this would induce execution of the then branch (which was previously unexplored).

Konure selects these values by generating suitable constraints and solving them with an SMT solver.

Generating Code: Inference terminates once all non-terminals in the hypothesized program are expanded, indicating that no unexplored paths remain. Konure’s inferred program can express database queries, bounded loops over query results, and conditional statements based on non-empty query results. These DSL constructs map straightforwardly to constructs in popular imperative languages. Konure currently generates Python code.

Evaluation: Konure has been evaluated on five applications, including a chat room, task manager, blogging application, and student registration [73], which have been written in multiple programming languages, including Java and Ruby on Rails. Konure inferred a subset of their commands, which contain SQL queries, conditionals, loops, and outputs.

User Manual Input: To use Konure the user provides the necessary command and data specifications. The command
specification identifies the API command to be inferred, and the data specification provides information on the database schema. Konure uses these to automate its exploration of the application’s behavior and infer a model for it.

6 Automatically Replacing In-Memory Data Structures with Databases

As discussed in Section 5, developing a module that interacts with a database to access and persist data has many advantages, but also implementation challenges.

In this example, we show how program inference can infer a database replacement for existing in-memory data structures in a program. This approach allows developers to write a program in a plain imperative language, such as Python, using familiar, standard programming constructs and data structures (i.e., no need to develop a database model or use database-related frameworks). The approach then infers the program and regenerates a new program that accesses data using a database (instead of the original data structures).

A key distinction here is that while our previous example demonstrated how Konure infers the use of an existing database in an imperative program, this example (and the corresponding system) focuses on a program that does not yet use a database but rather performs computations that can be modeled and replaced with a database.

We introduce Nero, an active learning system we developed to infer the use of standard in-memory data structures (e.g., lists and dictionaries) in a program, model their use with database queries, and replace the in-memory data structures with a database in a regenerated program [84].

This approach presents multiple advantages: the developer need not be familiar with databases, and similarly to the case seen in Section 5, the module’s development does not become tied to a particular database system, as the regenerated code can be adapted to other database systems.

6.1 A New User Post Querying Service

A developer has been tasked with writing a new variant of the module associated with users’ posting history (Section 5). Figure 12 presents the module code written by the developer for this new version.

The module uses a list, users, to track each user’s first/last names, and uses the corresponding index position as an id. Post information is stored in the post_contents dictionary, which maps a post title to the corresponding post contents (a string). The application uses another list, posts, to track tuples of user ID and post titles, representing users’ posts.

Nero refers to the code written by the developer as a seed program, which satisfies certain program properties that enable efficient inference. In this case, the code has two functions: do_post and do_listuserposts. Function do_post takes a user ID, a string representing the post contents, and a post title. The user ID is used to query the users list, retrieving and printing the user’s name. The function then registers the user’s post by appending the user ID and the post title to the posts list. Finally, the function inserts the corresponding post contents (post_str) into a dictionary (post_contents) using the post title as a key. Function do_listuserposts takes as input a user ID, queries for the user’s name and prints it, and then iterates over the posts list to identify the user’s posts and prints out the corresponding post title and contents.

6.2 Approach

Nero treats a program’s control logic and data as two separate components. This conceptual separation often occurs in data-intensive programs. Nero uses data accesses to infer the program’s control logic. To do so, Nero carefully chooses the input values, data structure contents, and executes the program multiple times.

Figure 13 provides an overview of the Nero system. As in the BDP (Figure 7) and Konure (Figure 10) pipelines, the grey box with bolded dotted edges corresponds to the original application, while the light blue boxes with dashed edges correspond to the inference components.

Our technique starts with a seed program. Nero restricts the space of programs that it can infer through a domain-specific language (DSL). In this case, that DSL can be expressed as a subset of Python. Additionally, Nero requires a command specification, which outlines the commands we want to infer, and a data specification, that details the types for the global data structures used by the module and which our inferred model will replace with a database.
Randomized Input Solver

In the posting module, a (partial) command specification would correspond to

```json
[
  "post", "do_post", "int", "str", "str"],
]
```

This specification states that the command post is implemented by the do_post function in the seed program, and it takes three arguments: an integer and two strings.

A (partial) data specification would be

```json
{
  "users": [
    [
      "str", "str"],
    ["id", "first_name", "last_name"],
    ["Users"
  ]
}
```

This specification states that the users data structure contains entries corresponding to two strings (first_name and last_name). The corresponding database schema is a table named users, with three columns: a virtual primary key (id) and the first and last name columns (corresponding to the two strings in the original data structure’s entries).

**Execution:** Nero takes the seed program, along with the two specification files, and generates boilerplate code to produce a read-eval-print-loop (REPL) that can be used to interact with the application. This REPL is used to execute commands with new inputs and explore the application’s behavior. This has the benefit of reducing the code that the seed program developer has to write, as well as standardizing the interface that Nero uses for inference.

**Dynamic Analysis:** Nero relies on dynamic instrumentation to produce execution traces. These traces contain information on the data structures accessed and assigned to. To collect these traces Nero uses the REPL wrapper described previously, and records any accesses (and parameters used) to lists or dicts during execution (as well as certain key functions, like `print`) by using instrumented variants of lists/dicts.

For example, the expression `users[user_id_query]` in line 16 of Figure 12 would produce the following entry in the execution trace (assuming that `user_id_query` has a value of 2, the user’s data structure has 1 as an instrumentation identifier (UID), and the corresponding entry in the list contains `("john", "doe")`):

```python
list UID1 __getitem__ [2] ["john", "doe"]
```

**Inferring Program Constructs:** Nero collects the traces of data accesses and infers the program’s data and control flow. To do this Nero identifies the origin of different values observed in the trace, the presence of loops, and the constraints for any conditional loops. Nero repeatedly executes the program with new inputs to reduce ambiguity and explore new paths.

For example, Nero can infer data structure accesses, such as `users[user_id_query]`, directly from the trace. If Nero initializes data structures and chooses input values carefully, it will also be able to infer that the argument to this lookup (`user_id_query`) corresponds to an input parameter to the post command (implemented by `do_post`). However, the origin of values can be ambiguous if multiple possible locations could have produced that value. This highlights the importance of repeated execution and inference for Nero’s success.

While the example presented here includes user identifiers and post identifiers, Nero does not rely on knowing that these are unique identifiers (and map to a database primary key). The inference algorithm, however, does rely on accessing the input parameters to the application and successfully identifying that the input parameter is used as an index/key to access a list/dict. We believe the number of programs that take an input parameter value and then use it to access a program data structure is significant.

Loops over data structures in the program are identified through execution trace events creating iterators. Conditional loops, on the other hand, require that Nero infer a predicate that induces execution of the body. To do so, Nero uses a randomized approach to inferring data constraints to generate input/data structure values. The algorithm is inspired by simulated annealing [40]. It fine tunes the temperature parameter to balance the inherent tension in value generation: Nero needs to induce repeated values across structures to lead to the conditional loop’s execution, but inference also requires unique values to resolve ambiguity in the program’s logic.
Randomized Input Solver: While values are generated randomly by Nero, they are subject to constraints that will 1) avoid application crashes, 2) reduce inference ambiguity, and 3) explore program paths (by inducing conditional loop execution). These constraints are discovered progressively during inference and are used in a graph-based algorithm to populate possible values.

Nero’s randomized value generation demonstrates the flexibility of such an approach when the domain of values grows and the application complexity increases, compared to a deterministic, constraint-based solving approach.

Code Generation: Nero’s inferred model consists of database queries that can effectively capture the original program behavior. Nero can generate a skeleton program, consisting of database boilerplate such as schema definitions and database connection/cursor management, and integrate the queries into this skeleton to produce an executable program. The current implementation of Nero uses Python and its accompanying SQLite module [57].

Evaluation: We evaluated Nero by using it to infer four Python seed programs, including a task management program, a chat room program, and a blog program. These programs are adapted from database-backed applications [1–3], by rewriting a subset of their functionality in Python using standard in-memory data structures. For all these programs, Nero fully inferred the target functionality and produced regenerated programs that use a database.

Comparison to Object Relational Mapping: Object Relational Mapping (ORM) frameworks [4, 5] facilitate access to a database by providing objects which an imperative program can use to query/modify a database. While both ORMs and Nero share the goal of integrating database usage into applications, they take orthogonal approaches. Namely, an application that uses an ORM must already be implemented to use the corresponding API. In contrast, Nero’s core focus is the inference task: it infers the data usage information and its relational mapping directly from executions of the seed program, which can be written with standard data structures. Additionally, the model that Nero infers, while represented in terms of SQL queries, does not need to be limited to producing database executions, as highlighted in Section 7.3.

User Manual Input: To use Nero the user must provide a seed program, which has semantics corresponding to the desired database-backed application but is written in a subset of plain Python (i.e., the Nero’s DSL) using standard in-memory data structures. The user also provides a data and command specification. The user may also specify the desired names to use for the database tables and columns in the regenerated program. The command specification describes the commands of the application, specifically, the names of the entry functions in the seed program, as well as the datatypes for their arguments.

7 Active Learning for Software Engineering as a General Paradigm

We discussed three systems, BDP, Nero, and Konure [73], which employ active learning to infer a target program. These three systems leverage domain knowledge to enable the design of effective inference algorithms. This approach is consistent with the history of development trends in programming languages and software engineering in general—compiler optimizations, for example, were first designed for specific code patterns and later developed into sophisticated systems effective for a significantly wider range of programs [26, 39, 50].

We anticipate that future research in active learning of programs will exploit modularity to extend to expressive domains, where target applications consist of multiple (inference) modules. Potential domains include software for data retrieval, image processing, embedded systems, distributed systems, network protocols, user interfaces, and certain numerical computations. Potential use cases include software migration, software understanding, security analysis, robust code generation, etc. There is a large design space for this line of research, as demonstrated by the different systems discussed.

Broadly speaking, we describe active learning for software engineering in terms of three components: an observation model, an inference model, and a generation model. We next place the three systems in the context of each of these paradigm components, point towards different research directions, and highlight open questions.

7.1 Observational Model

The observation model roughly outlines what aspects of the program’s behavior the technique observes and how it does so. All three systems described use a greybox technique. BDP and Nero use program instrumentation to obtain execution traces. While the former is focused on tracking concrete executions and building byte-level expressions over the inputs, the latter only records instrumented method calls along with information on the concrete input and return values. Konure, yet another greybox approach, intercepts SQL queries as well as the observed input and outputs.

Other models of observation present different trade-offs. For example, blackbox interaction may limit the complexity of the target application that can be inferred, or it may lead to uncertain inference, but it also removes assumptions such as implementation language or direct access to the application’s execution. For example, Hecate [71] uses a blackbox approach to interact and infer programs that store and retrieve data, as long as the applications conform to a model for programs using a relational database. Compared to this blackbox approach, the greybox approach where SQL queries are visible (Konure) infers more sophisticated database programs because of the richer information available.
A whitebox observation model is yet another alternative, which would allow incorporating additional program analysis, or might facilitate rewrites of the program that reduce (or remove) inference ambiguity (e.g., using SSA).

Exciting research directions that target the observational model include identifying domains and applications that can be successfully inferred with limited interaction (e.g., not just black box, but also limited numbers of queries). Similarly, we have assumed that the application’s behavior is observed truthfully and there is no adversarial interaction. However, one could imagine scenarios where we are trying to infer an application that is actively adding noise to the outputs or non-deterministically picking execution paths.

**Producing New Inputs:** A key advantage of active learning is that the technique can use new inputs to improve inference. How these inputs are produced may vary depending on the technique.

For example, Nero produces values using a constrained random value generator. BDP, in contrast, relies on having access to a large number of example inputs and filtering down to those that can not be parsed with the currently hypothesized program. Konure uses an SMT solver, along with constraints implied by program traces, to produce new values.

Depending on the application, generating new inputs can pose challenges. For example, a highly structured custom binary input would require a sophisticated generation process, in contrast with a common format such as bitmap images. Integrating input generation techniques developed in areas such as fuzzing into an active approach could broaden the range of applications that can be inferred.

### 7.2 Inference Model

The inference model roughly aims to answer the question: “How do we represent the observations made in a way that we can identify and generalize common application patterns?” A key component to successfully identify these shared patterns is the use of an intermediate representation amenable to inference. A good representation should capture the behavior of interest and constrain the space of possible hypotheses derivable from an observation.

We have found carefully designed DSLs to be a particularly effective intermediate representation. For example, BDP uses a simple language that describes data structures as sequences of bytes (or words), allowing it to identify repetitions and generalizations. Nero constrains the seed program to a DSL so that it can identify key control flow. Similarly, Konure’s DSL represents hypothesized (partially inferred) programs and guides inference through hierarchical expansion of non-terminals (i.e., unexplored subprograms).

DSLs have already been found useful in constraining the search space in other automated programming areas [7]. They are modular by definition [51], and thus are well matched to the modular approach we believe is critical to scale active learning-based program inference to large applications. The three systems presented all incorporate knowledge about the target domain to make inference more efficient; incorporating this information through a DSL is simple—For example, extend the grammar with a corresponding production or modify the semantics of an operator. Finally, a DSL facilitates the use of existing analysis techniques (such as structural induction proofs) to reason about the correctness of the inference, as done in Konure [73].

Different inference models may reason differently about the observed executions and their generalization. For example, BDP partitions the space of inputs (based on executions observed and their input files) and iteratively generalizes a DDP based on each partition’s members, which when combined with flag expressions, capture the application behavior. Konure, in contrast, maintains a program representation guaranteed to cover all observed executions, and modifies that hypothesis based on satisfiability criteria for newly observed traces.

Different approaches to inference may also yield varied properties. For example, Konure’s inference algorithm guarantees termination and that the paths in the application are fully explored if no non-terminals remain in the hypothesized program.

Identifying appropriate inference procedures for different intermediate representations, domains, and applications, is an exciting direction for future research. Other parts of the paradigm also play a role: for example, can inference be made more efficient under a whitebox observational model? Can we formulate a generalized inference procedure based on an input DSL and application? Or is efficient inference restricted to custom procedures for each group of applications we want to infer? Can we develop statistical techniques that can reason effectively for programs that violate many standard statistical assumptions (as might be the case for a stateful, long-running program)?

### 7.3 Generation Model

There are many possible uses for the inferred model of an application’s behavior. There is no reason, a priori, that these models need to be used for code generation. We term this portion of the paradigm the *generation model*, as we can generate different outputs based on the representation we have learned. For example, an inferred model could be used to generate documentation for a program (or be used as documentation itself). How best to integrate these inferred models, beyond code generation, into existing developer workflows remains an open question.

Assuming the goal is to generate code, a question about the possible impact on software maintenance arises. We believe that code generated from an inferred model has the potential to be simpler (for example, see the code generation discussion in Section 4), as restricted by the chosen generation
model. Program inference, in this regard, holds particular promise in rejuvenating legacy code. We also believe inferred models can help identify application functionality that can be modularized, producing self-contained and reusable components for software development. Jointly, these properties would contribute to reduce the burden of software maintenance through simpler, modular, and extensible (generated) programs.

8 Challenges and Threats to Validity

Successful inference must avoid inferring incorrect properties. A carefully designed representation, such as a DSL, can be used to invalidate particularly undesirable outcomes. For example, both Nero and Konure have DSLs that only allow bounded loops. Furthermore, the representation, when co-designed with the inference algorithm, can lead to desirable inference guarantees. Konure, for example, guarantees that its inference terminates and when it does so no unexplored paths remain.

We anticipate that the main difficulty in deploying our active learning-based approach to program inference will be in appropriately matching the target applications and the restrictions imposed by our techniques. In particular, developers using these systems in practice will need to understand the limitations of each technique. As such, clear documentation of these inference systems is a key design goal.

Active learning requires a supply of new inputs. Depending on the application, generating these examples may be a challenging problem in its own right (e.g., highly structured inputs). However, we have already successfully inferred programs in two domains (data retrieval/storage and binary parsing) and believe that many productive applications can be handled with relatively simple inputs. Techniques from other research areas, such as fuzzing, can also be integrated to help.

To infer large-scale applications, we propose to exploit the modularity of modern software to enable effective (and scalable) inference. This direction leverages the trend of increasingly popular third-party library usage [6, 16, 52, 79, 81, 89]. Applications with less modular designs will likely present a cost-of-inference challenge and the need for refining the observation and inference models used. For example, BDP would incur a larger overhead when inferring a monolithic application, where parser data structures might be populated throughout the application (requiring broader instrumentation). However, existing work on program partitioning [21, 22, 56, 77] could be adapted to induce some degree of modularity and reduce the cost of inference. Other practices, such as the use of global variables, may also influence the inference strategy. For example, an existing prototype [65] successfully developed an inference strategy that handled reads and writes of a global data structure. The key to this success is identifying the effective boundary of the target behavior we are interested in inferring, interposing the appropriate observation model, and tailoring the inference model as needed.

9 Related Work

Program Synthesis: Program synthesis is an active area of research [7, 28, 36, 38, 66], particularly programming-by-example [64, 75, 85], where the user provides input-output examples as a form of specification. These techniques often define a DSL that captures the space of programs. Synthesizing a correct program (out of potentially many) then becomes a search over the DSL for a program that is consistent with the examples provided.

Oracle-guided synthesis [38, 43, 76] extends the search to include an oracle, often an SMT solver, which can validate proposed programs and provides counter-examples when the proposed program is incorrect.

The systems discussed here perform inference by (partially) observing the behavior of a full application. Rather than rely on a fixed corpus of input-output examples, the application’s execution is used to define the target specification. Incorporating active learning allows these systems to successfully and efficiently expand the input-output examples automatically as needed.

Reverse Engineering and Transformations: Reverse engineering has been successfully used to understand, maintain and rewrite otherwise unfamiliar applications (e.g., legacy code) [20, 49, 78]. However, many reverse engineering tools require significant human effort and expertise [44], and assume the tool has access to extensive static information, such as source code or the compiled binary, or dynamic information, such as whole-program instruction-level instrumentation [19, 27, 66]. Work automating the reverse engineering process [23, 48, 55] has focused on narrow inference tasks like communication protocols rather than whole programs.

Similarly, source code transformations, by definition, rely on significant access to the original application, which for example rules out transformation of remote applications. Recent work [41] has explored enabling source code transformations across multiple languages, but the scope of these transformations remains restricted.

Our vision for active learning for software engineering addresses the hurdles highlighted. For example, all three systems presented make limited use of human intervention and automate a significant portion of the inference process. The observational models presented enable inference on applications where the source code may not be available. Finally, representing the inferred model in a DSL allows us to employ transformations that can be re-used across possible target languages, as the program translation step is handled by an independent generation model.
Language Inference: Inference of regular and context-free languages has been a well-studied problem [32, 58, 87]. These techniques have been applied to, for example, the grammar for a program’s inputs [13], rules of a board game [42], the partitioning of objects in an image [30], among others. The examples we presented here highlight the potential for inferring full program functionality.

Model-based Software Engineering: The use of models in software engineering has enabled diverse applications such as program verification [15], testing [18], generation [47], and reuse [35]. The user may be tasked with providing a model specification written in a higher-level language, or a simplified model may be extracted automatically (if the target software already exists for analysis or execution).

The models inferred by BDP, Nero and Konure reflect full functionality of particular application commands, rather than a simplified model (except where a pruned model is intentionally desired). All three systems use a DSL to constrain the search space for their inference procedures, but do not expose these directly to the user.

Active Learning: Active learning has been extensively explored in the machine learning community [10, 82, 86] and successfully applied to a variety of tasks, including complex image classification [88], data cleaning and labeling [33], symbolic systems [9], among others.

Previous work in the software engineering community has also explored applications of active learning. For example, active learning has been used to: derive program assertions from test suite executions [59], infer likely points-to specifications using black-box observation [14], improve program behavior classification [17], and synthesize Datalog programs [74].

The three systems we discussed here use active learning for the broader task of inferring full functionality of key commands in applications.

10 Conclusion
Pervasive modularity in modern software has created an opportunity for deploying active learning at scale, enabling a state-of-the-art approach to security, code generation, adaptation, documentation, and more. We walked through how three existing inference techniques (along with prototypes) infer key, and varied, modules in a representative social networking application. We outlined the general paradigm, contextualized the systems discussed, and identified open questions and challenges.

Acknowledgments
We thank the anonymous reviewers for their helpful comments and suggestions. This research was funded in part by National Science Foundation grant CNS-1513687, and DARPA grant FA8650-15-C-7564.

References


[-]