Active learning for inference and regeneration of computer programs that store and retrieve data

Active Learning for Inference and Regeneration of Computer Programs That Store and Retrieve Data

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

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

Varun Mangalick
MIT EECS & CSAIL
USA
vrm@mit.edu

Abstract

As modern computation platforms become increasingly complex, their programming interfaces are increasingly difficult to use. This complexity is especially inappropriate given the relatively simple core functionality that many of the computations implement. We present a new approach for obtaining software that executes on modern computing platforms with complex programming interfaces. Our approach starts with a simple seed program, written in the language of the developer's choice, that implements the desired core functionality. It then systematically generates inputs and observes the resulting outputs to learn the core functionality. It finally automatically regenerates new code that implements the learned core functionality on the target computing platform. This regenerated code contains boilerplate code for the complex programming interfaces that the target computing platform presents. By providing a productive new mechanism for capturing and encapsulating knowledge about how to use modern complex interfaces, this new approach promises to greatly reduce the developer effort required to obtain secure, robust software that executes on modern computing platforms.

CCS Concepts • **Software and its engineering** → *Source code generation*; *Software development techniques*; *Software reverse engineering*;

Keywords Active Learning, Program Inference

ACM Reference Format:
Martin C. Rinard, Jiasi Shen, and Varun Mangalick. 2018. Active Learning for Inference and Regeneration of Computer Programs That Store and Retrieve Data. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18), November 7–8, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3276954.3276959

1 Introduction

Within the last decade, undergraduate computer science enrollments, both within and outside the major, have dramatically increased [Zweben and Bizot 2016]. As a result, undergraduates are now acquiring basic programming skills as a normal part of their college education. Indeed, the ability to write (relatively simple) programs, like the ability to read, write, and perform basic mathematical reasoning, is now increasingly seen as part of the personal portfolio of a literate person in our culture [Lohr 2017; Smith 2016].

At the same time, the systems on which even simple production software must execute are becoming increasingly complex. Some decades ago most software executed on a single machine, with the programming environment providing a few simple abstractions (such as file system interfaces) for accessing the devices attached to that machine. Most software today, in contrast, is expected to execute in complex, networked, distributed computing platforms. A common scenario, for example, is for a program to compute over data stored across many machines in a cloud computing environment to generate results that are then distributed via the Internet for graphical presentation on remote devices.

Modern software environments rely heavily on software packages that help developers deal with the resulting complexity. Examples include application server frameworks such as JBoss and IBM WebSphere, key/value storage systems such as Redis, NoSQL databases such as HBase, distributed memory caching systems such as memcached, and cluster computing frameworks such as Spark and MapReduce. While the implementations of such systems encapsulate the otherwise potentially overwhelming complexity of coordinating the actions of the components of large distributed computing systems, the programming interfaces they provide are far from easy to use (as can be seen in the large volume of questions posted to web sites such as Stack Overflow [StackOverflow 2018]). Indeed, developers that work with such systems spend much of their time constructing appropriate search terms to find previously developed code that they can copy and adapt for their needs. The complexity of these programming interfaces can be seen as especially inappropriate

Onward! '18, November 7–8, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6031-9/18/11…$15.00
https://doi.org/10.1145/3276954.3276959

given the relatively simple core functionality that many of the computations implement. At a conceptual level, such computations often simply store and retrieve data or perform simple computations on the stored data. The complexity comes not from the core functionality that the computation implements, but from the computing platform on which the computation executes.

We propose a new approach for developing software for such computing platforms. Instead of coding to complex programming interfaces that existing software packages export, developers implement the core functionality in a *seed program* using the programming language of their choice. The seed program may use only the simplest standard programming interfaces, such as standard text input and output interfaces.

Our system first interacts with the seed program to learn its core functionality. The system then regenerates (a potentially augmented version of) the computation that uses sophisticated software packages to implement the learned core functionality on the new, potentially much more complex computing platform. In effect, the knowledge and expertise required to use the relevant software packages are all encapsulated in the regenerator. This approach can be particularly productive in a world in which basic programming skills are widely available, but the specialized knowledge and expertise required to productively use specialized software packages is scarcer. This is the case for the world we are now entering as a society: such specialized knowledge of specialized software packages is constantly changing, available to fewer people, and more difficult to use for everyone regardless of their skill level. Our approach is founded on several principles:

- **Programs as Specifications:** We propose to use programs as (partial, noisy) specifications of the desired core functionality. In comparison with using standard specification languages based on formal logic, programming is a relatively widely available skill within our society (and a skill that promises to become more available over time). In comparison with natural language specifications, programs provide precision and the ability to explore and learn the specification by observing the program as it produces outputs in response to targeted synthesized inputs.
- **Program Inference via Active Learning:** Starting with a *seed program* that (mostly) implements the desired core functionality, the system uses active learning to reverse engineer the seed program. The result is an inferred representation of the core functionality that the seed program implements.

  In this paper we present a black box inference algorithm that interacts with the seed program by generating inputs and observing the resulting outputs. An advantage of using a black box approach is that the seed program can use any language or implementation methodology. For example, the black box approach readily works with obfuscated seed programs. Gray box and white box approaches can also be appropriate — they can obtain certain kinds of information more quickly by observing aspects of the implementation, but may require more involved mechanisms that (dynamically or statically) instrument, monitor, and/or analyze aspects of the program and/or its execution [Mendis et al. 2015; Shen and Rinard 2017, 2018a; Wu 2018].
- **Noisy, Partial Programs:** Developing a fully correct program that handles every corner case correctly is known to be much more difficult than developing a program that implements most of the desired functionality correctly. The inference algorithms are therefore designed to work with such mostly correct programs, in some cases by working with inputs that are unlikely to trigger rare corner cases, in others by identifying and discarding undesirable behavior (noise) from the seed program that should not be part of the specification. This approach can reduce developer effort by enabling the developer (of the seed program) to focus on the desired core functionality while omitting error checking and corner case code (potentially with such code inserted automatically during regeneration).
- **Focused Domains:** To make the inference tractable, each inference algorithm focuses on a specific domain. Ideally, the domain can be *finitely testable*, i.e., each computation in the domain can be uniquely identified from within all of the computations in the domain by a finite set of inputs. In this paper we capture the domain with a collection of data structures. Another promising approach is to capture the domain via a domain-specific language [Shen and Rinard 2017, 2018a; Wu 2018].
- **Augmented Regeneration:** Working with the learned representation of the core computation, regenerate the program for the target context. This regeneration may involve simply producing a computation that uses the facilities that the target context provides. In most cases, however, we expect that a productive regeneration will augment the core computation with additional capabilities. These may include additional error or security vulnerability checks, data consistency or cleaning checks, robustness and recovery code, or the generation of a graphical user interface for the program.
- **Reinterpretation:** Many modern programming languages support a simple and basic model of computation (sequential execution, file input and output, standard data structures, a single address space) that usually enables straightforward implementation of the desired core functionality. In many cases, however, the goal is to implement this core functionality in a more complex environment — to operate on distributed data, to work with data stored in a relational database or key/value store, to access specialized computing devices, to execute time-consuming computations in

parallel, to package the core functionality into an appealing graphical user interface potentially accessed via the Internet, or to access values available via remote sensors. To support such implementations, the regeneration algorithm reinterprets standard constructs to translate them into implementations that operate successfully in the new, more complex target context.

1.1 Encapsulated Knowledge

The regenerator encapsulates the knowledge of how to use the complex software components that the regenerated code uses. Over the last several decades, the field has explored a variety of approaches for capturing and communicating this kind of knowledge. Examples include user manuals, textbooks, example programs, and, more recently, web sites such as Stack Overflow [StackOverflow 2018]. All of these mechanisms require the developer to examine the provided code and modify it to adapt it for their purpose in their system. Regeneration enables the developer to immediately obtain working code that implements the desired core functionality without the need to examine and/or modify the code (although the developer may very well do so if he or she desires). In this sense the regenerator can provide a more robust encapsulation of the (in some cases quite involved) knowledge required to productively use the powerful but complex target components.

1.2 Current Scope

We initially focus on programs that store, retrieve, and/or delete data because this class of programs combines 1) widespread applicability with 2) simple enough semantics to support relatively straightforward inference and regeneration algorithms. We anticipate several use cases:

- **New Software:** In this use case, the seed implementation is developed from scratch, for example by implementing a simple text-based interface in a widely-taught language such as Python. These use cases typically involve substantial reinterpretation and augmentation to re-implement the functionality on more complex production computing environments and/or to provide the system with an enhanced user interface.
- **Software Archeology:** In this use case, the developer starts with a legacy system that implements the desired functionality. Here one goal is to start with a system that runs in an obsolete or otherwise undesirable computing context to obtain a regenerated version that can operate successfully in a more modern context. Another goal is to start with a system that may have defects or security vulnerabilities to generate a program without defects or vulnerabilities (by, for example, systematically generating appropriate checks and code). Yet another goal is to improve performance by replacing an inefficient implementation with a more efficient regenerated implementation. Even another goal is to replace a program that has been maintained so intensively that its lifecycle is over and it is no longer feasible to continue to maintain it [Belady and Lehman 1976].
- **Targeted Functionality Extraction:** In this use case, the seed program implements a range of functionality, only part of which comprises the desired core functionality. The developer provides a limited interface specification that targets only the desired core functionality, with the program inference system oblivious to the remaining undesired functionality.

1.3 Future Scope

In the longer term, we envision several directions for growing the scope of this idea (some of these directions have already been explored in concurrent research [Shen and Rinard 2017, 2018a; Wu 2018]).

- **Recursive Component Decomposition:** Many large applications may have sufficiently complex behavior to make direct inference infeasible. But this behavior may arise via the interaction of multiple inferrable components. A strategy that recursively subdivides the application to identify, learn, and regenerate some or all of these components, then composes the regenerated components to obtain a final regenerated application, can enable the application of program inference and regeneration to such applications. One particularly appealing aspect of this approach is that it would enable the appropriately targeted deployment of different learning algorithms to learn different components that implement different computational patterns.
  One approach would leave the components in place, observing not just inputs and outputs, but all traffic on component interfaces. Concurrent research has used this approach to learn and regenerate database-backed web applications [Shen and Rinard 2018a] or to replace Python data structures with a backend database [Wu 2018].
  Another approach would extract and learn each component in isolation [Amidon et al. 2015], enabling the direct application of active learning to each component (instead of working with the interactions that the application generates between components running in place as part of the full application).
- **Merging Applications:** Learning functionality from multiple applications would enable the regeneration of applications that combine all of the functionality into a single unified regenerated application. This could be particularly useful, for example, for augmenting an existing application with desirable features implemented in other applications. Here inference and regeneration would be an alternative to direct inter-application code transfer [Sidiroglou-Douskos et al. 2017, 2015].

- **Partial Inference Algorithms:** Our current approach uses active learning to uniquely identify the inferred program within the target computational domain. We anticipate the development of inference algorithms that aspire only to partially identify the program within this domain. The motivation might be to improve the performance of the inference algorithm or to work with a more general or complex class of computations. In such scenarios the inference algorithm may only aspire to infer the program with a high probability or a concise representation (in comparison with the other programs that are consistent with the observed inputs and outputs). Here the goal would be to generate inputs and observed outputs that distance the inferred program from other candidate programs that are also consistent with the inputs and observed outputs. The inferred partial models may help debugging or error discovering [Aarts et al. 2013; De Ruiter and Poll 2015].
- **New Domains:** In this research we focus on programs that store and retrieve data. We anticipate generalizing the approach to new domains, for example programs that read and write data in more sophisticated ways and programs that perform numerically intensive computations. Applications that access databases comprise a particularly compelling target, with encouraging initial results that highlight the potential of this domain [Shen and Rinard 2017, 2018a; Wu 2018].
- **Enhanced Regeneration and Reinterpretation:** We also expect the program regeneration and reinterpretation algorithms to grow in sophistication to deliver regenerated applications that operate successfully in more complex or challenging environments, with the value of the regeneration growing with the complexity of the environments. We expect environments characterized by distribution, errors, stringent security or privacy needs, or scale to be particularly compelling regeneration targets.

1.4 User Inputs Versus Active Learning

Instead of using active learning, an alternative approach is to use dynamic monitoring to obtain inputs for interacting with the seed program. We implemented a system that observes the inputs, outputs, and database traffic from a running system in normal use and then synthesizes a model of the application from this information [Shen and Rinard 2018b]. Preliminary results indicate that the monitored user inputs may be insufficient for exploring the functionality in rare usage scenarios, especially for more sophisticated applications. When the observations are insufficient, the system may synthesize incorrect programs. These results highlight the utility of active learning in inferring accurate seed program semantics in this context.

1.5 Implications for the Field

This approach promises to substantially reduce the time and effort required to obtain programs that work with complex programming interfaces on modern complex hardware platforms. By automating the generation of error, privacy, and security checking code, it promises to improve program robustness and reliability.

But despite these instrumental advantages, perhaps the most important implication is the transformative effect this approach can have on the activities and daily lives of people who work in the field. Modern software development is increasingly becoming an activity in which developers spend their time searching for arcane code sequences that (for often poorly understood reasons) happen to deliver something close to the desired functionality. Automating the generation of these code sequences can free developers to focus on the core technical challenges of automating central activities that human society relies on to function smoothly. The profession will (once again) provide the world's best platform for creative people to spend their days in inspiring technical work.

1.6 Paper Structure

The remainder of the paper is structured as follows. We present an example of a course registration system in Section 2. The seed program is written with a text interface in Python, with the registration information stored in native Python data structures. We present a regeneration for an HTTP-based web server that uses Python, Flask, and SQL. In Section 3 we present the program inference algorithms. We survey related work in Section 4 and conclude in Section 5.

2 Example

We next present an example that highlights how our proposed approach works for a simple student course registration system. We implemented a prototype that successfully infers store, retrieve, and delete operations and their mappings.

2.1 Command Interface for Seed Program

Our approach starts with a command interface that the user provides for the registration system. The command interface is in the form of operations, or parameterized commands, that the registration system should implement:

- **enroll (name, id, year):** Enroll a student to take classes. Store the student's name, id, and graduation year.
- **id (name) → id:** Retrieve the id of the student with the given name.
- **name (id) → name:** Retrieve the name of the identified student given the student's id.
- **add (id, class):** Add a class to the list of classes for which the student is registered. The student is identified by the student's id.

- **drop (id, class):** Drop the specified class from the student’s registration list. The student is identified by the id.
- **classes (id) → class:** Retrieve the class or classes for which the identified student is currently registered.

Here each operation has a command (**enroll**, **id**, **name**, **add**, **drop**, or **classes**), parameters (**name**, **id**, **year**, or **class**), and outputs (name, id, or list(class)). The parameter names all identify disjoint sets of (abstract) objects so that, for example, the **name** parameter of the **enroll** operation is drawn from the same set of objects as the **name** parameter of the **id** operation, while the **class** parameter (from the **add** and **drop** operations) is drawn from a different disjoint set of objects.

2.2 Seed Program

The developer next implements a seed program, written in Python, that implements the core functionality. Python is a widely taught language that many developers find easy to use. The program implements a simple text-based interface that is designed to accept and process one command per line. While a text-based interface may not be as easy to use as a more involved graphical or browser interface, it is much easier to implement and supports text-based program inference systems.

Figure 1 presents the seed program in our example. The program maintains four data structures: `classes`, which maps each student id to the list of classes for which the student is registered, `ids`, which maps student names to corresponding student ids, `names`, which maps student ids to corresponding names, and `years`, which maps student ids to corresponding years of graduation. The main loop reads and implements a command for each line of input. The seed program is quite simple — there is little input validation (for example, the program will accept any string as a student name, student id, class id, or year), little error checking (for example, the code that implements the **drop** and **classes** operations does not check if the `classes` map has an entry for the provided `id`, which leaves the program vulnerable to KeyErrors), no corner case checks, and no attempt to provide useful messages to the user of the program. The text-based interface is straightforward to implement using basic Python input and string handling constructs. There is no need for the developer to learn and use more complex Python packages for building HTTP servers, graphical user interfaces, or interacting with database storage systems.

2.3 Program Inference

The program inference algorithm exploits the availability of the seed program to learn the functionality. Unlike most machine learning and program synthesis approaches, which are limited to working with a provided set of input/output pairs, the program inference algorithm can purposefully select the inputs it provides to the seed program to target and resolve ambiguities. We next outline how our prototype program inference algorithm exploits this ability (as well as the structure present in the provided interface to the seed program) to learn the core functionality. The inference algorithm is designed to work with programs that have the following sets of properties:

```
import sys

classes = {} # id -> [
     classes]
ids = {} # name -> id
names = {} # id -> name
years = {} # id ->
     graduation year

while (True):
  line=sys.stdin.readline()
  if not line: break;
  list = str.split(line)
  cmd = list[0]

  if cmd == "enroll":
    name, id, year = list
        [1:4]
    names[id] = name
    ids[name] = id
    years[id] = year

  elif cmd == "id":
    name = list[1]
    print ids[name]

  elif cmd == "name":
    id = list[1]
    print names[id]

  elif cmd == "add":
    id, num = list[1:3]
    if not id in classes:
      classes[id] = []
    if not (num in classes[id
        ]):
      classes[id].append(num)

  elif cmd == "drop":
    id, num = list[1:3]
    classes[id].remove(num)

  elif cmd == "classes":
    id = list[1]
    print classes[id]
```

Figure 1. Python seed program for a class registration system

- **Key/Value Maps:** The program works with a fixed set of maps. Each map contains relations that map a key to a value or to a list of values. Note that the program is not required to implement the maps using any particular data structure or mechanism — because the program inference algorithms for this example only generate inputs and observe the resulting outputs, they are oblivious to the particular map implementation technique.
- **Store, Retrieve, and Remove Operations:** The program implements three kinds of operations, specifically *store* operations, which store one or more relations between the parameters of the operation into one or more of the maps, *retrieve* operations, which use the parameter as a key to retrieve and return a value (or list of values) from one of the maps, and *remove* operations, which remove one or more relations from the maps.
- **Initial Empty Maps:** When the program runs, it starts with empty maps.

The provided interface distinguishes the retrieve operations, which return values, from the store/remove operations, which return nothing.

2.4 Store/Retrieve Pair Inference

The program inference algorithm first repeatedly executes selected operations with selected parameters starting from empty maps to discover *store/retrieve pairs* — paired operations in which the first operation stores a relation into a map and the second operation retrieves and returns the corresponding value (or list of values) stored by the first operation (Section 3.1). Each store/retrieve pair is *mediated* by a key/value pair chosen from the parameters of the store operation — the first parameter of this pair is the key of the stored relation; the second parameter is the corresponding value.

The program inference algorithm repeatedly starts with an empty map, executes an **enroll** operation with a unique **name** and **id**, then executes **classes**, **name**, and **id** operations to determine if one of these operations returns one of the enroll parameters. If so, the inference algorithm has discovered a store/retrieve pair backed by a map. In the example the inference algorithm discovers that **enroll/name** is a store/retrieve pair mediated by the **id/name** parameters of the **enroll** operation, **enroll/id** is a store/retrieve pair mediated by the **name/id** parameters, and **add/classes** is a store/retrieve pair mediated by the **id/class** parameters of the **add** method.

2.5 Keep, Replace, or List Inference

After the pairs of store/retrieve operations as well as their keys are determined, the inference algorithm next uses this information to determine which store operations accumulate the stored values in lists, which overwrite the old mapping with the new mapping, and which leave the old mapping in place and discard the new mapping (Section 3.2). The algorithm executes two store operations that insert different values into the same map under the same key, then executes the corresponding retrieve operation to determine if the retrieve returns both values, the first inserted value only, or the second inserted value only. In our example the inference algorithm determines that the **enroll** operation overwrites the old mappings and the **add** operation accumulates the stored values in lists.

2.6 Store/Delete Pair Inference

The next step is to use the store/retrieve pairs to identify *store/delete pairs* in which the first operation stores a relation and the second operation removes the relation (Section 3.4). The inference algorithm executes the store operation, then a candidate delete operation, then the corresponding retrieve operation. If the retrieve operation does not return the stored value, then the inference algorithm has discovered a store/delete pair. In our example the inference algorithm determines that the **add** and **drop** operations comprise a store/delete pair.

2.7 Map Inference

Finally, the inference algorithm uses the store/retrieve pairs to determine which operations work with the same map and which work with different maps (Section 3.5). The basic idea is to execute the seed program twice, once with one store operation and once with another store operation. Both operations insert the same relation, but potentially into different maps. Both executions next execute the same retrieve operation (the paired retrieve operation from the first store operation). If both executions return the same value, the inference algorithm concludes that they both accessed the same map (working under the assumption that the retrieve operation always accesses the same map). The code generation algorithm uses the resulting inferred equivalence classes of operations to determine how many maps to generate and which maps each operation accesses.

2.8 Tolerating Seed Program Errors

In any of these steps, the inference algorithm tolerates noise in the seed program when it crashes due to unchecked errors. For example, a Python retrieve operation may crash from a KeyError when attempting to look up a key that does not exist in a map (the **classes**, **name**, and **id** operations). In this situation, the inference algorithm decides that the retrieved value is **Nil**. Also, a candidate delete operation may crash from a KeyError when attempting to look up a key that does not exist in a map or, when the key does exist, may crash from a ValueError when attempting to remove a value that does not exist in a list (the **drop** operation). In these situations, the inference algorithm decides that the candidate delete operation does not correspond to the store operation and immediately goes to the next iteration of the closest enclosing loop. This design makes our algorithm robust against certain kinds of noise in the seed program while still learning the core functionality.

2.9 A Web Server with HTTP Interface

Here we present a regenerated program in Python that uses the Flask [Flask 2018] web framework (we also have regenerated versions in C using Redis, with both text and web interfaces [Rinard and Shen 2017]). Flask is a lightweight, flexible Python web framework for web application backends or servers. It is widely used for implementing industry micro-services. The regenerated program utilizes the SQLite3 package for non-volatile data storage. SQLite3 creates a local file which stores all the information associated with given tables and provides SQL access to read or modify the data in the file.

Regenerated based on the specified command interface (Section 2.1), the web server contains routes. Each route allows the web server to handle an HTTP request that implements a command in the seed program. For each command, the inputs and outputs use the JSON data format. Figure 2

```
import sqlite3
from flask import *
app = Flask(__name__)
DATABASE = 'students.db'
def get_db():
  db = getattr(g, '_database', None)
  if db is None:
    db = g._database = sqlite3.connect(DATABASE)
  db.row_factory = sqlite3.Row
  return db
def query_db(query, args=()):
  cur = get_db().execute(query, args)
  rv = cur.fetchall()
  cur.close()
  return rv
def write_db(query, args=()):
  db = get_db()
  cur = db.execute(query, args)
  db.commit()
  cur.close()
def init_db():
  with app.app_context():
    db = get_db()
    with app.open_resource('schema.sql', mode='r') as f:
      db.cursor().executescript(f.read())
    db.commit()
@app.teardown_appcontext
def close_connection(exception):
  db = getattr(g, '_database', None)
  if db is not None:
    db.close()
```

Figure 2. Boilerplate code in regenerated Python/Flask/SQL program

```
DROP TABLE IF EXISTS students;
CREATE TABLE IF NOT EXISTS students (
  id INTEGER NOT NULL,
  name STRING NOT NULL,
  year INTEGER NOT NULL,
  classes STRING NOT NULL
);
```

Figure 3. Regenerated SQL script to create database tables

contains the boilerplate code for building a Flask web server using an SQLite3 database. This code needs to be implemented once, then it is systematically regenerated for each seed program. Figure 3 contains the regenerated SQL script for creating tables in the database. The information about tables and columns are converted from the inferred mappings. This script is executed when a user initializes the database for the web server. Figures 4 and 5 present the regenerated HTTP request handlers. Each request handler corresponds to a command in the seed program. They contain the boilerplate code to convert inputs from JSON and convert outputs to JSON. They perform the inferred operations on the database using SQL queries.

In our approach, the knowledge of how to successfully handle URL requests in a web server and of how to successfully use SQLite, including the specialized knowledge of boilerplate code sequences, is all encapsulated inside the regenerator for automated oblivious reuse by system users.

```
@app.route('/enroll', methods = ['POST'])
def enroll_mapping():
  name = request.json['name']
  id = request.json['id']
  year = request.json['year']
  if len(query_db('SELECT * FROM students WHERE id = ? OR
       name = ?', (id, name))) == 0:
    write_db('INSERT INTO students (id, name, year,
        classes) VALUES (?, ?, ?, ?)', (id, name, year,
        ''))
  return ('', 200)

@app.route('/id', methods = ['POST'])
def id_mapping():
  name = request.json['name']
  data = {}
  for row in query_db('SELECT * FROM students WHERE name
      = ?', (name)):
    data['id'] = row['id']
  resp = Response(json.dumps(data), status=200, mimetype
      ='application/json')
  return resp

@app.route('/name', methods = ['POST'])
def name_mapping():
  id = request.json['id']
  data = {}
  for row in query_db('SELECT * FROM students WHERE id =
      ?', (id)):
    data['name'] = row['name']
  resp = Response(json.dumps(data), status=200, mimetype
      ='application/json')
  return resp
```

Figure 4. HTTP request handlers for commands enroll, id, and name

2.10 Installation, Configuration, and Documentation

In this section we have focused on the source code that the regenerator produces. But in modern computer systems the source code is only part of the solution. In addition to encapsulating the knowledge of how to develop code, the regenerator can also encapsulate knowledge of how to install and configure the relevant subsystems, in the form of natural language narratives or scripts that implement installation and configuratoin.

Our current regenerator implementation produces code without comments or documentation. While many developers may very well use the regenerated code directly without examining it, others may wish to examine, understand, or even modify the code. For these uses, the regenerator can also produce helpful documentation or comments.

3 Program Inference Algorithms

We next present program inference algorithms for programs whose operations store, remove, and delete relations from maps. The algorithms take as inputs the seed program *SP*, a set of potential store/remove operations *S* (each of which may store or remove relations), with each operation of the form $\mathsf{sop}\ \mathsf{p}_1 \ldots \mathsf{p}_k$ (here sop is the name of the operation

```python
@app.route('/add', methods = ['POST'])
def add_mapping():
  id = request.json['id']
  num = request.json['num']
  for row in query_db('SELECT * FROM students WHERE id =
      ?', (id)):
    classes = row['classes'] + str(num) + ';'
    write_db('UPDATE students SET classes = ? WHERE id =
         ?', (classes, id))
  return ('', 200)

@app.route('/drop', methods = ['POST'])
def drop_mapping():
  id = request.json['id']
  num = request.json['num']
  for row in query_db('SELECT * FROM students WHERE id =
      ?', (id)):
    classes = row['classes'].replace(str(num) + ';', '')
    write_db('UPDATE students SET classes = ? WHERE id =
         ?', (classes, id))
  return ('', 200)

@app.route('/classes', methods = ['POST'])
def classes_mapping():
  id = request.json['id']
  data = {}
  data['classes'] = []
  for row in query_db('SELECT * FROM students WHERE id =
      ?', (id)):
    data['classes'] = row['classes']
  resp = Response(json.dumps(data), status=200, mimetype
      ='application/json')
  return resp
```

Figure 5. HTTP request handlers for commands add, drop, and classes

and $\mathsf{p}_1 \ldots \mathsf{p}_k$ are the names of the k parameters), and a set of potential retrieve operations R, with each operation of the form $\mathsf{rop}\ \mathsf{p} \longrightarrow \mathsf{q}$ (here rop is the name of the operation, which takes a single parameter p and returns a value or list of values q). The algorithm partitions the operations into S and R based on whether they return a value (operations in R) or not (operations in S). The algorithms are designed to infer the complete functionality of these programs so that the regenerated program can serve as drop-in replacement for the original program.

As presented, the algorithm works with potential retrieve operations that take a single parameter and return a single value or list of values. It is straightforward to generalize the algorithms to work with potential retrieve operations with multiple parameters that may return multiple retrieved values. We also note that the algorithms as currently formulated assume that the application conforms to the target class of applications to infer — if presented with an application outside this class of applications, the algorithms will almost certainly fail to infer a correct model of application behavior.

The inference algorithms represent the inferred information with a collection of (conceptual) data structures. Examples include *SRP* (a set of inferred store/retrieve pairs that records which operations retrieve items stored by other operations, Section 3.1), *KRL* (a set of inferred tuples that records whether operations keep, replace, or accumulate items inserted under the same key, Section 3.2), *SDP* (a set of inferred store/delete pairs that record which operations delete items, Section 3.4), and *M* (a set of inferred tuples that identify operations that store items into the same map, Section 3.5). The regeneration algorithm works with these inferred data structures 3.6.

```
Inputs:
  $SP$-Seed Program
  $S = \{\mathsf{sop}_1\ \mathsf{p}^1_1 \ldots \mathsf{p}^1_{k_1}, \ldots, \mathsf{sop}_n\ \mathsf{p}^n_1 \ldots \mathsf{p}^n_{k_n}\}$
  $R = \{\mathsf{rop}_1\ \mathsf{p}_1 \longrightarrow \mathsf{q}_1, \ldots, \mathsf{rop}_m\ \mathsf{p}_m \longrightarrow \mathsf{q}_m\}$
Output:
  $SRP = \{\langle \mathsf{sop}_1, \mathsf{rop}_1, k_1, i_1, j_1 \rangle, \ldots,$
      $\langle \mathsf{sop}_l, \mathsf{rop}_l, k_l, i_l, j_l \rangle\}$
Algorithm:
  $SRP = \emptyset$
  for $\mathsf{sop}\ \mathsf{p}_1 \ldots \mathsf{p}_k \in S$
    for $\mathsf{rop}\ \mathsf{p} \longrightarrow \mathsf{q} \in R$
      choose distinct $v_1, \ldots, v_k$
      for $1 \leq i \leq k$
        $v = \mathsf{sop}\ v_1 \ \ldots\ v_k; \mathsf{rop}\ v_i \mid SP$
        for $1 \leq j \leq k$
          if $v = v_j$ or $v = [v_j]$
            $SRP = SRP \cup \{\langle \mathsf{sop}, \mathsf{rop}, k, i, j \rangle\}$
```

Figure 6. Store/Retrieve Pair *SRP* Inference Algorithm

3.1 Store/Retrieve Pair Inference Algorithm

Figure 6 presents the store/retrieve pair inference algorithm. The algorithm takes as inputs the seed program *SP*, a set of potential store/remove operations *S* (each of which may store or remove relations), with each operation of the form $\mathsf{sop}\ \mathsf{p}_1 \ldots \mathsf{p}_k$ (here sop is the name of the operation and $\mathsf{p}_1 \ldots \mathsf{p}_k$ are the names of the k parameters), and a set of potential retrieve operations R, with each operation of the form $\mathsf{rop}\ \mathsf{p} \longrightarrow \mathsf{q}$ (here rop is the name of the operation, which takes a single parameter p and returns a value or list of values q). The algorithm partitions the operations into S and R based on whether they return a value (operations in R) or not (operations in S). As presented, the algorithm works with potential retrieve operations that take a single parameter and return a single value or list of values. It is straightforward to generalize the algorithms to work with potential retrieve operations with multiple parameters that may return multiple retrieved values.

The algorithm produces as output a set of store/retrieve pairs *SRP*, with each pair of the form $\langle \mathsf{sop}, \mathsf{rop}, k, i, j \rangle$. Here sop is a potential store operation with k parameters that stores a relation that maps its i'th parameter to its j'th parameter. rop is a retrieve operation that, when given the

```
Inputs:
  $SP$-Seed Program
  $S = \{\mathsf{sop}_1\ \mathsf{p}^1_1 \ldots \mathsf{p}^1_{k_1}, \ldots, \mathsf{sop}_n\ \mathsf{p}^n_1 \ldots \mathsf{p}^n_{k_n}\}$
  $R = \{\mathsf{rop}_1\ \mathsf{p}_1 \rightarrow \mathsf{q}_1, \ldots, \mathsf{rop}_m\ \mathsf{p}_m \rightarrow \mathsf{q}_m\}$
  $SRP = \{\langle \mathsf{sop}_1, \mathsf{rop}_1, k_1, i_1, j_1\rangle, \ldots, \langle \mathsf{sop}_l, \mathsf{rop}_l, k_l, i_l, j_l\rangle\}$
Output:
  $KRL = \{\langle \mathsf{sop}_1, k_1, i_1, j_1, krl_1\rangle, \ldots, \langle \mathsf{sop}_l, k_l, i_l, j_l, krl_l\rangle\}$
Algorithm:
  $KRL = \emptyset$
  for $\langle \mathsf{sop}, \mathsf{rop}, k, i, j\rangle \in SRP$
    choose distinct $v_1, \ldots, v_k, u_1, \ldots, u_k$
    such that $v_i = u_i$
    $v = \mathsf{sop}\ v_1\ \ldots\ v_k; \mathsf{sop}\ u_1\ \ldots\ u_k;\ \mathsf{rop}\ v_i \mid SP$
    if $v = v_j$  $krl = \mathsf{Keep}$
    if $v = u_j$  $krl = \mathsf{Replace}$
    if $v = [v_j, u_j]$  $krl = \mathsf{List}$
    $KRL = KRL \cup \{\langle \mathsf{sop}, k, i, j, krl\rangle\}$
```

Figure 7. Keep/Replace/List *KRL* Inference Algorithm

i'th parameter (the key) of a previously executed sop operation, returns the j'th parameter (the stored value) of that operation.

The algorithm itself enumerates all potential store/retrieve pairs to collect all pairs that exhibit the required store/retrieve behavior. Specifically, it runs the the seed program *SP* (starting with empty maps) first on a potential store operation sop $v_1 \ldots v_k$, then on a potential retrieve operation rop v_i, and collects the resulting value v that the potential retrieve operation returns. We use the notation $v = \mathsf{sop}\ v_1 \ldots v_k;\ \mathsf{rop}\ v_i \mid SP$ to denote running the seed program *SP* on these two operations to obtain the returned value v. If the resulting value v matches one of the parameters v_i of the potential store operation, then the algorithm has found a store/retrieve pair (that it then collects into the output set of store/receive pairs *SRP*).

3.2 Keep, Replace, or List Inference Algorithm

The keep, replace, or list inference algorithm explores the behavior of the seed program when multiple relations with the same key are stored in the same map. The algorithm infers three different possible behaviors:

- **Keep:** Keep original relation and drop subsequent stores.
- **Replace:** Replace existing relation with new relation.
- **List:** Accumulate the values from multiple stores into a list of values stored under the key.

Figure 7 presents the keep, replace, or list inference algorithm. The algorithm takes as inputs the seed program *SP*, a set of potential store/remove operations *S*, a set of potential retrieve operations *R*, and the store/retrieve pairs *SRP* from the store/retrieve pair inference algorithm (Figure 6). It produces as output a set of tuples $\langle \mathsf{sop}, k, i, j, krl\rangle$, where sop is an operation with k parameters that stores a relation into some map. The i'th parameter is the key and the j'th parameter is the value of this stored relation. $krl \in \{\mathsf{Keep}, \mathsf{Replace}, \mathsf{List}\}$ specifies whether the operation keeps the original relation, replaces the original relation, or accumulates the stores into a list of values.

```
Inputs:
  $SP$-Seed Program
  $S = \{\mathsf{sop}_1\ \mathsf{p}^1_1 \ldots \mathsf{p}^1_{k_1}, \ldots, \mathsf{sop}_n\ \mathsf{p}^n_1 \ldots \mathsf{p}^n_{k_n}\}$
  $R = \{\mathsf{rop}_1\ \mathsf{p}_1 \rightarrow \mathsf{q}_1, \ldots, \mathsf{rop}_m\ \mathsf{p}_m \rightarrow \mathsf{q}_m\}$
  $SRP = \{\langle \mathsf{sop}_1, \mathsf{rop}_1, k_1, i_1, j_1\rangle, \ldots, \langle \mathsf{sop}_l, \mathsf{rop}_l, k_l, i_l, j_l\rangle\}$
Output:
  $SCP = \{\langle \mathsf{sop}_1, k_1, i_1, j_1, \mathsf{sop}'_1, k'_1, i'_1\rangle, \ldots,$
          $\langle \mathsf{sop}_o, k_o, i_o, j_o, \mathsf{sop}'_o, k'_o, i'_o\rangle\}$
Algorithm:
  $SCP = \emptyset$
  for $\langle \mathsf{sop}, \mathsf{rop}, k, i, j\rangle \in SRP$
    choose distinct $v_1, \ldots, v_k$
    for $\langle \mathsf{sop}'\ p_1\ \ldots\ p_{k'}\rangle \in S$
      for $1 \leq i' \leq k'$
        choose distinct $u_1, \ldots, u_{k'}$
        such that $v_i = u_{i'}$
        $v = \mathsf{sop}\ v_1 \ldots v_k; \mathsf{sop}'\ u_1 \ldots u_{k'}; \mathsf{rop}\ v_i \mid SP$
        if $v = \mathbf{Nil}$
          $SCP = SCP \cup \{\langle \mathsf{sop}, k, i, j, \mathsf{sop}', k', i'\rangle\}$
```

Figure 8. Store/Clear Pair *SCP* inference algorithm

The algorithm iterates over all of the store/retrieve pairs in *SRP* (from the store/retrieve pair inference algorithm) to find operations that store relations in some map. It executes the seed program *SP*, invoking the store operation twice with the same key but different values. It then retrieves the value stored under the key to determine if the store operations kept the first relation, replaced the first relation with the second relation, or accumulated the values into a list stored under the key.

3.3 Store/Clear Pair Inference Algorithm

In addition to storing relations, operations may also remove relations. If an operation removes relations based only on the key, we call the operation a *clear* relation; if the operation removes relations based on both the key and value, we call the operation a *delete* operation (Section 3.4).

Figure 8 presents the store/clear pair inference algorithm. The algorithm takes as inputs the seed program *SP*, a set of potential store/remove operations *S*, a set of potential retrieve operations *R*, and the store/retrieve pairs *SRP* from the store/retrieve pair inference algorithm (Figure 6).

The algorithm produces as output a set of store/clear pairs *SCP*, with each pair of the form $\langle \mathsf{sop}, k, i, j, \mathsf{sop}', k', i'\rangle$. Here sop is an operation with k parameters that stores a relation that maps its i'th parameter (the key) to its j'th parameter (the value). sop' is an operation with k' parameters that

```
Inputs:
  $SP$-Seed Program
  $S = \{\mathsf{sop}_1\ \mathsf{p}^1_1 \dots \mathsf{p}^1_{k_1}, \dots, \mathsf{sop}_n\ \mathsf{p}^n_1 \dots \mathsf{p}^n_{k_n}\}$
  $R = \{\mathsf{rop}_1\ \mathsf{p}_1 \rightarrow \mathsf{q}_1, \dots, \mathsf{rop}_m\ \mathsf{p}_m \rightarrow \mathsf{q}_m\}$
  $SRP = \{\langle \mathsf{sop}_1, \mathsf{rop}_1, k_1, i_1, j_1\rangle, \dots, \langle \mathsf{sop}_l, \mathsf{rop}_l, k_l, i_l, j_l\rangle\}$
  $SCP = \{\langle \mathsf{sop}_1, k_1, i_1, j_1, \mathsf{sop}'_1, k'_1, i'_1\rangle, \dots,$
          $\langle \mathsf{sop}_o, k_o, i_o, j_o, \mathsf{sop}'_o, k'_o, i'_o\rangle\}$
Output:
  $SDP = \{\langle \mathsf{sop}_1, k_1, i_1, j_1, \mathsf{sop}'_1, k'_1, i'_1, j'_1\rangle, \dots,$
          $\langle \mathsf{sop}_o, k_o, i_o, j_o, \mathsf{sop}'_o, k'_o, i'_o, j'_o\rangle\}$
Algorithm:
  $SDP = \emptyset$
  for $\langle \mathsf{sop}, \mathsf{rop}, k, i, j\rangle \in SRP$
    choose distinct $v_1, \dots, v_k$
    for $\langle \mathsf{sop}'\ p_1 \dots p_{k'}\rangle \in S$
      for $1 \leq i' \leq k'$, $1 \leq j' \leq k'$, $i' \neq j'$
        choose distinct $u_1, \dots, u_{k'}$
        such that $v_i = u_{i'}, v_j = u_{j'}$
        if $\langle \mathsf{sop}, k, i, j, \mathsf{sop}', k', i'\rangle \notin SCP$
          $v = \mathsf{sop}\ v_1 \dots v_k; \mathsf{sop}'\ u_1 \dots u_{k'}; \mathsf{rop}\ v_i \mid SP$
          if $v =$ Nil or $v = [\ ]$
            $SDP = SDP \cup \{\langle \mathsf{sop}, k, i, j, \mathsf{sop}', k', i', j'\rangle\}$
```

Figure 9. Store/Delete Pair (*SDP*) inference algorithm

clears the stored relation from the map. To clear the relation, the keys, specifically the i'’th parameter of sop′ and i’th parameter of sop, must have the same value.

The algorithm first uses the inferred store/retrieve pairs *SRP* to iterate over all operations that insert relations into maps. It then iterates over all potential operations that may clear the stored relation, executing the seed program *SP* on the two operations sop $v_1 \ \dots \ v_k$ (the operation that stores the relation) and sop′ $u_1 \ \dots \ u_{k'}$ (the potential clear operation) in sequence. The algorithm then executes the retrieve operation rop v_i from the store/retrieve pair to determine if the potential clear operation actually cleared the stored relation. If the retrieve operation returns nothing (v = **Nil**) after the seed program executes the store and potential clear operation, the algorithm has found a store/clear pair that it then collects into the output set of store/clear pairs *SCP*. To avoid finding operations that delete relations based on both the key and the value, the algorithm ensures that all of the parameters $v_1, \dots, v_k$ and $u_1, \dots, u_{k'}$ are distinct (with the exception of the key parameters $v_i = v_{i'}$) so that the potential clear operation will not be given the value from the stored key/value relation.

3.4 Store/Delete Pair Inference Algorithm

The store/clear pair inference algorithm (Figure 8) infers operations that remove relations based on a given key regardless of the value to which the key maps. Some operations, however, remove relations based not only on the key, but also on the value. Such operations are often used, for example, to delete specific values within a list of values accumulated under the same key. We call such operations *delete* operations (as opposed to the clear operations from Section 3.3, which remove relations based only on the key, not the value). The **add** and **drop** operations from the example in Section 2 are one example of a store/delete pair. The store/delete pair inference algorithm infers operations that delete relations based on both the key and the value.

Figure 9 presents the store/delete pair inference algorithm. The algorithm takes as inputs the seed program *SP*, a set of potential store/remove operations *S*, a set of potential retrieve operations *R*, the store/retrieve pairs *SRP* from the store/retrieve pair inference algorithm (Figure 6), and the store/clear pairs *SCP* from the store/clear pair inference algorithm (Figure 8).

The algorithm produces as output a set of store/delete pairs *SDP*, with each pair of the form ⟨sop, k, i, j, sop′, k', i', j'⟩. Here sop is an operation with k parameters that stores a relation that maps its i’th parameter (the key) to its j’th parameter (the value). sop′ is an operation with k' parameters that deletes the stored relation from the map. To delete the relation, the i'’th and j'’th parameters of sop′ must be the same as the i’th and j’th parameters of sop, respectively. Delete operations sop′ typically work with lists of values stored under the same key to delete single list values while leaving the other values in the list intact. The **drop** operation in Section 2 is an example of an operation that deletes a relation based on both the key and the value.

The algorithm uses the inferred store/retrieve pairs *SRP* (Algorithm 6) and *S* to enumerate the possible store/delete pairs. It first runs sop $v_1 \dots v_k$, which inserts the relation, then sop′ $u_1 \dots u_{k'}$, which may delete the relation. If then runs rop v_i and checks the return value to see if the relation was deleted. It collects pairs in which the relation was deleted into the output set of inferred store/delete pairs *SDP*, skipping pairs with a corresponding store/clear pair in *SCP* — the algorithm only collects store/delete pairs in which both the key and the value must match for the operation to delete the stored relation.

3.5 Map Inference Algorithm

The store/retrieve pair inference algorithm (Figure 6) finds operations that insert relations into some map. It does not, however, attempt to determine which operations insert relations into the same map. This information is critical for code regeneration — if two different operations insert relations into the same map, the code regenerator must ensure that the regenerated operations access the same map.

The map inference algorithm finds operations that insert relations into the same map. The algorithm enumerates pairs of operations that store relations to find operations that store relations into the same map. It finds these operations by executing the seed program *SP* twice. The first execution

```
Inputs:
  $SP$-Seed Program
  $S = \{\mathsf{sop}_1\ \mathsf{p}^1_1 \ldots \mathsf{p}^1_{k_1}, \ldots, \mathsf{sop}_n\ \mathsf{p}^n_1 \ldots \mathsf{p}^n_{k_n}\}$
  $R = \{\mathsf{rop}_1\ \mathsf{p}_1 \rightarrow \mathsf{q}_1, \ldots, \mathsf{rop}_m\ \mathsf{p}_m \rightarrow \mathsf{q}_m\}$
  $SRP = \{\langle \mathsf{sop}_1, \mathsf{rop}_1, k_1, i_1, j_1\rangle, \ldots, \langle \mathsf{sop}_l, \mathsf{rop}_l, k_l, i_l, j_l\rangle\}$
Output:
  $M = \{\{\langle \mathsf{sop}^1_1, k^1_1, i^1_1, j^1_1\rangle, \ldots, \langle \mathsf{sop}^1_{m_1}, k^1_{m_1}, i^1_{m_1}, j^1_{m_1}\rangle\},$
                 $\ldots,$
       $\{\langle \mathsf{sop}^n_1, k^n_1, i^n_1, j^n_1\rangle, \ldots, \langle \mathsf{sop}^n_{m_n}, k^n_{m_n}, i^n_{m_n}, j^n_{m_n}\rangle\}\}$
Algorithm:
  $M = \{\{\langle \mathsf{sop}, k, i, j\rangle\}\ .\ \langle \mathsf{sop}, \mathsf{rop}, k, i, j\rangle \in SRP\}$
  for $\langle \mathsf{sop}_1, \mathsf{rop}_1, k_1, i_1, j_1\rangle \in SRP$
    for $\langle \mathsf{sop}_2, \mathsf{rop}_2, k_2, i_2, j_2\rangle \in SRP$
      choose distinct $v_1, \ldots, v_{k_1}, u_1, \ldots, u_{k_2}$
      such that $v_{i_1} = u_{i_2},\ v_{j_1} = u_{j_2}$
      $v = \mathsf{sop}_1\ v_1\ \ldots\ v_{k_1}; \mathsf{rop}_1\ v_{i_1}\ |\ SP$
      $u = \mathsf{sop}_2\ u_1\ \ldots\ u_{k_2}; \mathsf{rop}_1\ v_{i_1}\ |\ SP$
      if $v = u$
        $M = \mathbf{Union}(M, \langle \mathsf{sop}_1, k_1, i_1, j_1\rangle, \langle \mathsf{sop}_2, k_2, i_2, j_2\rangle)$
```

Figure 10. Map (M) inference algorithm

executes the first operation, then the corresponding retrieve operation. The second execution executes the second operation, then again the corresponding retrieve operation for the first operation (which retrieves the stored value from the same map that the first operation stored into). If both executions return the same value, the algorithm infers that the two operations store into the same map. The algorithm uses the output *SRP* of the store/retrieve inference algorithm to find operations that store relations.

Figure 10 presents the map inference algorithm. The algorithm takes as inputs the seed program *SP*, a set of potential store/remove operations *S*, a set of potential retrieve operations *R*, and the store/retrieve pairs *SRP* from the store/retrieve pair inference algorithm (Figure 6). It represents each map as a set of tuples *KRL* $\{\langle \mathsf{sop}_1, k_1, i_1, j_i\rangle, \ldots, \langle \mathsf{sop}_m, k_m, i_m, j_m\rangle\}$. Each tuple $\langle \mathsf{sop}, k, i, j\rangle$ represents a store operation sop with k parameters that stores a relation from its i'th parameter to the j'th parameter.

The algorithm works with M, which is a set of sets of tuples. Given a set $T \in M$, all tuples in T represent operations that store into the same map. M partitions the set of tuples (no tuple is in two sets in M). The algorithm initializes M to contain singleton sets of tuples (representing operations that store into a different maps), then unions these sets as it finds pairs of operations that store into the same map. It uses the operation **Union**$(M, \langle \mathsf{sop}_1, i_1, j_1\rangle, \langle \mathsf{sop}_2, i_2, j_2\rangle)$, which finds the sets $T_1, T_2 \in M$ that contain $\langle \mathsf{sop}_1, i_1, j_1\rangle \in T_1$ and $\langle \mathsf{sop}_2, i_2, j_2\rangle \in T_2$, then computes the union of T_1 and T_2 to return $(M - \{T_1, T_2\}) \cup \{T_1 \cup T_2\}$.

3.6 Regeneration Algorithm

Working with the inferred information, the code regeneration algorithm performs the following steps. The specific details of each step depend on the precise characteristics of the target computing environment. The code regeneration algorithm encapsulates the knowledge of how to use the computing environment for the target computation, with specific implementation decisions about how to best use this environment, such as whether and where to insert caches, left to the implementor of the regeneration code (potentially guided by automated performance measurement experiments on different versions of regenerated code executing on the target computing platform).

- **Initialization Code:** Many computing environments and packages require complex initialization code sequences. The code regeneration algorithm automatically generates this code.
- **Map Regeneration:** Working with the output M of the map inference algorithm, the code regenerator generates a map for each set of tuples $T \in M$. The specific implementation of each map will vary depending on the target computing environment. Examples of potential map implementations include Python data structures, Redis [Redis 2018] maps, and SQL tables.
- **Command Loop Regeneration:** The regenerated command loop reads each command and its parameters, then invokes an (automatically generated) procedure that implements the command. Depending on the characteristics of the target computing environment, the code regenerator can systematically generate (potentially new) input validation checks and recovery code for malformed inputs.
- **Store Regeneration:** For each tuple in the inferred keep-/replace/list set *KRL*, the code regenerator generates code that stores the inferred relation in the inferred map as determined by M. Specifically, for each tuple $\langle \mathsf{sop}, k, i, j, krl\rangle \in KRL$, the regenerated code for sop stores a relation that maps the i'th parameter (the key) to the j'th parameter (the value) in the map for the set $T \in M$, where $\langle \mathsf{sop}, k, i, j\rangle \in T$. *krl* determines whether the operation keeps, replaces, or accumulates in a list any existing relations with the same key.
- **Clear Regeneration:** For each tuple in the inferred store/-clear set *SCP*, the code regenerator generates code that clears relations with the inferred key from the inferred map (as determined by M). Specifically, for each entry $\langle \mathsf{sop}, k, i, j, \mathsf{sop}', k', i'\rangle \in SCP$, the regenerated code for sop′ clears relations whose key is the i''th parameter of sop′ from the map for the set $T \in M$, where $\langle \mathsf{sop}, k, i, j\rangle \in T$.
- **Delete Regeneration:** For each tuple in the inferred store/delete set *SDP* the code regenerator generates code that deletes relations with the inferred key and value from the inferred map (as determined by M). Specifically, for

```
  elif cmd == "nameByYear":
    id, year = list[1:3]
    if years[id] >= year:
      print names[id]

  elif cmd == "addWithName":
    id, name, num = list[1:3]
    if names[id] == name:
      if not id in classes:
        classes[id] = []
      if not (num in classes[id]):
        classes[id].append(num)
```

Figure 11. Example Python seed program extensions with boolean conditions

```
@app.route('/nameByYear', methods = ['POST'])
def nameByYear__mapping():
  id = request.json['id']
  year = request.json['year']
  data = {}
  for row in query_db('SELECT * FROM students WHERE id =
      ? AND year >= ?', (id, year)):
    data['name'] = row['name']
  resp = Response(json.dumps(data), status=200, mimetype
      ='application/json')
  return resp

@app.route('/addWithName', methods = ['POST'])
def add_mapping():
  id = request.json['id']
  name = request.json['name']
  num = request.json['num']
  for row in query_db('SELECT * FROM students WHERE id =
      ? AND name = ?', (id, name)):
    classes = row['classes'] + str(num) + ';'
    write_db('UPDATE students SET classes = ? WHERE id =
        ?', (classes, id))
  return ('', 200)
```

Figure 12. HTTP request handlers for commands nameByYear and addWithName

each entry $\langle \mathsf{sop}, k, i, j, \mathsf{sop}', k', i', j' \rangle \in SDP$, the regenerated code for sop' clears relations whose key and value are the i''th and j''th parameters of sop' from the map for the set $T \in M$, where $\langle \mathsf{sop}, k, i, j \rangle \in T$.

3.7 Extensions For Conditional Operations

Apart from implementing the algorithms in the previous sections, we also implemented a preliminary version of an extension that infers commands with conditional operations that execute only when certain boolean conditions hold. For example, this extension allows the seed program in Section 2 to have additional commands as in Figure 11:

- **nameByYear (id, year) → name:** Retrieve the name of the student with the given id, but only when the student's graduation year is earlier than the specified year.
- **addWithName (id, name, class):** Add a class to the list of classes for which the student is registered. The student is identified by the student's id and validated by providing the correct name.

The **nameByYear** operation is conditioned on the **year** parameter being less than or equal to the corresponding value stored by the **enroll** operation (Figure 1). The **addWithName** operation is conditioned on the **name** parameter being equal to the corresponding value stored by the **enroll** operation.

Figure 12 presents the regenerated HTTP request handlers for these commands, in Python using Flask [Flask 2018]. Each request handler corresponds to a command in the seed program. They contain the boilerplate code to convert inputs from JSON and convert outputs to JSON. They perform the inferred operations on the database using SQL queries. These SQL queries contain WHERE clauses that enforce the inferred conditions for an operation to execute.

The extended SRP inference algorithm works with retrieve operations that enforce boolean conditions of the following form. Each input parameter may be compared against at most one existing value stored in a map. The retrieve operation may check if the input parameter is equal to, greater than, greater than or equal to, less than, less than or equal to, or unequal to the stored value. If all these boolean conditions hold, the retrieve operation returns the corresponding value from the map.

Figure 13 presents the extended SRP algorithm that infers boolean conditions in retrieve operations. The algorithm takes as inputs the seed program SP, a set of potential store/remove operations S, with each operation of the form $\mathsf{sop}\ \mathsf{p}_1 \ldots \mathsf{p}_k$, and a set of potential retrieve operations R, with each operation of the form $\mathsf{rop}\ \langle \mathsf{r}_1, \ldots \mathsf{r}_t \rangle \longrightarrow \mathsf{q}$. Each retrieve operation takes a list of parameters $\mathsf{r}_1, \ldots \mathsf{r}_t$ and returns a value q.

The algorithm produces as outputs a set of store/retrieve pairs SRP, with each pair of the form $\langle \mathsf{sop}, \mathsf{rop}, k, t, C, J \rangle$. Here sop is a potential store operation with k parameters that stores the J'th parameter in a map. rop is a retrieve operation with t parameters that, when given parameters that satisfy the boolean conditions implied by C (see below), returns the J'th parameter (the stored value) of the sop operation. The boolean condition C contains a set of pairs, with each pair of the form $\langle\, i, L, E, G, j \,\rangle$ which describes whether the i'th parameter of rop is allowed to be less than (when L is true), equal to (when E is true), or greater than (when G is true) the j'th parameter of sop. Only when these conditions hold will the rop operation return the stored value.

The algorithm enumerates all potential store/retrieve pairs to collect all pairs that exhibit the required store/retrieve behavior over the same map. Specifically, it runs the seed program SP (starting with empty maps) first on a potential store operation $\mathsf{sop}\ v_1 \ldots v_k$, then on a potential retrieve operation $\mathsf{rop}\ x_1 \ldots x_t$, and collects the resulting value v that the potential retrieve operation returns. We use the same notation as in Section 3.1. If the resulting value v matches one of the parameters v_j of the potential store operation, then the algorithm has found a potential store/retrieve pair (that it then collects into the J variable for further analysis). Here the rop parameters $x_1 \ldots x_t$ are chosen to differ slightly (the ways they differ are specified by $o_1 \ldots o_t$) from a subset of the values used for $v_1 \ldots v_k$ (the correspondence is specified by $w_1 \ldots w_t$). The algorithm exhaustively checks all potential comparison results and collects the observed parameter relations in *LEG*. *LEG* contains a set of pairs, with each pair in the form $\langle\, i, o_i, w_i \,\rangle$ which describes that when the i'th parameter of rop has the o_i relation with the w_i'th parameter of sop, the retrieve operation rop successfully returned a value. Then, the algorithm enumerates the information collected in *LEG* for each pair of parameters (one for sop and one for rop). If changing their relation has any effect on the execution results, then the algorithm infers that these two parameters has a condition that must be satisfied for the rop operation to return a value. Specifically, the algorithm observes whether rop executed under three sets of different inputs: when the i'th parameter of rop is less than, equal to, or greater than the j'th parameter of sop. If these three results differ, then the retrieve operation has checked the relation between these two parameters. This result is stored in C.

We next describe how to integrate the boolean condition inference into other parts of the algorithm. In Section 3.1, each retrieve operation executes when a given parameter matches the key of an existing record. This key is the parameter of the corresponding store operation for uniquely identifying a record in the map. After extending the algorithm to support conditional retrieve operations, we note that the retrieve operations may use more than one parameter to identify records. In other words, keys may now contain multiple parameters. Hence, the store operations need to be extended accordingly. A key step is to infer the appropriate set of parameters that comprise a key. For example, in each store/retrieve pair, the key used by the store operation may differ from the retrieve operation parameters that enforce the equal condition. To address this problem, the algorithm calls the store operation twice, each time using slightly different parameter values. The algorithm then calls the corresponding retrieve operation to check whether the first store operation was overwritten by the second store operation. The algorithm repeats this process and concludes with the minimal set of store operation parameters needed to uniquely identify a record.

The *KRL*, *SCP*, *SDP*, and *M* algorithms can also be extended to support the additional functionality for boolean conditionals and multi-parameter keys.

```
Inputs:
  $SP$-Seed Program
  $S = \{\mathsf{sop}_1\ \mathsf{p}^1_1 \ldots \mathsf{p}^1_{k_1}, \ldots, \mathsf{sop}_n\ \mathsf{p}^n_1 \ldots \mathsf{p}^n_{k_n}\}$
  $R = \{\mathsf{rop}_1\ \langle \mathsf{r}^1_1, \ldots, \mathsf{r}^1_{t_1} \rangle \rightarrow \mathsf{q}_1, \ldots,$
       $\mathsf{rop}_m\ \langle \mathsf{r}^m_1, \ldots, \mathsf{r}^m_{t_m} \rangle \rightarrow \mathsf{q}_m\}$
Output:
  $SRP = \{\langle \mathsf{sop}_1, \mathsf{rop}_1, k_1, t_1, C_1, j_1 \rangle, \ldots,$
       $\langle \mathsf{sop}_l, \mathsf{rop}_l, k_l, t_l, C_l, j_l \rangle\}$
Algorithm:
  $SRP = \emptyset$
  for sop $\mathsf{p}_1 \ldots \mathsf{p}_k \in S$
    for rop $\langle \mathsf{r}_1 \ldots \mathsf{r}_t \rangle \rightarrow \mathsf{q} \in R$
      choose distinct $v_1, \ldots, v_k$
      $J$ = Nil
      for ordered subset $w_1, \ldots, w_t$ of $1, \ldots, k$
        for $o_1, \ldots, o_t \in \{<, =, >\}$
          choose $x_1, \ldots, x_t$ such that
              $x_i\ o_i\ v_{w_i}$ for $i = 1, \ldots, t$
          $v$ = sop $v_1 \ldots v_k$;rop $x_1 \ldots x_t \mid SP$
          if $v \neq$ Nil
            for $1 \leq j \leq k$
              if $v = v_j$
                $J = j$
            for $1 \leq i \leq t$
              $LEG = LEG \cup \{\langle\, i, o_i, w_i \,\rangle\}$
      if $J \neq$ Nil
        $C = \emptyset$
        for $1 \leq i \leq t,\ 1 \leq j \leq k$
          $L$ = (whether $\langle\, i, <, j \,\rangle \in LEG$)
          $E$ = (whether $\langle\, i, =, j \,\rangle \in LEG$)
          $G$ = (whether $\langle\, i, >, j \,\rangle \in LEG$)
          if not $L = E = G$
            $C = C \cup \{\langle\, i, L, E, G, j \,\rangle\}$
        $SRP = SRP \cup \{\langle \mathsf{sop}, \mathsf{rop}, k, t, C, J \rangle\}$
```

Figure 13. Extended Store/Retrieve Pair Inference Algorithm with Boolean Conditions

3.8 Discussion

The inference algorithms highlight the utility of working with a seed program instead of a set of given input/output pairs. The ability to repeatedly run the program on chosen sets of inputs and pairs of operations (a form of active learning) enables the algorithms to comprehensively explore the behavior of the seed program to infer the (conceptual) maps that the program maintains and how the commands manipulate these maps. There is no need to deal with behavior that is only partially exposed by given input/output pairs.

Because the inference algorithms interact with the seed program only by presenting it with inputs and observing the outputs, the seed program can be implemented in any language and use any mechanism to implement the inferred maps and operations. The approach therefore supports a wide range of developers with a wide range of technical preferences and skills. The inference algorithms also impose essentially no scalability requirements on the seed program — the generated inputs contain at most three operations per execution of the seed program (Figures 8 and 9).

4 Related Work

The closest related work uses gray box techniques to derive models of program components that store and retrieve data from either an external, observable database [Shen and Rinard 2017, 2018a] or instrumented data structures [Wu 2018]. The research presented in this paper differs in that (1) it uses black box techniques to work with programs with hidden internal maps that are not accessible to the inference algorithm, (2) it infers and regenerates programs that work with maps, and (3) it works with computations that store and retrieve data from maps, not computations whose behavior is captured by sequences of data access behavior.

Model-Driven Engineering. In model-driven engineering (MDE) [Brambilla et al. 2012; Schach 2007; Schmidt 2006], developers specify functionality in high-level models, often using domain-specific languages or formalisms such as Unified Modeling Language (UML), which are then used to generate low-level platform-dependent implementations. Use cases of MDE include migrating software across different platforms [Schach 2007] and automating the code generation for CRUD (create/read/update/delete) applications [Albert et al. 2010; Django 2018; Rails 2018]. Like the regenerators presented in this paper, MDE code generators encapsulate the knowledge of how to use specific computational platforms and enable the automatic generation of code for multiple platforms. In contrast to having developers work directly with high-level domain-specific models, our approach starts from an existing implementation, then infers the program functionality as a black box and regenerates a new implementation.

Software Modernization. Software modernization [Cánovas Izquierdo and García Molina 2014; Fuentes-Fernández et al. 2012; Sánchez Ramón et al. 2014] analyzes the source code of a legacy program, translates the program into a high-level modeling language, then uses this representation to generate a new program that implements the functionality in a more modern language. The translation strictly follows syntactic cues and usually requires human intervention. Our approach, in contrast, (1) works with the given implementation without analyzing code and (2) regenerates an augmented computation with additional error and security checks that implements the core functionality with complex new software components that execute on modern target platforms.

Partial Program Rejuvenation. Helium uses dynamic instrumentation to extract the functionality of computational stencil kernels embedded within production binaries [Mendis et al. 2015]. It then replaces the stencil kernel with a computation expressed in Halide [Ragan-Kelley et al. 2013]. The goal is to replace the legacy implementation with a version optimized for modern computational platforms. Program fracture and recombination [Amidon et al. 2015] works with multiple applications to automatically find efficient, sophisticated, and/or robust implementations of subcomputations across applications, then transfers subcomputations across implementations to maximize efficiency or robustness. A goal is to automatically replace simple code that executes on a single machine with more complex code that operates on parallel or distributed computing platforms. Our approach, in contrast, (1) models the full computation and regenerates the entire application, augmented as appropriate, (2) can work with incomplete or buggy implementations of the original program, and (3) targets programs that store and retrieve data in maps.

Stateless Model Extraction. Model extraction algorithms use queries to construct representations for programs, where the representations are stateless functions such as decision trees [Craven and Shavlik 1995; Tramèr et al. 2016] or symbolic rules [Towell and Shavlik 1993]. Model compression algorithms [Buciluă et al. 2006; Hinton et al. 2015] use machine learning models, such as neural networks, to mimic a machine learning model, often by generating inputs (training data) and observing the outputs from the given model. Our approach, in contrast, (1) infers stateful models that store/retrieve data across multiple queries and (2) regenerates a new program or programs, augmented as appropriate, that implement the core functionality on new implementation platforms.

Partial Model Learning. Algorithms for learning black-box state machines [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, using finite automata with states and transitions. State fuzzing tools [Aarts et al. 2013; De Ruiter and Poll 2015] are used to hypothesize state machines for given program implementations, which can help developers to discover bugs such as spurious state transitions. These algorithms extract partial models of the given programs. Our approach, in contrast, (1) extracts a complete representation of the core functionality, which, in turn, enables the regeneration (and replacement) of the initial program, (2) can work with programs with defects or that only partially implement the core functionality, (3) regenerates a new program or programs, augmented as appropriate, that implement the core functionality without defects or security vulnerabilities on new implementation platforms, and (4) represents the inferred programs as commands and key-value maps, which can capture a wide range of programs that store and retrieve data.

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]. Our approach, in contrast, uses active learning to reverse engineer a seed program (in effect using the seed program as a specification). Because this approach is not constrained by a given set of input/output pairs, it can select the inputs to purposefully target and resolve ambiguities.

[Alur et al. 2013] identifies a range of program synthesis problems for which it is productive to structure the search space as a domain-specific language and presents a framework for this approach. Our approach similarly uses a domain to structure the search space. Unlike the examples presented in [Alur et al. 2013], our approach exploits the structure of the domain to obtain an inference algorithm that uses active learning to progressively refine the knowledge of the map structures and the store/retrieve/remove operations.

Oracle-guided synthesis as implemented in Brahma [Jha et al. 2010] interacts with a program to infer a model that completely captures the behavior of the program. Our approach deploys an inference algorithm guided by assumptions about the store/retrieve behavior. Our approach maintains a structured representation that captures the inferred relations between maps, parameters, and commands. 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.

Inferring Models for Programs. [Gehr et al. 2015] present an active learning technique for learning commutativity specifications of data structures. [Bastani et al. 2017] present a technique for learning program input grammars. [Bastani et al. 2018] present a technique for learning points-to specifications. [Jeon et al. 2016] present a technique for learning models of design patterns that Java computations implement. Unlike our approach, all of these techniques focus on characterizing specific aspects of program behavior and do not aspire to capture the complete behavior of the application.

Mimic [Heule et al. 2015] traces the memory accesses of an opaque function to synthesize a model of the traced function. Our approach, in contrast, treats the seed program as a black box. Mimic 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. Our approach 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 the required domain.

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 defects. BuzzFuzz [Ganesh et al. 2009] generates inputs that target defects from coding oversights at the boundary between application and library code. DIODE [Sidiroglou-Douskos et al. 2015] generates inputs that target integer overflow errors. All these techniques dynamically analyze the execution of the program and use the resulting information to guide the input generation. Our approach, in contrast, (1) works with the given implementation as a black box, without analyzing code, (2) extracts a representation of the core functionality, and (3) regenerates a new program or programs, augmented as appropriate, that implement the core functionality without defects or security vulnerabilities on new implementation platforms.

5 Conclusion

Modern software systems are characterized by the pervasive use of complex components with arcane interfaces. Most developers that work with such systems spend their time constructing appropriate Google search terms to find previously developed code that they can copy and adapt for their needs.

We propose to encapsulate the knowledge of how to use modern complex systems inside a regenerator that works with an abstract representation of the core functionality of the program. This regenerator produces augmented programs that contain systematically generated security and input validation checks and implement graphical web interfaces. The abstract functionality can be inferred either from existing programs or from simple text-based programs implemented in simple computing environments. This approach promotes a more meaningful and powerful form of code reuse and enables programmers to focus on the core functionality that their programs implement.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This research was supported by DARPA (Grant FA8650-15-C-7564).

References

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

Manoli Albert, Jordi Cabot, Cristina Gómez, and Vicente Pelechano. 2010. Automatic generation of basic behavior schemas from UML class diagrams. *Software & Systems Modeling* 9, 1 (2010), 47–67.

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.

P. Amidon, E. Davis, S. Sidiroglou-Douskos, and M. Rinard. 2015. Program fracture and recombination for efficient automatic code reuse. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2015.7396314

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

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

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

L. A. Belady and M. M. Lehman. 1976. A model of large program development. *IBM Systems Journal* 15, 3 (1976), 225–252. https://doi.org/10.1147/sj.153.0225

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.

Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering* 1, 1 (2012), 1–182.

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

Javier Luis Cánovas Izquierdo and Jesús García Molina. 2014. Extracting Models from Source Code in Software Modernization. *Softw. Syst. Model.* 13, 2 (May 2014), 713–734. https://doi.org/10.1007/s10270-012-0270-z

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

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

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.

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.

Django. 2018. The Web framework for perfectionists with deadlines | Django. https://www.djangoproject.com/.

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

Flask. 2018. Flask. http://flask.pocoo.org/.

Rubén Fuentes-Fernández, Juan Pavón, and Francisco Garijo. 2012. A Model-driven Process for the Modernization of Component-based Systems. *Sci. Comput. Program.* 77, 3 (March 2012), 247–269. https://doi.org/10.1016/j.scico.2011.04.003

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.

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.

Steve Lohr. 2017. Where Non-Techies Can Get With the Programming. The New York Times. https://nyti.ms/2oxp31L.

Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. 2015. Helium: Lifting High-performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).* ACM, New York, NY, USA, 391–402. https://doi.org/10.1145/2737924.2737974

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

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

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13).* ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

Ruby On Rails. 2018. Ruby on Rails | A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern. https://rubyonrails.org/.

Redis. 2018. Redis. https://redis.io/.

Martin Rinard and Jiasi Shen. 2017. *Inference and Regeneration of Programs that Store and Retrieve Data.* Technical Report. http://hdl.handle.net/1721.1/108383

Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. 2014. Model-driven Reverse Engineering of Legacy Graphical User Interfaces. *Automated Software Engg.* 21, 2 (April 2014), 147–186. https://doi.org/10.1007/s10515-013-0130-2

Stephen R. Schach. 2007. *Object-Oriented and Classical Software Engineering* (7 ed.). McGraw-Hill, Inc., New York, NY, USA.

Douglas C Schmidt. 2006. Model-driven engineering. (2006).

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

Jiasi Shen and Martin Rinard. 2017. *Inference and Regeneration of Programs that Manipulate Relational Databases.* Technical Report. http://hdl.handle.net/1721.1/111067

Jiasi Shen and Martin Rinard. 2018a. *Using Active Learning to Synthesize Models of Applications That Access Databases.* Technical Report. http://hdl.handle.net/1721.1/117593

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

Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 95–105.

Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* 43–54.

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

Megan Smith. 2016. Computer Science For All. The White House. https://obamawhitehouse.archives.gov/blog/2016/01/30/computer-science-all.

StackOverflow. 2018. Stack Overflow. http://stackoverflow.com/.

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.

Jerry Wu. 2018. *Using Dynamic Analysis to Infer Python Programs and Convert Them into Database Programs.* Master's thesis. Massachusetts Institute of Technology.

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.

Stuart Zweben and Betsy Bizot. 2016. 2015 Taulbee Survey: Continued Booming Undergraduate CS Enrollment; Doctoral Degree Production Dips Slightly. *Computing Research News* 28, 5 (may 2016).