Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2017-006 April 24, 2017

# Inference and Regeneration of Programs that Store and Retrieve Data

Martin Rinard and Jiasi Shen

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

# Inference and Regeneration of Programs that Store and Retrieve Data

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

Jiasi Shen
EECS & CSAIL
MIT
jiasi@csail.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 both (a) boilerplate code for the complex programming interfaces that the target computing platform presents and (b) systematic error and vulnerability checking code that makes the new implementations robust and secure. 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.

## 1 Introduction

Within the last decade, undergraduate computer science enrollments, both within and outside the major, have dramatically increased [54]. 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 [32, 45].

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 [6]). 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 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 uses 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 our 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 much 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 far fewer people, and more difficult to use for everyone regardless of their skill level. Our approach is founded on several principles:

- **Program Inference via Active Learning:** Starting with a *seed program* that (mostly) implements the desired core functionality, the system reverse engineers the seed program to infer a representation of the core functionality. It uses active learning to drive the process — because the specification is a program, it is possible to design algorithms that systematically interact with the program to learn the core functionality that it 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. 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) analyze the program and/or its execution.
- **Noisy, Partial Specifications:** The inference algorithm should be designed to tolerate noise (in the form of implementation errors or overlooked corner cases) and partial implementations of the desired core functionality. The algorithm must therefore be able to isolate the desired common case behavior of the seed program and infer a general specification from that isolated behavior, all while identifying and discarding undesirable behavior (noise) that should not be part of the specification.

  Such algorithms relieve the developer of the seed program of the need to consider and implement obscure corner cases. The developer can instead simply implement the core common case functionality while omitting error checking and code that handles corner cases. An advantage is that implementing simply the core functionality is often substantially easier than implementing a robust program that considers and correctly handles all cases.
- **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 much 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 will reinterpret standard constructs to translate them into implementations that operate successfully in the new, more complex target context.

### 1.1 Scope

The initial scope of the approach is programs that implement simple core functionality (such as storing and retrieving data or performing simple calculations over that stored data) on complex hardware platforms. We anticipate several use cases:

- **New Software:** In this use case, the seed implementation is developed from scratch, typically 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.
- **Legacy Systems:** 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).
- **Targeted Functionality Extraction:** In this use case, the seed program implements a range of functionality, only part of which comprises the desired core functionality. In this case 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.

In the longer term, we expect the scope to grow in several directions. First, we expect the program inference algorithms to become progressively more sophisticated to be able to successfully infer a larger range of programs. Second, we expect

the program regeneration and reinterpretation algorithms to grow in sophistication to deliver code with increased functionality and code that operates in more complex environments. Third, we expect the approach to generalize to learn core functionality from multiple applications, then merge them into a single unified regenerated application.

Another direction is learning and regenerating system components. Consider, for example, a standard database-backed web application. Instead of interacting only with the user interface to learn the end to end application functionality, an alternative approach would instrument the application to also observe the interactions between the application and the database. Leveraging the availability of these observed interactions, it would then learn and regenerate the core functionality of the front end, leaving the database intact.

### 1.2 Programs as Specifications

An alternative perspective is that we propose to use programs as (partial, noisy) specifications of the desired core functionality. In this context our approach has the advantage that developing programs is, in comparison with using standard specification languages based on formal logic, 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.

Of course, it is common knowledge that developers struggle to produce correct programs. The inference algorithm must therefore tolerate the presence of defects. One baseline strategy is to simply ignore program crashes. Another strategy is to work with a circumscribed program representation that rules out many incorrect programs that the inference algorithm might otherwise infer. Yet another strategy would develop a probabilistic model of correct programs and infer only programs that are likely to be correct.

### 1.3 Encapsulated Knowledge

Yet another perspective is that 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 [6]. 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.

### 1.4 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 rapidly becoming a boring, mind-numbing activity in which developers spend their time searching (in many cases more or less mindlessly) for arcane code sequences that (for often poorly understood reasons) happen to deliver something close to the desired effect. Automating the generation of these code sequences can free developers to focus on the fascinating 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.

### 1.5 Paper Structure

The remainder of the paper is structured as followed. 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 two regenerations: 1) a text-based C regeneration that stores the registration information in a Redis database and 2) a regeneration that implements a graphical web browser interface using HTML, CSS, Go, JavaScript, and Redis. 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. All of the code discussed in this section and in the appendix (with the exception of the python seed program) was automatically regenerated by our implemented program inference and regeneration system. Figure 1 presents a high-level structure of this system. This system starts with the interface, in the form of operations (parameterized commands) that the registration system should implement for its users:

- **enroll name id:** Given a student's name and id, enroll the student to take classes.

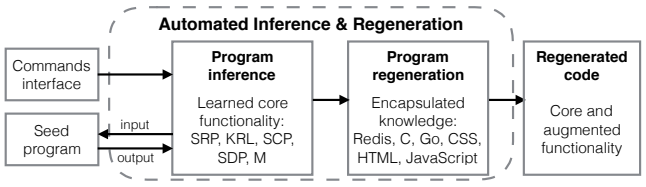


**Figure 1.** High-level structure of the program inference and regeneration system

- **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 identified student's registration list.
- **classes id → class:** Retrieve the class or classes for which the identified student is currently registered.
- **name id → name:** Retrieve the name of the identified student given the student's id.
- **id name → id:** Retrieve the id of the student with the given name.

Here each operation has a command (enroll, add, drop, classes, name, or id), parameters (name, id, and/or class), and outputs (name, id, and/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 parameters of the add and drop operations are drawn from a different disjoint set of objects.

### 2.1 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 for quick implementation tasks. 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 2 presents the seed program in our example. The program maintains three 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, and `names`, which maps student ids to corresponding names. The main loop reads and implements a command for each line of input. The seed program is quite simple — there is essentially no input validation (for example, the program will accept any string as a student name, student id, or class id), 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. And 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 the much more complex python packages for building graphical user interfaces or interacting with cloud storage systems.

```
import sys

classes = {}
ids = {}
names = {}

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

  if cmd == "enroll":
    name, id = list[1:3]
    ids[name] = id
    names[id] = name

  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 2.** Python seed program for a class registration system

### 2.2 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 one potential program inference algorithm exploits this ability (as well as the structure present in the provided interface to the seed program) to quickly learn the core functionality. The inference algorithm is designed to work with programs that have the following sets of properties:

- **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 retrieve operations (which return values) from store/remove operations (which return nothing).

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. In the example, 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.

After the pairs of store/retrieve operations are determined, the inference algorithm next uses the store/retrieve pairs 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.

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.

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.

In any of these steps, the inference algorithm tolerates noise in the seed program when it crashes due to unchecked errors. For example, a 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 next iteration of the closest enclosing loop. Note that this design makes our algorithm robust against certain kinds of noise while still learning the core functionality.

### 2.3 Regeneration for C and Redis

Redis [3] is an in-memory data structure store that supports a variety of simple data structures including lists, sets, and a key/value map over strings. It runs as a server and offers a range of features including persistence and replication. Redis application programming interfaces (APIs) have been developed for many languages including C. We present an example that illustrates the regeneration of a text-based C/Redis implementation of the registration program. Figure 3 presents a session from this implementation (we replace repeated text command prompts with ... after the first prompt).

#### 2.3.1 Command Loop

Regenerated based on the specified command interface, the main command loop first reads in the command and parameters, then invokes a regenerated procedure that implements the command (see Figure 10). The command loop first presents a prompt to the user. This prompt describes the commands and parameters. It then reads in the command

```
Select a command:
    1: enroll <name> <id>
    2: add <id> <class>
    3: drop <id> <class>
    4: classes <id>
    5: name <id>
    6: id <name>
> 1 Joe 100
enroll Joe 100 successful
...
> id Joe
id Joe -> 100
...
> name 100
name 100 -> Joe
...
```

```
> add 100 CS20
add 100 CS20 successful
...
> a 100 MA30
add 100 MA30 successful
...
> 4 100
classes 100 -> CS20:MA30
...
> drop 100 CS20
drop 100 CS20 successful
...
> classes 100
classes 100 -> MA30
```

**Figure 3.** Session from regenerated Redis/C implementation

and parameters and invokes a regenerated procedure that implements each command.

The regenerated command loop enables the user to specify the command either with a number or with a prefix of the command name (see Figure 11). It handles input errors by invoking the `resync` procedure (see Figure 13), which recovers by flushing the remainder of the current input line to leave the program ready to resume at the start of the next line (this recovery strategy is an instance of the filtered iterator concept [42]).

The regenerator makes the policy decision that it will support input commands and parameters that are at most `4096` characters long (other decisions are of course possible). The `input` procedure (see Figure 13), which reads in the token, contains the appropriate checks required to ensure that malicious or corner case inputs with large tokens do not overflow the token buffer. It also contains the input validation checks required to ensure that the tokens contain only alphanumeric characters (another policy decision taken in the regenerator). It is straightforward to replace these policy decisions with others — for example, to support tokens of arbitrary length or that contain a broader or narrower range of characters.

The automatically generated prompt and error handling code is one example of augmented regeneration — the regenerator enhances the usability and robustness of the regenerated application by systematically inserting appropriate error-handling code into the regenerated implementation.

### 2.3.2 Redis Initialization Code

Before reading the first command, the `main` program invokes the `initRedis` procedure (Figure 14), which uses the hiredis C client for Redis [2] (the code in this procedure is adapted from example C code in the hiredis-0.13.3 distribution). This procedure contains an obscure boilerplate code sequence that enables the program to successfully connect to the Redis server. With current development practices, developers typically use Google or search through examples provided with the packages they are using to find such code sequences. In our approach, the knowledge of how to successfully use Redis, including the specialized knowledge of obscure boilerplate code sequences, is encapsulated inside the regenerator for automated oblivious reuse by system users.

### 2.3.3 Map Implementation

By interacting with the seed program, the program inference engine infers several (virtual) maps. In our example these maps include:

- **id → name:** The operation sequence `enroll name id; name id` reveals the presence of this map — the `name id` operation returns the `name` from the `enroll` operation.
- **name → id:** The operation sequence `enroll name id; id name` reveals the presence of this map — the `id name` operation returns the `id` from the `enroll` operation.
- **id → list(class):** The operation sequence `add id class; classes id` reveals the presence of this map — the `classes id` operation returns the `class` from the `add id class` operation. The operation sequence `add id class1; add id class2; classes id` reveals that the `add id class` operation accumulates the classes into a list — the `classes id` operation returns both `class1` and `class2` from the previous `add` operations.

The C/Redis regenerator generates code that represents these maps in Redis.

It is the responsibility of the C/Redis regenerator to generate code that uses the Redis data structures to implement these inferred maps. In our example the regenerator will use the Redis hash, which maps keys to named fields, for this purpose. The Redis hash supports two relevant commands, the `HSET key field value` command, which maps the `key,field` pair to the `value`, and the `HGET key field` command, which returns the previously stored `value`. This data structure is designed to support an object-based perspective in which each key represents an object and each field represents an attribute of the object. The C/Redis regenerator encapsulates the perspective shift required to translate from the abstract maps which the developer uses to conceptualize the computation to the object-based hash that Redis provides. With this perspective shift, each name and id corresponds to an object, with name objects implementing an id field and id objects implementing name and class fields.

The C/Redis regenerator implements all of the tables in the Redis hash, generating key prefixes to ensure that keys from different inferred maps are distinct in the hash. For example, it implements a store of the relation `Joe→100` into the inferred `name→id` map with the Redis `HSET name:Joe id 100` command. Conceptually, this Redis command captures the information that Joe is a name object whose id field has the value 100. Similarly, it implements a store of the relation `100→Joe` into the inferred `id→name` map with the Redis `HSET id:100 name Joe` command. This Redis

command captures the information that 100 is an id object whose name field has the value Joe. This implementation strategy identifies name objects in the hash by prepending the `name:` prefix to the name and id objects by prepending the `id:` prefix to the id.

Figure 15 presents the code for the enroll, name, and id operations. The code for these operations uses the string-based hiredis [2] interface to invoke Redis commands from C. The code contains the standard hiredis boilerplate required to use these commands from C, including code to check for error cases when invoking the Redis command and code to deallocate the Redis reply objects. The code for the enroll command uses the Redis `HSET key field value` command to store the name and id in the hash. The code for the name and id operations uses the Redis `HGET key field` command to retrieve the stored values from the hash. While this code does not contain any additional validation or security checks on the user-provided name or id strings, it is straightforward to add any desired checks to this code. These checks would appear before the calls to the `redisCommand` procedure, which invokes the operations against the Redis database.

#### 2.3.4 Lists and Maps

The `add id class` operation accumulates the stored classes into a list. Redis hashes do not directly support lists — another conceptual mismatch that the C/Redis regenerator must navigate. The regenerator therefore encodes the list of classes into a single string, with the items in the list separated by a delimiter (our example uses the : character as the delimiter). To maintain this list, the regenerator also generates procedures that operate on the sets encoded as strings of items separated by delimiters. In this way the regenerator bridges a mismatch between the concepts present in the seed program and the implementation mechanisms that the target implementation context provides.

The inference algorithm determines that `add id class` operations accumulate the inserted `classes` into a list and that `drop id class` operations remove the `class` from the list (Sections 3.2, 3.3, and 3.4). Figure 16 presents the code for these operations (and the `classes` operation, which prints the list of classes). This code manipulates the list using the `lookup`, `insert`, and `delete` procedures, which manipulate lists implemented as strings of items separated by the delimiter. Figures 17 and 18 present the code for these list manipulation procedures. These procedures are low-level C code that operate on strings of characters separated by the : delimiter. Here the regenerator chose to implement lists as null-terminated character strings of at most 4096*10 characters. Other implementation choices (such as unbounded length character strings) are also possible. We note that this kind of low-level C code is notoriously difficult to get correct and is often the source of security vulnerabilities [33, 39].

### 2.4 A Web Application with Browser Interface

We next present an example that illustrates the regeneration of a web-form-based HTML/CSS/Go/JavaScript/Redis implementation of the registration program. This implementation has three components: 1) a regenerated HTML page that the user's browser loads, 2) a regenerated server, written in Go, that processes commands received from the user's browser, and 3) the Redis database that the Go server uses to store and retrieve data.

The HTML web page presents input forms implemented in JavaScript. The web page receives commands and arguments from the input forms, sends requests to the web server, and displays the server response. The web server contains the main functionality for the registration program. It is implemented in Go [1] and stores data with Redis [3].

All of these software components (HTML, CSS, JavaScript, Go, and Redis) offer both significant functionality and complex interfaces. Using these interfaces requires little to no fundamental computer science knowledge. Instead, they require extensive knowledge of how to precisely configure and invoke specific interfaces to obtain the desired effect. The standard way to develop code that uses these interfaces is to extensively Google key words and phrases that lead to code snippets that can be copied, then adapted, to implement the desired functionality. Our approach encapsulates this knowledge in reusable form in the regenerator. The developer can therefore be productively ignorant of the complex interfaces and obscure usage requirements that these software components present.

In our example, after the web server starts running, the user may access the registration program from the same machine in a web browser at http://localhost:8088/static/register.html. Figure 4 presents a screen shot of the web interface. The interface is laid out as a matrix, with the rows corresponding to commands and the columns corresponding to the command parameters. The last column contains submit buttons for the corresponding commands. The box at the bottom presents the output for each executed command.

#### 2.4.1 Browser Interface Layout

The regenerated web page contains HTML code that specifies the web page layout. The boilerplate code in the regenerated HTML header (Figure 19) specifies that the HTML body uses the jQuery JavaScript library and Cascading Style Sheets (CSS). It also uses the Cloudflare skeleton CSS file [5] to support the matrix layout. The regenerated HTML body (Figure 20) specifies the input form. Each input command has a designated `<div>` element that occupies a row on the web page, with the following contents: a text label that describes the command name, an input field for each argument, and a "submit" button for sending the command and arguments to the web server. The bottom of the HTML body contains a `<div>` element for displaying the server response.

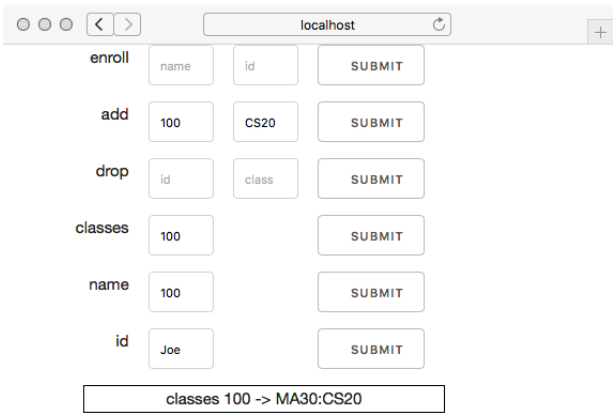


**Figure 4.** Web interface

#### 2.4.2 Communicating with Web Server

The regenerated web page also contains JavaScript code that listens to the input forms on the web page and communicates with the web server (Figures 21 and 22). This code binds a function to each of the "submit" buttons on the web page. The bound listener function reads arguments from the input forms, encodes the appropriate command and arguments as a URL, sends the URL to the web server as a request, and displays the server response at the bottom of the web page. The server response may contain values successfully returned from the database or error messages. The listener function executes each time the user clicks the corresponding "submit" button.

#### 2.4.3 Redis Initialization and Command Handlers

When the Go server starts running, it initializes its connection to the Redis database before accepting requests. The go server first invokes the `init` procedure, which uses the redis.v5 Go client for Redis [4]. Figure 23 presents the code for the `init` procedure (as is standard practice when developing applications that use these kinds of complex interfaces, this code is adapted from an example found on the Internet via a Google search). This procedure contains boilerplate code that enables the Go web server to successfully connect to the Redis database server as a client.

After initializing Redis, the program invokes the `main` procedure, which registers appropriate command handlers for incoming requests. Figure 24 presents the Go code for the `main` procedure (once again, this code is adapted from an example found on the Internet via a Google search). This procedure contains boilerplate code that enables the web server to successfully parse incoming URL requests by invoking the appropriate command handlers and parsing the appropriate arguments.

As in Section 2.3.2, our approach encapsulates the specialized knowledge of how to successfully use Redis and of how to handle URL requests all inside the regenerator, for automated oblivious reuse by system users.

#### 2.4.4 Map Implementation

The Go server implements the same strategy for representing data in the Redis database as the text interface implementations. The implementations can therefore interoperate — it is possible to access the same data in the same database with either the text or the web implementation.

The code that implements enroll, name, and id operations (Figure 25) presents the code the implements the enroll, name, uses the go-redis interface [4] to invoke Redis commands from Go. The code uses the `client` variable, initialized to hold a Redis client by the initialization code in Figure 23, to access the Redis database. Like the text interface implementations, the Go server uses the Redis `HSet` and `HGet` commands to store strings in the Redis hash. The code contains standard boilerplate required to use these commands from Go, including code to check for error cases in the invocation of the Redis command. Once again, this code was adapted from examples found on the Internet.

#### 2.4.5 Lists and Maps

As in Section 2.3.4, the regenerator encodes the list of classes into a single string and must generate code that operates these strings. Here the regenerated code uses the built-in Go string operations to construct the strings that implement the list of classes (the text interface implementations, in contrast, contain regenerated C code that implements the required string manipulations). Figure 26 presents the code for the add, drop, and classes operations. This code uses (once again), the Redis `HGet` and `HSet` operations and the Go string operations.

### 2.5 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. Installation and configuration of the multiple subsystems that the code works with can comprise a substantial obstacle to obtaining a working system. In addition to encapsulating the knowledge of how to develop code that works with these subsystems, the regenerator can also encapsulate knowledge of how to install and configure the relevant subsystems. This knowledge can take the form of either a natural language narrative of how to install and configure the subsystems (including installation and configuration options) or of scripts that directly implement the installation and configuration.

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 documentation or comments that support the

**Inputs:**
$SP$–Seed Program
$S = \{\mathtt{sop}_1\ \mathtt{p}_1^1 \ldots \mathtt{p}_{k_1}^1, \ldots, \mathtt{sop}_n\ \mathtt{p}_1^n \ldots \mathtt{p}_{k_n}^n\}$
$R = \{\mathtt{rop}_1\ \mathtt{p}_1 \rightarrow \mathtt{q}_1, \ldots, \mathtt{rop}_m\ \mathtt{p}_m \rightarrow \mathtt{q}_m\}$

**Output:**
$SRP = \{\langle \mathtt{sop}_1, \mathtt{rop}_1, k_1, i_1, j_1\rangle, \ldots, \langle \mathtt{sop}_l, \mathtt{rop}_l, k_l, i_l, j_l\rangle\}$

**Algorithm:**
$SRP = \emptyset$
**for** $\mathtt{sop}\ \mathtt{p}_1 \ldots \mathtt{p}_k \in S$
  **for** $\mathtt{rop}\ \mathtt{p} \rightarrow \mathtt{q} \in R$
    **choose distinct** $v_1, \ldots, v_k$
    **for** $1 \le i \le k$
      $v = \mathtt{sop}\ v_1 \ldots v_k; \mathtt{rop}\ v_i \mid SP$
      **for** $1 \le j \le k$
        **if** $v = v_j$ **or** $v = [v_j]$
          $SRP = SRP \cup \{\langle \mathtt{sop}, \mathtt{rop}, k, i, j\rangle\}$

**Figure 5.** Store/Retrieve Pair (*SRP*) inference algorithm

**Inputs:**
$SP$–Seed Program
$S = \{\mathtt{sop}_1\ \mathtt{p}_1^1 \ldots \mathtt{p}_{k_1}^1, \ldots, \mathtt{sop}_n\ \mathtt{p}_1^n \ldots \mathtt{p}_{k_n}^n\}$
$R = \{\mathtt{rop}_1\ \mathtt{p}_1 \rightarrow \mathtt{q}_1, \ldots, \mathtt{rop}_m\ \mathtt{p}_m \rightarrow \mathtt{q}_m\}$
$SRP = \{\langle \mathtt{sop}_1, \mathtt{rop}_1, k_1, i_1, j_1\rangle, \ldots, \langle \mathtt{sop}_l, \mathtt{rop}_l, k_l, i_l, j_l\rangle\}$

**Output:**
$KRL = \{\langle \mathtt{sop}_1, k_1, i_1, j_1, krl_1\rangle, \ldots, \langle \mathtt{sop}_l, k_l, i_l, j_l, krl_l\rangle\}$

**Algorithm:**
$KRL = \emptyset$
**for** $\langle \mathtt{sop}, \mathtt{rop}, k, i, j\rangle \in SRP$
  **choose distinct** $v_1, \ldots, v_k, u_1, \ldots, u_k$
  **such that** $v_i = u_i$
  $v = \mathtt{sop}\ v_1 \ldots v_k; \mathtt{sop}\ u_1 \ldots u_k; \mathtt{rop}\ v_i \mid SP$
  **if** $v = v_j$ $krl = \mathtt{Keep}$
  **if** $v = u_j$ $krl = \mathtt{Replace}$
  **if** $v = [v_j, u_j]$ $krl = \mathtt{List}$
  $KRL = KRL \cup \{\langle \mathtt{sop}, k, i, j, krl\rangle\}$

**Figure 6.** Keep/Replace/List (*KRL*) inference algorithm

ability of developers to use the regenerated code for a variety of purposes, including modifying the regenerated code or using the regenerated code to develop or enhance their understanding of how to use the target subsystems.

## 3 Program Inference Algorithms

We next present program inference algorithms for programs whose operations store, remove, and delete relations from maps.

### 3.1 Store/Retrieve Pair Inference Algorithm

Figure 5 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 $\mathtt{sop}\ \mathtt{p}_1 \ldots \mathtt{p}_k$ (here sop is the name of the operation and $\mathtt{p}_1 \ldots \mathtt{p}_k$ are the names of the $k$ parameters), and a set of potential retrieve operations *R*, with each operation of the form $\mathtt{rop}\ \mathtt{p} \rightarrow \mathtt{q}$ (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 $\langle \mathtt{sop}, \mathtt{rop}, k, i, j\rangle$. 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 $i$'th parameter (the key) of a previously executed sop operation, returns the $j$'th parameter (the stored value) of that operation.

The algorithm itself simply 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 $\mathtt{sop}\ v_1 \ldots v_k$, then on a potential retrieve operation $\mathtt{rop}\ v_i$, and collects the resulting value $v$ that the potential retrieve operation returns. We use the notation $v = \mathtt{sop}\ v_1 \ldots v_k;\ \mathtt{rop}\ v_i \mid SP$ 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 6 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 interence algorithm (Figure 5). It produces as output a set of tuples $\langle \mathtt{sop}, k, i, j, krl\rangle$, where sop is an operation with $k$ parameters that stores a relation into some map. The $i$'th parameter is the key

**Inputs:**
$SP$–Seed Program
$S = \{\mathsf{sop}_1\ \mathsf{p}_1^1 \ldots \mathsf{p}_{k_1}^1, \ldots, \mathsf{sop}_n\ \mathsf{p}_1^n \ldots \mathsf{p}_{k_n}^n\}$
$R = \{\mathsf{rop}_1\ \mathsf{p}_1 \rightarrow \mathsf{q}_1, \ldots, \mathsf{rop}_m\ \mathsf{p}_m \rightarrow \mathsf{q}_m\}$
$SRP = \{\langle \mathsf{sop}_1, \mathsf{rop}_1, k_1, i_1, j_1\rangle, \ldots, \langle \mathsf{sop}_l, \mathsf{rop}_l, k_l, i_l, j_l\rangle\}$

**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** $v =$ **Nil**
        $SCP = SCP \cup \{\langle \mathsf{sop}, k, i, j, \mathsf{sop}', k', i'\rangle\}$

**Figure 7.** Store/Clear Pair (*SCP*) inference algorithm

and the $j$'th parameter is the value of this stored relation. $krl \in \{\mathtt{Keep}, \mathtt{Replace}, \mathtt{List}\}$ specifies whether the operation keeps the original relation, replaces the original relation, or accumulates the stores into a list of values.

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 7 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 interence algorithm (Figure 5).

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). $\mathsf{sop}'$ is an operation with $k'$ parameters that clears the stored relation from the map. To clear the relation, the keys, specifically the $i'$'th parameter of $\mathsf{sop}'$ and $i$'th parameter of sop, must have the same value.

**Inputs:**
$SP$–Seed Program
$S = \{\mathsf{sop}_1\ \mathsf{p}_1^1 \ldots \mathsf{p}_{k_1}^1, \ldots, \mathsf{sop}_n\ \mathsf{p}_1^n \ldots \mathsf{p}_{k_n}^n\}$
$R = \{\mathsf{rop}_1\ \mathsf{p}_1 \rightarrow \mathsf{q}_1, \ldots, \mathsf{rop}_m\ \mathsf{p}_m \rightarrow \mathsf{q}_m\}$
$SRP = \{\langle \mathsf{sop}_1, \mathsf{rop}_1, k_1, i_1, j_1\rangle, \ldots, \langle \mathsf{sop}_l, \mathsf{rop}_l, k_l, i_l, j_l\rangle\}$
$SCP = \{\langle \mathsf{sop}_1, k_1, i_1, j_1, \mathsf{sop}'_1, k'_1, i'_1\rangle, \ldots,$
$\langle \mathsf{sop}_o, k_o, i_o, j_o, \mathsf{sop}'_o, k'_o, i'_o\rangle\}$

**Output:**
$SDP = \{\langle \mathsf{sop}_1, k_1, i_1, j_1, \mathsf{sop}'_1, k'_1, i'_1, j'_1\rangle, \ldots,$
$\langle \mathsf{sop}_o, k_o, i_o, j_o, \mathsf{sop}'_o, k'_o, i'_o, j'_o\rangle\}$

**Algorithm:**
$SDP = \emptyset$
**for** $\langle \mathsf{sop}, \mathsf{rop}, k, i, j\rangle \in SRP$
  **choose distinct** $v_1, \ldots, v_k$
  **for** $\langle \mathsf{sop}'\ p_1\ \ldots\ p_{k'}\rangle \in S$
    **for** $1 \leq i' \leq k',\ 1 \leq j' \leq k',\ i' \neq j'$
      **choose distinct** $u_1, \ldots, u_{k'}$
      **such that** $v_i = u_{i'}, v_j = u_{j'}$
      **if** $\langle \mathsf{sop}, k, i, j, \mathsf{sop}', k', i'\rangle \notin SCP$
        $v = \mathsf{sop}\ v_1 \ldots v_k; \mathsf{sop}'\ u_1 \ldots u_{k'}; \mathsf{rop}\ v_i \mid SP$
        **if** $v =$ **Nil or** $v = [\ ]$
          $SDP = SDP \cup \{\langle \mathsf{sop}, k, i, j, \mathsf{sop}', k', i', j'\rangle\}$

**Figure 8.** Store/Delete Pair (*SDP*) inference algorithm

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 $\mathsf{sop}\ v_1\ \ldots\ v_k$ (the operation that stores the relation) and $\mathsf{sop}'\ u_1\ \ldots\ u_{k'}$ (the potential clear operation) in sequence. The algorithm then executes the retrieve operation $\mathsf{rop}\ v_i$ 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 $v_1, \ldots, v_k$ and $u_1, \ldots, u_{k'}$ 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 7) 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 8 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 5), and the store/clear pairs *SCP* from the store/clear pair inference algorithm (Figure 7).

The algorithm produces as output a set of store/delete pairs *SDP*, with each pair of the form $\langle \mathsf{sop}, k, i, j, \mathsf{sop}', k', i', j' \rangle$. 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 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 5) and *S* to enumerate the possible store/delete pairs. It first runs sop $v_1 \ldots v_k$, which inserts the relation, then sop′ $u_1 \ldots u_{k'}$, 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 5) 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 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

**Inputs:**
*SP*–Seed Program
$S = \{\mathsf{sop}_1\ \mathsf{p}_1^1 \ldots \mathsf{p}_{k_1}^1, \ldots, \mathsf{sop}_n\ \mathsf{p}_1^n \ldots \mathsf{p}_{k_n}^n\}$
$R = \{\mathsf{rop}_1\ \mathsf{p}_1 \rightarrow \mathsf{q}_1, \ldots, \mathsf{rop}_m\ \mathsf{p}_m \rightarrow \mathsf{q}_m\}$
$SRP = \{\langle \mathsf{sop}_1, \mathsf{rop}_1, k_1, i_1, j_1 \rangle, \ldots, \langle \mathsf{sop}_I, \mathsf{rop}_I, k_I, i_I, j_I \rangle\}$

**Output:**
$M = \{\{\langle \mathsf{sop}_1^1, k_1^1, i_1^1, j_1^1 \rangle, \ldots, \langle \mathsf{sop}_{m_1}^1, k_{m_1}^1, i_{m_1}^1, j_{m_1}^1 \rangle\},$
$\ldots,$
$\{\langle \mathsf{sop}_1^n, k_1^n, i_1^n, j_1^n \rangle, \ldots, \langle \mathsf{sop}_{m_n}^n, k_{m_n}^n, i_{m_n}^n, j_{m_n}^n \rangle\}\}$

**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** $v_{i_1} = u_{i_2},\ v_{j_1} = u_{j_2}$
    $v = \mathsf{sop}_1\ v_1\ \ldots\ v_{k_1}; \mathsf{rop}_1\ v_{i_1}\ \mid SP$
    $u = \mathsf{sop}_2\ u_1\ \ldots\ u_{k_2}; \mathsf{rop}_1\ v_{i_1}\ \mid SP$
    **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 9.** Map ($M$) inference algorithm

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 9 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 5). It represents each map as a set of tuples $\{\langle \mathsf{sop}_1, k_1, i_1, j_i \rangle, \ldots, \langle \mathsf{sop}_m, k_m, i_m, j_m \rangle\}$. 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 $\mathbf{Union}(M, \langle \mathsf{sop}_1, i_1, j_1 \rangle, \langle \mathsf{sop}_2, i_2, j_2 \rangle)$, 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.

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

### 3.7 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 7 and 8).

We note that because of incomplete input validation or corner case error handling, the inference algorithms may trigger crashes in the seed program. For example, the store/retrieve, store/clear, and store/delete pair inference algorithms will trigger KeyErrors and ValueErrors in the student registration program from Section 2 when it executes operations that attempt to retrieve or remove nonexistent relations. These crashes are caused by the incomplete input validation and error handling of the seed program. The inference algorithms view this kind of behavior as noise and, when learning the seed program's core functionality, do not incorporate results from inputs that trigger such noisy behaviors.

## 4 Related Work

***Software modernization.*** Software modernization tools [14, 20, 40] analyze the source code of a legacy program, translate the program into a high-level modeling language, then use this representation to generate a new program that implements the same functionality in a more modern language. The translation strictly follows syntactic cues and usually requires human intervention. Our approach, in contrast, (a) works with the given implementation as a black box, without analyzing code and (b) 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 [34] uses dynamic instrumentation to extract the functionality of computational stencil kernels embedded within production binaries. It then replaces the stencil kernel with a computation expressed in the Halide [38] domain-specific language. The goal is to replace the legacy implementation with a version optimized for modern computational platforms. Program fracture and recombination [9] 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, (a) models the full computation and regenerates the entire application, augmented as appropriate, (b) can work with incomplete

or buggy implementations of the original program, and (c) targets programs that store and retrieve data in maps.

***Stateless model extraction.*** Model extraction algorithms use queries to construct representations for given programs, where the representations are stateless functions such as decision trees [17, 50] or symbolic rules [49]. Model compression algorithms [12, 28] use machine learning models, such as neural networks, to mimic a given machine learning model, typically by generating inputs (training data) and observing the outputs from the given model. Our approach, in contrast, (a) infers stateful models that can store and retrieve data across multiple queries and (b) regenerates a new program or programs, augmented as appropriate, that implement the core functionality without defects or security vulnerabilities on new implementation platforms.

***Partial model learning.*** Algorithms for learning black-box state machines [8, 10, 15, 16, 19, 25, 29, 35, 37, 51, 52] construct partial representations of program functionality, using finite automata with states and transition rules. State fuzzing tools [7, 18] are used to hypothesize state machines for given program implementations, which can aid developers in discovering bugs such as spurious state transitions. Network function state model extraction [53] performs program slicing on given code and models the sliced partial programs as packet-processing automata. These algorithms extract partial models of the given programs. Our approach, in contrast, (a) extracts a complete representation of the core functionality, which, in turn, enables the regeneration (and replacement) of the initial program, (b) can work with programs with defects or that only partially implement the core functionality, (c) regenerates a new program or programs, augmented as appropriate, that implement the core functionality without defects or security vulnerabilities on new implementation platforms, and (d) 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 algorithms typically generate programs by solving constraints specified as logical formulas [27, 31], input/output examples [11, 24, 26, 30, 36, 44], or templates [46–48]. These techniques do not work with existing implementations, but require (partial or complete) specifications in the form of logical formulas or input/output examples. Our approach, in contrast, (a) works with a concrete program implementation rather than abstract specifications, which can be especially useful for regenerating new implementations for legacy software, (b) actively and automatically executes the given implementation as needed to infer the core functionality, without requiring user interaction and is not restricted to a limited set of training data, (c) can work with programs with defects or that only partially implement the core functionality, (d) automatically regenerates a new program or programs, augmented as appropriate, that implement the core functionality without defects or security vulnerabilities on new implementation platforms.

Counterexample-guided inductive synthesis (CEGIS) [46] uses finite programs (whose input is bounded and terminate on all inputs after a bounded number of operations) as specifications, to generate more efficient implementations that always produce the correct outputs. Oracle-guided program synthesis [30] uses hypothetical I/O oracles (which always return the correct results) to guide the synthesis of loop-free programs. Our approach, in contrast, (a) works with stateful programs that store and retrieve unbounded amounts of data (as opposed to finite programs that implement functions), (b) can work with programs with defects or that only partially implement the core functionality, and (c) regenerates new programs, augmented with additional security checks, that use complex programming interfaces to execute on modern distributed computing platforms.

***Concolic Testing.*** Concolic testing [13, 22, 23, 41] generates inputs that systematically explore all execution paths in the program. The goal is to find inputs that expose software defects. BuzzFuzz [21] generates inputs that target defects that occur because of coding oversights at the boundary between application and library code. DIODE [43] 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, (a) works with the given implementation as a black box, without analyzing code, (b) extracts a representation of the core functionality, and (c) 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 (often largely mindlessly) 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.

## References


[1] The Go Programming Language. https://golang.org/. (????).

[2] Hiredis. https://redislabs.com/lp/hiredis/. (????).

[3] Redis. https://redis.io/. (????).

[4] Redis.v5. https://gopkg.in/redis.v5. (????).

[5] Skeleton: Responsive CSS Boilerplate. http://getskeleton.com/. (????).

[6] Stack Overflow. http://stackoverflow.com/. (????).

[7] 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. DOI: http://dx.doi.org/10.1109/ICSTW.2013.60

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

[9] 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. DOI: http://dx.doi.org/10.1109/HPEC.2015.7396314

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

[11] Angela Bonifati, Radu Ciucanu, and Slawek Staworko. 2014. Interactive Join Query Inference with JIM. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1541–1544. DOI: http://dx.doi.org/10.14778/2733004.2733025

[12] 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. DOI: http://dx.doi.org/10.1145/1150402.1150464

[13] 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. DOI: http://dx.doi.org/10.1145/1180405.1180445

[14] 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. DOI: http://dx.doi.org/10.1007/s10270-012-0270-z

[15] 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. DOI: http://dx.doi.org/10.1007/s00165-016-0355-5

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

[17] 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.

[18] 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.

[19] 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. DOI: http://dx.doi.org/10.1007/978-3-319-41540-6_25

[20] 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. DOI: http://dx.doi.org/10.1016/j.scico.2011.04.003

[21] 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. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070546

[22] 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. DOI: http://dx.doi.org/10.1145/1065010.1065036

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

[24] Patrice Godefroid and Ankur Taly. 2012. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 441–452. DOI: http://dx.doi.org/10.1145/2254064.2254116

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

[26] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. DOI: http://dx.doi.org/10.1145/1926385.1926423

[27] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 62–73. DOI: http://dx.doi.org/10.1145/1993498.1993506

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

[29] 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. DOI: http://dx.doi.org/10.1007/978-3-319-11164-3_26

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

[31] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 304–314. DOI: http://dx.doi.org/10.1145/512529.512566

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

[33] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic Runtime Error Repair and Containment via Recovery Shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14')*. ACM, New York, NY, USA, 227–238. DOI: http://dx.doi.org/10.1145/2594291.2594337

[34] 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. DOI: http://dx.doi.org/10.1145/2737924.2737974

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

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

[37] 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. DOI: http://dx.doi.org/10.1145/1081180.1081189

[38] 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. DOI: http://dx.doi.org/10.1145/2491956.2462176

[39] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*. 303–316.

[40] 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. DOI: http://dx.doi.org/10.1007/s10515-013-0130-2

[41] 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. DOI: http://dx.doi.org/10.1145/1081706.1081750

[42] Jiasi Shen and Martin Rinard. 2015. Filtered Iterators for Safe and Robust Programs in RIFL. http://hdl.handle.net/1721.1/100542. (2015). http://hdl.handle.net/1721.1/100542 MIT-CSAIL-TR-2015-036.

[43] 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. DOI: http://dx.doi.org/10.1145/2694344.2694389

[44] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 289–299. DOI: http://dx.doi.org/10.1145/2025113.2025153

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

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

[47] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2013. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer* 15, 5 (2013), 497–518. DOI: http://dx.doi.org/10.1007/s10009-012-0223-4

[48] Ankur Taly, Sumit Gulwani, and Ashish Tiwari. 2011. Synthesizing switching logic using constraint solving. *International Journal on Software Tools for Technology Transfer* 13, 6 (2011), 519–535. DOI: http://dx.doi.org/10.1007/s10009-010-0172-8

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

[50] 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.

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

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

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

[54] 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).

# A Appendix

```
int main(int argc, char **argv) {
  redisContext *c;
  token name, id, class;
  int len;

  c = initRedis(argc, argv);

  while (1) {
    int cmd = prompt();
    if (cmd < 1) { break; }
    switch (cmd) {
      case 1:
        len = input(name);
        if (len < 1) { resync(); continue; }
        len = input(id);
        if (len < 1) { resync(); continue; }
        doenroll(c, name, id);
        break;

      case 2:
        len = input(id);
        if (len < 1) { resync(); continue; }
        len = input(class);
        if (len < 1) { resync(); continue; }
        doadd(c, id, class);
        break;

      case 3:
        len = input(id);
        if (len < 1) { resync(); continue; }
        len = input(class);
        if (len < 1) { resync(); continue; }
        dodrop(c, id, class);
        break;

      case 4:
        len = input(id);
        if (len < 1) { resync(); continue; }
        doclasses(c, id);
        break;

      case 5:
        len = input(id);
        if (len < 1) { resync(); continue; }
        doname(c, id);
        break;

      case 6:
        len = input(name);
        if (len < 1) { resync(); continue; }
        doid(c, name);
        break;

      default:
        resync();
        continue;
    }
  }
}
```

**Figure 10.** Command Loop for C/Redis Implementation

```
int prompt() {
  token t;
  printf("Select a command:\n\
    \t1: enroll <name> <id>\n\
    \t2: add <id> <class>\n\
    \t3: drop <id> <class>\n\
    \t4: classes <id>\n\
    \t5: name <id>\n\
    \t6: id <name>\n> ");
  int len = input(t);
  if (len <= 0) return -1;
  if (strcmp(t, "1") == 0 || prefix(t, "enroll")) {
    return 1;
  }
  if (strcmp(t, "2") == 0 || prefix(t, "add")) {
    return 2;
  }
  if (strcmp(t, "3") == 0 || prefix(t, "drop")) {
    return 3;
  }
  if (strcmp(t, "4") == 0 || prefix(t, "classes")) {
    return 4;
  }
  if (strcmp(t, "5") == 0 || prefix(t, "name")) {
    return 5;
  }
  if (strcmp(t, "6") == 0 || prefix(t, "id")) {
    return 6;
  }
  return -1;
}
```

**Figure 11.** Prompt Code for Text Interface

```
typedef char token[4096];

void resync() {
  int c;
  c = getc(stdin);
  while ((c != '\n') && (c != EOF)) {
    c = getc(stdin);
  }
}
```

**Figure 12.** Error Recovery Code for C/Redis Implementation

```
typedef char token[4096];

int input(token str) {
  int c;
  int n = 0;

  while (1) {
    if (((int) sizeof(token) - 1) == n) {
      str[n] = '\0';
      return -n;
    }
    c = getc(stdin);
    if (isspace(c)) {
      if (0 < n) {
        str[n++] = '\0';
        return n;
      }
    } else if (c == EOF) {
      str[n] = '\0';
      return n;
    } else {
      if (isalnum(c)) {
        str[n++] = c;
      } else {
        str[n] = '\0';
        return -n;
      }
    }
  }
}
```

**Figure 13.** Text Input Code for C/Redis Implementation

```
redisContext *initRedis(int argc, char **argv) {
    redisContext *c;
    const char *hostname = (argc > 1) ? argv[1] : "
         127.0.0.1";
    int port = (argc > 2) ? atoi(argv[2]) : 6379;

    struct timeval timeout = { 1, 500000 }; // 1.5
         seconds
    c = redisConnectWithTimeout(hostname, port, timeout);
    if (c == NULL || c->err) {
        if (c) {
            printf("Connection error: %s\n", c->errstr);
            redisFree(c);
        } else {
            printf("Connection error: can't allocate
                 redis context\n");
        }
        exit(1);
    }
    return c;
}
```

**Figure 14.** Redis Initialization Code for C/Redis Implementation

```
void doenroll(redisContext *c, token name, token id){
  redisReply *reply;

  reply = redisCommand(c, "HSET id:%s name %s", id, name)
       ;
  if (reply == NULL) return;
  freeReplyObject(reply);
  reply = redisCommand(c, "HSET name:%s id %s", name, id)
       ;
  if (reply == NULL) return;
  freeReplyObject(reply);
  printf("enroll %s %s successful\n", name, id);
}

void doname(redisContext *c, token id){
  redisReply *reply;

  reply = redisCommand(c, "HGET id:%s name", id);
  if (reply == NULL || reply->str == NULL) return;
  printf("name %s -> %s\n", id, reply->str);
  freeReplyObject(reply);
}

void doid(redisContext *c, token name){
  redisReply *reply;

  reply = redisCommand(c, "HGET name:%s id", name);
  if (reply == NULL || reply->str == NULL) return;
  printf("id %s -> %s\n", name, reply->str);
  freeReplyObject(reply);
}
```

**Figure 15.** Code for Enroll, Name, and Id Operations

```
void doadd(redisContext *c, token id, token class){
  redisReply *reply;
  char *s;
  list new;

  reply = redisCommand(c, "HGET id:%s class", id);
  if (reply == NULL) return;
  if (reply->str == NULL) {
    s = empty;
  } else {
    s = reply->str;
  }
  if (lookup(s, class) < 0) {
    if (0 < insert(s, class, new)) {
      freeReplyObject(reply);
      reply = redisCommand(c, "HSET id:%s class %s", id,
           new);
      if (reply == NULL) return;
    }
  }
  freeReplyObject(reply);
  printf("add %s %s successful\n", id, class);
}

void dodrop(redisContext *c, token id, token class){
  redisReply *reply;
  list new;

  reply = redisCommand(c, "HGET id:%s class", id);
  if (reply == NULL) return;
  if (reply->str != NULL) {
    delete(reply->str, class, new);
    freeReplyObject(reply);
    reply = redisCommand(c, "HSET id:%s class %s", id,
         new);
    if (reply == NULL) return;
    freeReplyObject(reply);
  } else {
    freeReplyObject(reply);
  }
  printf("drop %s %s successful\n", id, class);
}

void doclasses(redisContext *c, token id){
  redisReply *reply;

  reply = redisCommand(c, "HGET id:%s class", id);
  if (reply == NULL || reply->str == NULL) return;
  printf("classes %s -> %s\n", id, reply->str);
  freeReplyObject(reply);
}
```

**Figure 16.** Code for Add, Drop, and Classes Operations

```
typedef char list[4096*20];
list empty="\0";

int insert(list l, token t, list n) {
  int i = 0;
  while (l[i] != '\0' && i < (int) sizeof(list)-2) {
    n[i] = l[i];
    i++;
  }
  if (0 < i) {
    n[i++] = ':';
  }
  int j = 0;
  while ((i < (int) sizeof(list)-1) && (t[j] != '\0')) {
    n[i++] = t[j++];
  }
  n[i] = '\0';
  if (t[j] != '\0') return -(i+1);
  else return i;
}

int delete(list l, token t, list n) {
  int i, j, k;
  int p;

  p = 0; i = 0; k = 0;
  while (l[i] != '\0' && k < (int) sizeof(list)-1) {
    j = 0;
    p = k;
    while (l[i] != ':' && l[i] != '\0' && t[j] != '\0' &&
          l[i] == t[j]) {
      n[k] = l[i];
      i++; j++; k++;
    }
    if (t[j] == '\0') {
      if (l[i] == ':') {
        k = p;
        i++;
        continue;
      } else if (l[i] == '\0') {
        if (0 < p) {
          n[p-1] = '\0';
        }
        k = p;
        continue;
      }
    }
    while (l[i] != ':') {
      n[k] = l[i];
      if (l[i] == '\0') return i;
      i++; k++;
    }
    n[k] = ':';
    i++; k++;
  }
  n[k] = '\0';
  if (l[i] != '\0') {
    return -i;
  }
  return i;
}
```

**Figure 17.** Code for Insert and Delete Procedures

```
int lookup(list l, token t) {
  int i, j;
  int n;

  i = 0;
  n = 0;
  while (l[i] != '\0') {
    j = 0;
    while (l[i] != ':' && l[i] != '\0' && t[j] != '\0' &&
          l[i] == t[j]) {
      i++; j++;
    }
    if ((l[i] == ':' || l[i] == '\0') && t[j] == '\0') {
      return n;
    }
    while (l[i] != ':' && l[i] != '\0') {
      if (l[i] == '\0') return -1;
      i++;
    }
    i++; n++;
  }
  return -1;
}

void print(char *l) {
  int i = 0;
  while (1) {
    while (l[i] != ':') {
      if (l[i] == '\0') return;
      putchar(l[i]);
      i++;
    }
    putchar(' ');
    i++;
  }
}
```

**Figure 18.** Code for lookup and print Procedures

```
<head>
  <script
    src="https://code.jquery.com/jquery-3.2.0.min.js"
    integrity="sha256-JAW99MJVpJBGcbzEuXk4Az05s/
        XyDdBomFqNlM3ic+I="
    crossorigin="anonymous"></script>
  <script src="form.js"></script>

  <link rel="stylesheet"
    href="https://cdnjs.cloudflare.com/ajax/libs/skeleton
        /2.0.4/skeleton.min.css" />
  <style>
    #response{
      border:1px solid black;
      text-align:center;
    }
    .command{
      text-align:right;
    }
  </style>
</head>
```

**Figure 19.** Code for browser interface HTML header that refers to JavaScript files and contains CSS

```
<body class="container">
  <div class="enroll"> <div class="row">
    <div class="two columns command">enroll</div>
    <input type="text" id="name" class="two columns"
        placeholder="name">
    <input type="text" id="id" class="two columns"
        placeholder="id">
    <button id="submit" class="three columns">submit</
        button>
    </div>
  </div>

  <div class="add"> <div class="row">
    <div class="two columns command">add</div>
    <input type="text" id="id" class="two columns"
        placeholder="id">
    <input type="text" id="class" class="two columns"
        placeholder="class">
    <button id="submit" class="three columns">submit</
        button>
    </div>
  </div>

  <div class="drop"> <div class="row">
    <div class="two columns command">drop</div>
    <input type="text" id="id" class="two columns"
        placeholder="id">
    <input type="text" id="class" class="two columns"
        placeholder="class">
    <button id="submit" class="three columns">submit</
        button>
    </div>
  </div>

  <div class="classes"> <div class="row">
    <div class="two columns command">classes</div>
    <input type="text" id="id" class="two columns"
        placeholder="id">
    <div class="two columns"> </div>
    <button id="submit" class="three columns">submit</
        button>
    </div>
  </div>

  <div class="name"> <div class="row">
    <div class="two columns command">name</div>
    <input type="text" id="id" class="two columns"
        placeholder="id">
    <div class="two columns"> </div>
    <button id="submit" class="three columns">submit</
        button>
    </div>
  </div>

  <div class="id"> <div class="row">
    <div class="two columns command">id</div>
    <input type="text" id="name" class="two columns"
        placeholder="name">
    <div class="two columns"> </div>
    <button id="submit" class="three columns">submit</
        button>
    </div>
  </div>

  <div id="response" class="id row nine columns"> </
      div>
</body>
```

**Figure 20.** Code for browser interface HTML body

```
$(document).ready(function() {
    var _name_ = $("#name", $(".enroll")).val();
    var _id_ = $("#id", $(".enroll")).val();
    var url = "/enroll/" + _name_ + "/" + _id_;
    $.post({
      url: url,
      success: function (resp) {
        var str = "enroll " + _name_ + " " + _id_ + "
             successful";
        $("#response").html(str);
      },
      error: function (resp) {
        var str = "error " + resp.status + ": " + "enroll
              " + _name_ + " " + _id_ + " " + resp.
             statusText;
        $("#response").html(str);
      }
    })
  })

  $("#submit", $(".add")).click(function() {
    var _id_ = $("#id", $(".add")).val();
    var _class_ = $("#class", $(".add")).val();
    var url = "/add/" + _id_ + "/" + _class_;
    $.post({
      url: url,
      success: function (resp) {
        var str = "add " + _id_ + " " + _class_ + "
             successful";
        $("#response").html(str);
      },
      error: function (resp) {
        var str = "error " + resp.status + ": " + "add "
             + _id_ + " " + _class_ + " " + resp.
             statusText;
        $("#response").html(str);
      }
    })
  })

  $("#submit", $(".drop")).click(function() {
    var _id_ = $("#id", $(".drop")).val();
    var _class_ = $("#class", $(".drop")).val();
    var url = "/drop/" + _id_ + "/" + _class_;
    $.post({
      url: url,
      success: function (resp) {
        var str = "drop " + _id_ + " " + _class_ + "
             successful";
        $("#response").html(str);
      },
      error: function (resp) {
        var str = "error " + resp.status + ": " + "drop "
              + _id_ + " " + _class_ + " " + resp.
             statusText;
        $("#response").html(str);
      }
    })

})
```

**Figure 21.** JavaScript code for communicating commands Enroll, Id, and Name with the web server

```
$(document).ready(function() {
  $("#submit", $(".name")).click(function() {
    var _id_ = $("#id", $(".name")).val();
    var url = "/name/" + _id_;
    $.get({
      url: url,
      success: function (resp) {
        var str = "name " + _id_ + " -> " + resp;
        $("#response").html(str);
      },
      error: function (resp) {
        var str = "error " + resp.status + ": " + "name "
              + _id_ + " " + resp.statusText;
        $("#response").html(str);
      }
    })
  })

  $("#submit", $(".id")).click(function() {
    var _name_ = $("#name", $(".id")).val();
    var url = "/id/" + _name_;
    $.get({
      url: url,
      success: function (resp) {
        var str = "id " + _name_ + " -> " + resp;
        $("#response").html(str);
      },
      error: function (resp) {
        var str = "error " + resp.status + ": " + "id " +
              _name_ + " " + resp.statusText;
        $("#response").html(str);
      }
    })
  })

  $("#submit", $(".enroll")).click(function() {
  })

  $("#submit", $(".classes")).click(function() {
    var _id_ = $("#id", $(".classes")).val();
    var url = "/classes/" + _id_;
    $.get({
      url: url,
      success: function (resp) {
        var str = "classes " + _id_ + " -> " + resp;
        $("#response").html(str);
      },
      error: function (resp) {
        var str = "error " + resp.status + ": " + "
             classes " + _id_ + " " + resp.statusText;
        $("#response").html(str);
      }
    })
  })

})
```

**Figure 22.** JavaScript code for communicating commands Add, Drop, and Classes with the web server

```
package main

import (
  "fmt"
  "github.com/gorilla/mux"
  "log"
  "net/http"
  "time"
  "gopkg.in/redis.v5"
  "strings"
)

var client *redis.Client

func init() {
  client = redis.NewClient(&redis.Options{
    Addr:         ":6379",
    DialTimeout:  10 * time.Second,
    ReadTimeout:  30 * time.Second,
    WriteTimeout: 30 * time.Second,
    PoolSize:     10,
    PoolTimeout:  30 * time.Second,
  })
}
```

**Figure 23.** Redis initialization code for web server

```
func main() {
  print("=== Server ===\n")
  var dir = "."
  router := mux.NewRouter().StrictSlash(true)
  router.PathPrefix("/static/").Handler(http.StripPrefix(
       "/static/", http.FileServer(http.Dir(dir))))
  router.HandleFunc("/enroll/{name}/{id}", doenroll)
  router.HandleFunc("/add/{id}/{class}", doadd)
  router.HandleFunc("/drop/{id}/{class}", dodrop)
  router.HandleFunc("/classes/{id}", doclasses)
  router.HandleFunc("/name/{id}", doname)
  router.HandleFunc("/id/{name}", doid)
  log.Fatal(http.ListenAndServe(":8088", router))
}
```

**Figure 24.** Input handler initialization code for web server

```
func doenroll(w http.ResponseWriter, r *http.Request){
  vars := mux.Vars(r)
  id := vars["id"]
  name := vars["name"]
  var err error

  err = client.HSet("id:" + id, "name", name).Err()
  if err != nil {
    http.Error(w, err.Error(), http.
         StatusInternalServerError)
    return
  }
  err = client.HSet("name:" + name, "id", id).Err()
  if err != nil {
    http.Error(w, err.Error(), http.
         StatusInternalServerError)
    return
  }
}

func doname(w http.ResponseWriter, r *http.Request){
  vars := mux.Vars(r)
  id := vars["id"]
  var val string
  var err error

  val, err = client.HGet("id:" + id, "name").Result()
  if err != nil {
    http.Error(w, err.Error(), http.StatusNotFound)
    return
  }
  fmt.Fprintf(w, val)
}

func doid(w http.ResponseWriter, r *http.Request){
  vars := mux.Vars(r)
  name := vars["name"]
  var val string
  var err error

  val, err = client.HGet("name:" + name, "id").Result()
  if err != nil {
    http.Error(w, err.Error(), http.StatusNotFound)
    return
  }
  fmt.Fprintf(w, val)
}
```

**Figure 25.** Server code for enroll, id, and name operations

```
func doadd(w http.ResponseWriter, r *http.Request){
  vars := mux.Vars(r)
  id := vars["id"]
  class := vars["class"]
  var val string
  var err error

  val, err = client.HGet("id:" + id, "class").Result()
  if err != nil || val == "" {
    err = client.HSet("id:" + id, "class", class).Err()
    if err != nil {
      http.Error(w, err.Error(), http.
           StatusInternalServerError)
      return
    }
  } else if strings.Index(val, class) == -1 {
    err = client.HSet("id:" + id, "class", val + ":" +
         class).Err()
    if err != nil {
      http.Error(w, err.Error(), http.
           StatusInternalServerError)
      return
    }
  } else {
    http.Error(w, class, http.StatusFound)
    return
  }
}

func dodrop(w http.ResponseWriter, r *http.Request){
  vars := mux.Vars(r)
  id := vars["id"]
  class := vars["class"]
  var val string
  var err error

  val, err = client.HGet("id:" + id, "class").Result()
  if err != nil {
    http.Error(w, err.Error(), http.StatusNotFound)
    return
  }
  new := strings.Replace(val + ":", class + ":", "", -1)
  new = strings.TrimSuffix(new, ":")
  if (val == new) {
    http.Error(w, class, http.StatusNotFound)
    return
  }
  err = client.HSet("id:" + id, "class", new).Err()
  if err != nil {
    http.Error(w, err.Error(), http.
         StatusInternalServerError)
    return
  }
}

func doclasses(w http.ResponseWriter, r *http.Request){
  vars := mux.Vars(r)
  id := vars["id"]
  var val string
  var err error

  val, err = client.HGet("id:" + id, "class").Result()
  if err != nil {
    http.Error(w, err.Error(), http.StatusNotFound)
    return
  }
  fmt.Fprintf(w, val)
}
```

**Figure 26.** code for add, drop, and classes operations