M^3: Semantic API Migrations

Bruce Collie
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
bruce.collie@ed.ac.uk

Philip Ginsbach*
GitHub Software UK
Oxford, United Kingdom
ginsbach@github.com

Jackson Woodruff
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
J.C.Woodruff@sms.ed.ac.uk

Ajitha Rajan
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
arajan@inf.ed.ac.uk

Michael F.P. O'Boyle
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
mob@inf.ed.ac.uk

ABSTRACT

Library migration is a challenging problem, where most existing approaches rely on prior knowledge. This can be, for example, information derived from changelogs or statistical models of API usage.

This paper addresses a different API migration scenario where there is no prior knowledge of the target library. We have no historical changelogs and no access to its internal representation. To tackle this problem, this paper proposes a novel approach (M^3), where probabilistic program synthesis is used to *semantically* model the behavior of library functions. Then, we use an SMT-based code search engine to discover similar code in user applications. These discovered instances provide potential locations for API migrations.

We evaluate our approach against 7 well-known libraries from varied application domains, learning correct implementations for 94 functions. Our approach is integrated with standard compiler tooling, and we use this integration to evaluate migration opportunities in 9 existing C/C++ applications with over 1MLoC. We discover over 7,000 instances of these functions, of which more than 2,000 represent migration opportunities.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Software verification and validation*; Compilers.

ACM Reference Format:
Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael F.P. O'Boyle. 2020. M^3: Semantic API Migrations. In *Proceedings of the 35th International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual.* ACM, New York, NY, USA, 13 pages.

*Work performed while at the University of Edinburgh

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE'20, September 21–25, 2020, Virtual
© 2020 Association for Computing Machinery.

1 INTRODUCTION

1.1 API Migration

Libraries are a fundamental feature of software development. They allow the sharing of common code, separation of concerns and a reduction in overall development time. However, libraries are not static. They continually evolve to provide increased functionality, security and performance. Unfortunately, upgrading software to match library evolution is a significant engineering challenge for large code bases.

Given the wide-scale nature of the problem, there is much prior work in the area under various headings (e.g. library upgrade, API evolution or library migration). Work in these areas aims to answer the same question: when (and how) can a program using API X be transformed to one that uses API Y while preserving its behavior? This is a difficult problem even when X and Y have similar interfaces. It becomes more challenging if their behaviors do not match, and requires surrounding code to be factored in.

There are several approaches to this migration problem: if examples exist of previous successful migrations, then these examples can be used to derive mapping rules [51]. This approach requires that a full history of the application's source code is available, annotated with the libraries in use at each commit. Neural models have been used successfully to predict properties of programs based on learned vector-space embeddings [32]. However, these approaches require large training sets and are imprecise with respect to program semantics. A more precise (but less automatic) approach is to use expert knowledge to encode known migration patterns [43, 53].

All these prior approaches require some knowledge of the API. In this paper we tackle the challenging task of API migration *without* any prior knowledge of the source or target libraries. Here, we do not have access to the library's source code, nor to a corpus of example usages of the library. While this scenario may seem draconian, it is often the case in practice [26]. Libraries may be closed-source [50] or distributed in binaries for convenience [29], and could even be implemented as hardware [10]. In this paper we propose a novel approach which automatically learns pattern-based semantic migrations, but without up-front expert knowledge.

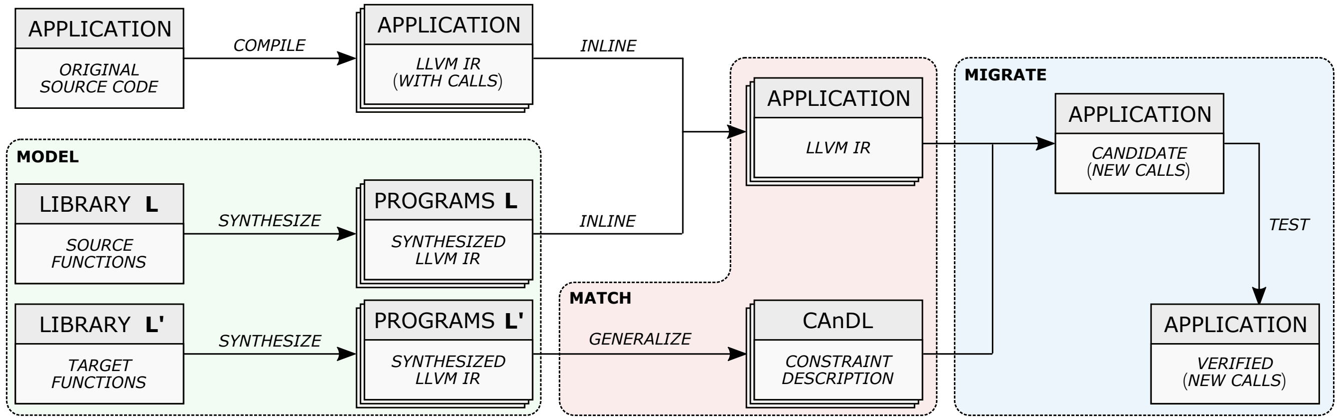

Figure 1: A summary of the M^3 workflow. Models for library functions are synthesized. Source functions are inlined while synthesized target functions are generalized into constraint descriptions, which are then used to search compiled user code for potential migrations.

1.2 M^3: Model, Match and Migrate

The key to our approach is to derive a model that is actual executable code. We call such an approach *semantics-based* migration. Given a specification for a library function (type signature, function name, library binary containing its implementation), M^3 attempts to automatically *Model* its behavior using program synthesis and checks correctness with respect to automatically generated input-output examples. It inlines the learned program models, then uses compiler-based constraint analysis to *Match* regions of application code with compatible libraries. Finally, we *Migrate* these regions by replacing application code with library calls.

A useful feature of this approach is that as well as library migration, it allows the refactoring of library-free user code to use libraries. This is because the synthesized models are themselves code, and are inlined and analyzed together with application code. Complex refactorings that integrate contextual code around an API call are enabled by this approach.

Our approach, while having the benefit of not requiring library vendors to release their source code, relies on the ability to synthesize programs in a reasonable time. We build on methods from sketch-based synthesis [48] to discover program structure before performing a directed enumerative search. We incorporate new probabilistic models to more effectively navigate the large search space. We evaluate our approach across 7 libraries, synthesizing 94 functions, and match them to over 7,000 old library calls across 10 applications with up to 1MLoC. We were able to successfully migrate more than 2,000 of these calls to another library.

Summary of Contributions. We provide a novel and efficient program synthesizer for real-world library functions. Additionally, we detail a method for matching similar code in applications using solver-aided techniques. Using this procedure, we are able to discover opportunities for usages of a source library **L** or application code **C** to be migrated to a target library **L'**. Furthermore, we can migrate source libraries with surrounding contextual code (**C** + **L**) to target libraries. This is achieved *without any knowledge* about the implementation of either library.

2 OVERVIEW

In this section, we first present a high-level summary of the M^3 workflow, then show an example of the type of migrations it enables in practice.

2.1 M^3 Workflow

Figure 1 shows the flow of data through M^3. It takes as input an application, along with specifications for source and target library APIs (currently used **L** and potential targets **L'**). The end result is a modified application that references the target libraries. We highlight the three distinct phases: **Model**, **Match** and **Migrate**.

2.1.1 Model. We assume that the source code for libraries is not available, as is often the case in practice [26]. The first phase of M^3 is Model: the synthesis of programs equivalent to functions in both the source and target libraries. The programs we synthesize are in the form of LLVM [27] intermediate representation; this representation allows us to directly integrate synthesized programs in existing compiler toolchains, and to benefit from robust program manipulation libraries. The synthesis process is specified using randomly-generated input-output examples (see Section 3.1.1).

2.1.2 Match. The second phase, Match, uses the synthesized implementations of source and target library functions in two ways. First, we inline the synthesized code of the *source* library functions into the user application at each call site. Secondly, we generalize the synthesized code of the *target* library functions to a constraint-based description that allows for matching code to be efficiently searched for.

Performing inlining means that the behavior of the library function and the *context* in which it appears are unified; migrations that require splitting, merging or moving functionality can be discovered and performed.

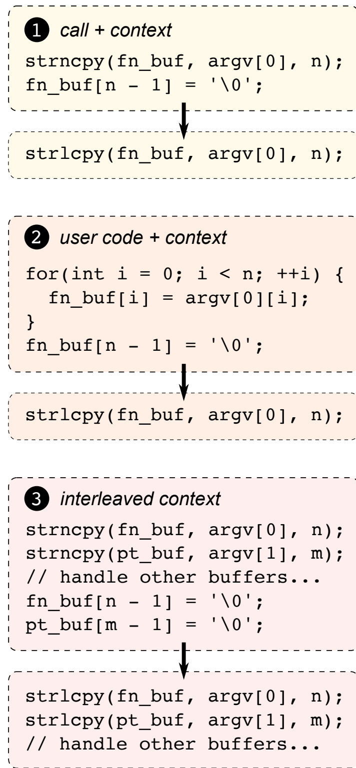

Figure 2: Example of three contexts in which M^3 is able to perform contextual API migrations using only the behavior of the source and target functions.

2.1.3 Migrate. Once matches are found, we verify whether or not potential migrations are correct. First, we perform basic integration testing using random examples on the new code. This helps to eliminate false positive matches. At this stage, the migration can be performed automatically, although in practice the user would be asked to confirm the migration (as is usual with API migration tools). We perform integration testing to check correctness of Migrate.

2.2 Example

To demonstrate the types of migration that M^3 offers, we use a running example taken from the Common Weakness Enumeration (CWE) database [1]. If the standard `strncpy` function is used to copy a C string, null-termination is not guaranteed. This can lead to buffer over-reads, and so alternative functions often exist to perform a terminated copy (for example, `strlcpy` on BSD, `StringCchCopy` on Windows or application-specific implementations). CWE-126 identifies a common pattern of manually adding string terminators that can be replaced by these functions; doing so is a useful API migration task.

Figure 2 shows the three patterns identified in CWE-126 that can be refactored for safety. The first case ❶ is the simplest: a call to `strncpy` is immediately followed by an explicit termination. This migration could be performed using tools such as Refaster [53], but would require an expert to encode it manually.

The second case ❷ highlights the utility of M^3: after performing inlining, the code that explicitly calls `strncpy` is no different to code that performs an explicit loop. Both of these patterns exist in real code, and can be migrated equivalently using M^3. Because many different syntaxes might represent the same semantics, writing source-code based tools that discover loops in this way is a hard problem [20]; M^3's compiler integration and IR-level search allows it to handle loops and other control flow statements seamlessly.

Finally, the third case ❸ shows a complex migration where calls to `strncpy` are interleaved with their respective terminations. By operating at the IR level, M^3 is able to identify that no dependencies exist between the calls, and so the migration is possible. In general, source code-based tools, even with expert knowledge, are less able to make this determination.

Unifying these different forms of migration without requiring up-front expert knowledge or library source code is the key advantage of M^3.

3 MODEL

The Model phase of M^3 is a program synthesizer; it aims to generate functions that behave equivalently to target library functions. Model uses component-based sketching [24] together with novel learned probabilistic models to efficiently search for the most likely structure for correct solutions. Then, a gradual refinement process is used to instantiate working programs from these structures. Candidates are tested against the target function using randomly-generated inputs; the adequacy of this testing strategy is validated using branch coverage.

3.1 Correctness

Providing a formal proof of total correctness for this type of synthesis problem is extremely complex [15]. In this paper, we define correctness using the standard formulation of *observational equivalence*: a candidate is correct if it behaves identically to the target over a particular set of input examples. Most work in synthesis using input-output examples shares this formulation [14, 47].

This definition relies on a good enough set of input examples being available. We cannot rely on the user knowing enough of the target's semantics to produce a set of minimal, interesting examples [19, 33] (and in fact wish to abstract this process away from the user). We therefore resort to random generation of input examples.

3.1.1 Generating Test Inputs. The Model phase supports the primitive C types **`char`**, **`int`** and **`float`**, and pointers to these types. Values of integer and floating-point types are generated by sampling values uniformly in the range $[-64, 64]$, and for characters from their entire range. For pointer data, blocks of 4,096 elements are allocated (to allow for large computed indices based on input data). Each element of these blocks is sampled according to the appropriate scalar sampling method.

Existing work on fuzzing and automated testing [60] generally observes that interesting behavior most often occurs at small input values; our input range was selected to provide a varied distribution of values while also maintaining a reasonable probability of generating small (and therefore interesting) inputs.

Our input generation methodology can be easily generalized to more types; further primitive types (e.g. differently sized integers or

booleans) follow the same methodology, while aggregate types (e.g. a C **struct**) can be generated compositionally over their individual elements. Generating data structures with internal invariants or unusual distributions is an open problem [46].

3.1.2 Testing Coverage. It is important that the randomly generated inputs properly exercise the possible behaviors of both the target and candidate. While it is not possible to measure coverage for a black-box target in the absence of source code, we measure *branch coverage* over each candidate during synthesis. New inputs are generated until full coverage is achieved.

Our results in Section 7.2 show that random testing and coverage measurement is an effective means to validate the behavior of synthesized programs.

3.2 Specification

Two inputs fully specify a synthesis problem: the type signature and name of the target function, and a library containing an implementation with that name. There are no requirements on the internal details of this implementation.

No further information about the target function is required. For example, base cases or semantic annotations (such as those used by λ^2 [19] or in the type-directed synthesis procedure demonstrated by Collie et al. [11]) are not required by our implementation, and we do not require manually created inputs to test candidate programs as other synthesizers such as Simpl [47] or SketchAdapt [33] do.

3.3 Fragment-Based Sketching

Program synthesis commonly divides the search for a solution into two phases. The first, sketching, aims to establish the *structure* of a solution. In its initial formulation, sketches were provided by the user based on their insight into the problem [48]. By doing so, search for programs with complex control flow could be reduced to more tractable problems. More recent approaches aim to synthesize the sketch as well [17, 33, 52]. Our approach falls into this group as it does not require the user to provide any sketch information. Instead, it uses a novel probabilistic approach.

We aim to build sketches compositionally from smaller *fragments*, which represent independent elements of program structure. For example, a program that performs a linear search may comprise a loop fragment composed with a conditional test fragment. Some fragments are parameterized; in these cases different variants of the fragment are instantiated depending on the available information for a given problem. The full set of fragments used by M^3 to perform synthesis are listed below, along with C-like pseudocode describing their semantics.

3.3.1 Fragment Corpus. The library of fragments used by Model is given below. The `use` function represents code generated that may use a particular value, `?` is possible composition, and `_P` is a placeholder value of appropriate type.

Linear A basic block into which instructions should later be synthesized.

Fixed Loop Template for a loop with known upper bound, parameterized on an optional pointer `ptr` and an integer `x`:

```
for(int i = 0;i < x;++i) { ? }
for(int i = 0;i < x;++i) { use(ptr[i]);? }
```

Delimiter Loop Template parameterized on a pointer `ptr`:

```
while(*ptr++ != _P) { use(*ptr); ? }
```

Loop A catch-all for iterations not covered by the two more specialized fragments:

```
while(_P) { ? }
```

If, If-Else Conditional control flow:

```
if(_P) { ? }
if(_P) { ? } else { ? }
```

Seq Execute two fragments, one after the other:

```
? ; ?
```

Affine, Index Synthesize affine and general index expressions respectively, parameterized on `ptr`. For example:

```
int a_v = ptr[_P * _P + _P]; // e.g.
int v = ptr[_P - _P];        // e.g.
```

The available set of fragments for a synthesis problem depends on which ones can be properly instantiated; we write **F** for this set. In this work we use only the fragments described above, but it is possible for users to extend the corpus of fragments (for example, to specialize for a particular problem domain with partially-known structure).

3.3.2 Composition. We define an intuitive composition operation between any two fragments, with $\circ$ the left-associative composition operator.

3.4 Probabilistic Models

The set of possible fragment compositions for some problems is very large. Model uses two cooperating probabilistic models to reduce the size of the search space. The first predicts which fragments from the available set are most likely to appear anywhere in a correct solution, and the second uses a Markov model to identify compositions of fragments most likely to yield a correct program.

3.4.1 Fragment Likelihood. We use a random forest classification model to predict, for each fragment $f \in \mathbf{F}$, whether it appears in a correct solution program. The classifier C takes as input a fragment f and type signature τ, and outputs a prediction of whether this fragment will appear in a correct solution for a function with that type signature. Applying the classifier C to every fragment produces a predicted set of fragments $\mathbf{F_0} \subseteq \mathbf{F}$:

$$\mathbf{F_0} \triangleq \{f \in \mathbf{F} \mid C(f, \tau)\}$$

We achieved a mean Jaccard score of 0.82 between **F** and $\mathbf{F_0}$ using this predictor; this means that the predictor does not significantly over- or under-approximate.

3.4.2 Composition Sampling. There are a large number of potential compositions over $\mathbf{F_0}$ that produce a sketch. It is therefore important to predict which compositions are the most likely to produce a correct solution.

We equate the linear sequence of fragments $f_1, f_2, \ldots, f_n$ with the composition $f_1 \circ f_2 \circ \cdots \circ f_n$. This allows us to sample compositions

using a simple Markov model. To do so, we add start and end symbols to the fragment vocabulary, and sample fragments according to the probability $\mathbb{P}(f_n|f_{n-1})$ until the end symbol is sampled.

The conditional probability $\mathbb{P}(f_n|f_{n-1})$ is trained using observed fragment composition bigrams. For example, if a sketch from composition $f \circ g \circ h$ produces a correct solution, then the bigrams $f \circ g$ and $g \circ h$ are both observed. Based on a matrix w of observation counts (where $w(f, f')$ is the observed count of $f \circ f'$), we define:

$$w'(f_i, f_j) \triangleq \begin{cases} w(f_i, f_j) & \text{if } f_j \in \mathbf{F_0} \\ 0 & \text{otherwise} \end{cases}$$

$$s(f) \triangleq \sum_{f \in \mathbf{F_0}} w(f, f')$$

$$s'(f) \triangleq \sum_{f \in \mathbf{F_0}} w'(f, f')$$

Then, the Markov probabilities can be given as:

$$\mathbb{P}(f_n|f_{n-1}) \triangleq b\frac{w'(f_n, f_{n-1})}{s'(f_{n-1})} + (1-b)\frac{w(f_n, f_{n-1})}{s(f_{n-1})}$$

where $b \in [0, 1]$

3.4.3 Training. To train these models, a 25% subset of our evaluation library functions was selected randomly. Ground truth sketches were constructed by hand for each target function in this subset and used to train both models. A 25% training set split was identified through manual parameter search; there is enough redundancy among the functions to enable the use of a small training proportion.

We do not believe the construction of such a training dataset to be particularly onerous when compared to statistical migration techniques, which often entail cleaning and preprocessing millions of lines of code. Additionally, it is possible to *bootstrap* our training set starting from the synthesized solutions to simple problems. Doing so provides no benefit to the model performance (only to the collection of training data), and so in this paper we do not examine the process.

3.5 Instruction Search

The final step in Model's synthesis process is to perform an enumerative search for candidate programs based on predicted sketches. Each fragment specifies a set of typed placeholder values; these identify where computation can be performed within that fragment. For example, in the LLVM code below, the values `%0` and `%2` represent (possibly distinct) values of type **`i32`**. Placeholders may also be untyped (`%1` below).

```
%0 = call i32 @ph_i32()
%1 = call void @ph()
%2 = call i32 @ph_i32()
%3 = call i1 @ph_i1()
```

To search for candidate programs based on a sketch with placeholders, Model assigns concrete values to each placeholder in turn; different choices of values produce different programs. As values are selected, the potential choices for other values may be restricted. The result of this is a lightweight constraint-solving problem (for example, if the value **`add i32`** `%0, %1` were selected for `%2`, then the type of `%1` would be restricted to **`i32`**). Precise details of how this constraint problem can be implemented by the compiler are given in prior work [12]. To optimize the traversal of a potentially large search space, Model uses the following heuristics:

- Placeholders of known type are assigned first.
- When selecting operands for unary or binary operators, operands located closer to the operator are prioritized.
- More common operators are prioritized (e.g. addition is attempted before division).
- A threshold for the total number of instructions is set and iteratively relaxed (e.g. initially programs of 3 concretized instructions are considered, then 4 if no successful candidate is found, etc.).

By assigning values in this way, a concrete program is gradually refined from a sketch. Fragments are not required to enforce concrete types on their constituent values, but can enforce constraints when they do (e.g. a conditional fragment requires a boolean value).

Once we have a complete program, we can compile and execute it using randomly generated input values.

4 MATCH

Once Model has synthesized a program with behavior equivalent to a target library function, the next step is Match: we aim to discover regions of code that are equivalent to the synthesized implementation.

4.1 Searching for Code Using CAnDL

Efficiently searching for sections within an application that satisfy particular criteria is a hard problem to express using traditional programming languages. The CAnDL language [20] allows for declarative specification of search patterns, which are compiled to constraint-satisfaction problems that can be efficiently resolved using backtracking search (in a manner similar to SMT solvers [4]).

CAnDL patterns specify dataflow relationships between values in LLVM IR programs, as well as properties of individual values. For example, the property "x is an add instruction, and y is a multiply with x as one of its operands" is a simple CAnDL pattern. These patterns amount to a set of *constraints* on the program that must be satisfied for the pattern to match it; searching for matching code is therefore a constraint satisfaction problem.

We use the standard CAnDL toolchain to efficiently solve such constraint problems over LLVM. Full details of the search algorithms can be found in [20]; in this paper we take as given an efficient solution procedure for CAnDL-compatible constraint problems over LLVM IR programs.

4.2 Translating LLVM to Constraints

In [20] the authors write CAnDL constraints by hand to match specific computational idioms (for example, polyhedral control flow or stencil codes) of interest to a domain-specific optimizer. By comparison, we aim to generate constraints automatically from synthesized code. Automatic generation of constraints is not a use case envisioned by the original authors, and so we contribute a novel algorithm for emitting constraint descriptions from example LLVM programs.

```
%iter = phi i64 [%new_iter,%loop], [0,%entry]
%addr = getelementptr i64,
          i64* %array, i64 %iter
%elem = load i64, i64* %addr
%niter = add i64 %iter, 1
```

(a) Fragment of LLVM code extracted from a function that computes the sum of an array of integers.

```
1   Constraint Generated
2   ( opcode{iter}       = phi
3   & opcode{addr}       = gep
4   & opcode{elem}       = load
5   & opcode{niter}      = add
6   & ir_type{0}         = literal
7   & ir_type{1}         = literal
8   & {niter}            = {iter}.arg[0]
9   & {0}                = {iter}.arg[1]
10  & {array}            = {addr}.arg[0]
11  & {iter}             = {addr}.arg[1]
12  & {addr}             = {elem}.arg[0]
13  & {iter}             = {new_iter}.arg[0]
14  & {1}                = {new_iter}.arg[1])
15  End
```

(b) CAnDL constraints generated from the LLVM code above. These constraints capture the structure of the code and can be efficiently searched for in large LLVM code bases.

Figure 3: LLVM code sample and its corresponding CAnDL constraints, as generated by Match.

Figure 3a shows a small fragment of LLVM IR; the code is in SSA form and can be described by a directed acyclic graph. Below, Figure 3b gives a set of CAnDL constraints that describe this fragment. Each instruction (as well as constants and function parameters) occurs as a variable name in the constraints; the constraint program serves as a description of the data flow. The data flow graph is serialized by classifying individual variables (lines 2–10), and then the interactions between them (lines 11–21). This description is passed to the CAnDL solver to efficiently find satisfying code.

Our constraint descriptions are built from a dataflow graph representation of LLVM IR, where vertices are instructions and edges capture the argument relation. Algorithm 1 shows how we generate a description of this graph structure.

Looping over the graph vertices (lines 4–17), the instruction opcode constraints are emitted, as well as the constraints that deal specifically with constant and function argument values. In a second loop (lines 18–20), the data flow graph is serialized by iterating over the graph edges and emitting positional argument constraints. The remaining lines of the algorithm generate the logical conjunctions holding the individual constraints together (lines 5–9) and produce the boilerplate CAnDL code (lines 2 and 21).

4.3 Post-Processing Constraints

This approach results in a constraint program that searches for exact sub-graph matches in user code, but is often too specific. We therefore apply a careful weakening of the constraints to produce a more general matching.

Algorithm 1 Emit Constraint Description

```
1:  function EmitConstraints(V,E)
2:      emit("Constraint Generated (")
3:      first ← true
4:      for v in V do
5:          if first then
6:              first ← false
7:          else
8:              emit("&")
9:          end if
10:         if op(v) = parameter then
11:             emit("ir_type", name(v), " = argument")
12:         else if op(v) = const then
13:             emit("ir_type", name(v), " = literal")
14:         else
15:             emit("opcode", name(v), " = ", op(v))
16:         end if
17:     end for
18:     for n, a, b in E do
19:         emit(name(a), " = ", name(b), ".args[", n, "]")
20:     end for
21:     emit(") End")
22: end function
```

Firstly, constraints that specify values to be function arguments are counterproductive; these constraints will not hold after inlining, so they are removed in post-processing. Secondly, some operators are commutative and therefore the positional argument constraints on them are too strict. They are replaced with a logical disjunction between the corresponding permutations. Finally, we remove instructions that correspond only to compiler-specific code generation idioms.

5 MIGRATE

Model and Match make up the bulk of the work done by M^3. The final step is to leverage the synthesized programs and generated constraints to generate appropriate API migrations. We produce source-level substitutions that can be applied manually to the source code, as well as automatically-tested IR transformations.

5.1 IR-level Replacements

Migrate is able to automatically apply a potential migration within an application being compiled. To do this, the IR values that matched against a library function's parameters and return value are identified. A call to the function is inserted with the appropriate arguments given, and uses of the matched return value are replaced with the new call's return value.

Table 1: Corpora used to evaluate M^3.

(a) Application source code for which migrations were tested.

Software	Description	LoC
`ffmpeg`	Media processing	1,061,655
`texinfo`	Typesetting	76,755
`xrdp`	Remote access protocol	75,921
`coreutils`	Utilities	66,355
`gems`	Graphics helpers	46,619
`darknet`	Deep learning	21,299
`caffepresso`	Deep learning	14,602
`nanvix`	Operating system	11,226
`etr`	Game	2,399
`androidfs`	Filesystem	1,840

(b) Library APIs for which synthesized implementations were learned and used to drive migration.

Library	Description
`string.h`	C standard library string handling
`StrSafe.h`	Safety-focused C string handling
`glm`	Graphics functions
`mathfu`	Mathematical functions
`BLAS`	Linear algebra
`Ti DSP`	DSP Kernels
`ARM DSP`	DSP Kernels

By doing this, we obtain a modified version of the application's code. Regions that match the generated constraints for library functions are replaced with calls to those library functions. Migrate extends the functionality used in the original CAnDL paper [20] by not requiring the migration process to be implemented manually for every relevant library function; having the synthesized code available to map values allows us to do this.

5.2 Validation

The primary usage of automated IR replacement is to validate migrations (i.e. to check whether or not performing the migration will result in a correct program). While formally proving this is unlikely to be possible for any API migration tool, Migrate performs two validation steps that provide some assurance that its suggestions are correct. First, we ensure that no dependencies to intermediate values in the pre-replacement code exist later in the function. Then, we test the code post-replacement with random IO examples using the same methodology as Model uses; our results in Section 7.4 show that this validation is effective.

Beyond these checks, the user is likely to still perform their own validation (e.g. running unit or integration tests). Other API migration tools share this characteristic; no changes suggested by refactoring tools to any codebase are likely to go untested.

5.3 Source-level Suggestions

Our methodology for this paper operates at the IR level, within the compiler; migrations are applied mechanically by performing substitutions of SSA values. Doing so allows us to automatically test applied migrations, but changes made at the IR level can be difficult for a user to understand.

We implemented a prototype tool that used LLVM's debugging libraries to generate source-level suggestions instead. Source-level suggestions are harder to apply mechanically, but allow for easier user insight into what changes have been made by the migration. Evaluating this tool is outside the scope of the paper (as its usage was not necessary for any of our research questions), but we hope to implement it more fully and perform a user study as future work.

6 EXPERIMENTAL DESIGN

To evaluate the success of M^3, we identify four research questions:

(RQ1) Feasibility and effectiveness of the Model phase: Can program synthesis be used effectively to learn the behavior of black-box library functions?

(RQ2) Correctness of synthesized programs: Do the synthesized programs behave the same as the target program over a particular set of inputs? The inputs used for this correctness check are randomly generated. To assess the adequacy of the random inputs in checking behaviors of the synthesized and target programs, we measure branch coverage achieved by the random inputs over them.

(RQ3) Accuracy of Match phase: Given synthesized implementations for library functions, can compatible instances in application code be accurately discovered? In this research question, we focus on ability and accuracy of the Match phase to discover inlined implementations of the *same* synthesized library functions in application code.

(RQ4) Accuracy of Migrate phase: Given instances of user code that match the constraints generated from a library function, can API migrations be correctly implemented? This research question investigates ability and accuracy of the Migrate phase in matching and migrating implementations in application code to *different* library functions.

6.1 Evaluation Corpora

6.1.1 Applications. We selected 9 widely-used applications to evaluate our approach against; they are listed in Table 1a. Each application is written in C or C++, and they cover a wide range of problem domains.

We selected these applications by manually searching GitHub and similar online repositories[1] for code that matched the following criteria: most importantly, we required a build system that permitted easy interposition of our compiler toolchain. For our purposes, this ruled out applications not written in C or C++, although with some additional engineering work any language with an LLVM frontend could be integrated.

When selecting applications, large, popular and real-world code was prioritized. We selected projects in active development, or those for which significant distribution and usage could be identified. We

[1] Using https://searchcode.com/

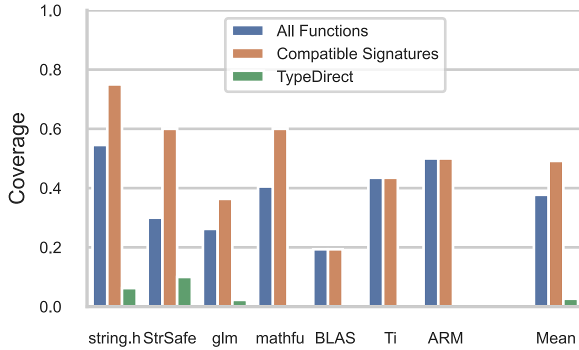

Figure 4: Proportion of each library's API that we were able to successfully synthesize, across all functions in the library as well as those with compatible type signatures. Results are also shown for TypeDirect [11].

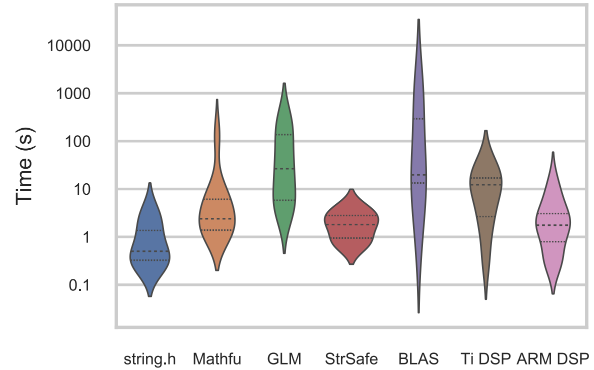

Figure 5: Distribution of synthesis times for each library API.

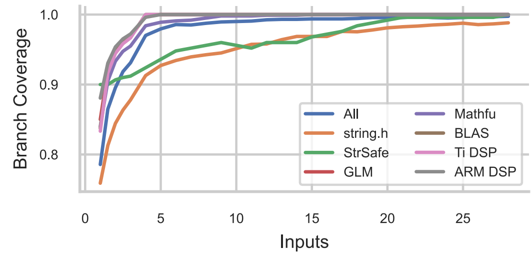

Figure 6: Corpus branch coverage achieved using randomly generated inputs. Coverage values are reported as the mean of three separate runs.

aimed for a diverse range of application domains with minimal duplication. No pre-selection of applications based on knowledge of their source code was performed; the authors were not familiar with these applications in advance.

6.1.2 Libraries. We selected 7 libraries to target for migration, from two broad problem domains: string processing and mathematical operations. Similar domains are commonly targeted by other migration tools (with different tooling and language contexts). We required libraries that could be called easily from C/C++ for compatibility with the synthesizer.

For string processing, our starting point was the standard C `string.h` header along with BSD extensions. We additionally selected the Microsoft `StrSafe.h` library that extends the standard functions with safer alternatives that avoid common security issues. We then selected five mathematical libraries with slightly different areas of application and platform support in order to evaluate the ability of M^3 to discover cross-vendor or cross-platform migrations. Other work [11] identifies the usefulness of this type of migration. Full details of the selected libraries are given in Table 1b.

7 RESULTS

7.1 RQ1: Feasibility and Effectiveness of Model

7.1.1 Library Coverage. Figure 4 shows the proportion of each library's API we were able to synthesize correctly across all functions in the library (shown as blue bars). As expected, we could not synthesize every function from each library. The primary reason for a synthesis failure was a function's type signature not being compatible with Model, for example those using pointers to pointers or complex structure types. Beyond these failures, there were a number of cases where internal data structure usage meant that Model's control flow fragments were not able to express the necessary structure (e.g. the control flow required to operate on the packed matrices in `strsm` from BLAS).

Model successfully synthesizes implementations for an average of 37% of the functions in each library evaluated (blue bars in Figure 4). Considering only functions with type signatures compatible with Model (brown bars in Figure 4), we were able to synthesize implementations for nearly 50% on average. This represents a significant proportion of each library's behavior—even in our worst performing case (BLAS), we are able to synthesize nearly 20% of all functions in the library. Performance on the BLAS library is limited by the high complexity of many of its constituent functions (e.g. solving systems of equations on packed matrix structures).

For each synthesis failure in our set of evaluation functions, we examined the reference function by hand to determine why it could not be synthesized. In some cases (e.g. `strtok` from `string.h`), the function demonstrated stateful behavior. Modeling this type of function is an open problem in program synthesis, with recent work addressing limited contexts such as heap manipulation [39]. Our synthesis methodology presumes that target functions are idempotent, and so does not support stateful functions. Doing so is interesting future work. A small number of functions (e.g. `ssyr2k` from `blas`) exhibit unusual control flow idioms not expressible using our set of fragments. However, the majority of failures are timeouts resulting from long required sequences of instructions in target functions.

Program synthesis over an entire library API is a challenging problem; the programs that we were able to synthesize are considerably more complex than comparable work in program synthesis while requiring less information to do so. [41, 47].

7.1.2 Difficulty. Figure 5 shows the distribution of candidate functions evaluated for the synthesis problems in each library. From these distributions we see that the majority (84%) of functions were synthesized in less than 2 minutes. We were able to evaluate approximately 1,000 candidates per second on an 8-core desktop-class machine.

The distribution of synthesis times is long-tailed; only two functions from the BLAS library took more than 2 hours to synthesize. These synthesis times are comparable to existing work in program synthesis, and could be further improved by using techniques such as hill-climbing to guide the search process.

7.1.3 Comparison to TypeDirect. We were only able to identify one other program synthesizer with library functions and migrations as an explicitly stated goal. In that work, partial semantic knowledge and type information is used to guide a synthesizer (TypeDirect) towards synthesized implementations of performance bottleneck functions [11]

Evaluation of TypeDirect is limited to 12 such functions, with synthesis guided by annotations that specify semantic properties of the target functions [11]. We restated our set of synthesis problems for TypeDirect and recorded how many it could synthesize. These results are compared to those achieved by Model in Figure 4 (green bars); Model performs significantly better across all the libraries evaluated, with TypeDirect failing to synthesize any function in four of the seven libraries. Additionally, TypeDirect took longer to synthesize the functions for which it was successful (up to 4 hours in some cases).

In sum, compared to TypeDirect, we find Model is (1) automated and easy to use, not relying on annotations to guide synthesis and (2) more widely applicable with better synthesis coverage across different libraries. This is due primarily to TypeDirect's focus on synthesis for specific accelerator libraries rather than general API migration.

7.2 RQ2: Correctness of Synthesis

For every synthesized library function, we automatically generated random and boundary value inputs and checked if outputs matched those from the target black-box function.

Random Testing. We generated test inputs for every synthesized candidate by uniformly sampling values in the range of the input data types, as described in Section 3.1.1. We found all the synthesized library implementations were behaviorally equivalent to the target functions with respect to the random inputs generated for them.

Manual Check. As well as testing using random IO examples, we examined each synthesized solution manually using our knowledge of their intended behavior. Only one program was judged to be incorrect: the `memmove` function from `string.h`. If the memory regions passed as arguments aliased (i.e. they overlapped), the synthesized implementation would exhibit incorrect behavior. Our testing methodology did not generate aliased memory. We generated a set of aliased inputs manually and were able to correctly synthesize `memmove`.

Table 2: Number of call sites where synthesized functions were inlined in each application, along with the proportion of these that were successfully rediscovered using Match.

Application	Inlined Calls (L→L) #Instances	Inlined Calls (L→L) Matched (%)	# User Code Matches (C→L)
`ffmpeg`	4,976	100%	24
`texinfo`	586	100%	1
`xrdp`	686	100%	0
`coreutils`	623	100%	16
`gems`	46	100%	61
`darknet`	128	100%	13
`caffepresso`	189	100%	0
`nanvix`	0	100%	16
`etr`	4	100%	45
`androidfs`	0	100%	2
Total	**7,238**		**178**

Boundary Value Testing. We additionally tested each synthesized candidate using boundary and outside range values for inputs. In every case, the synthesized candidate conformed to the expected behavior on these inputs.

Adequacy of testing. We assessed the adequacy of the generated inputs in exercising behaviors of the synthesized implementations by measuring the branch coverage achieved. Figure 6 shows the branch coverage achieved across the full set of library functions evaluated. With as few as 10 distinct inputs, more than 98% of the branch choices in our corpus of synthesized programs are evaluated. Typically, at most around 30 random inputs are needed to provide 100% branch coverage for a synthesized candidate. The numerical libraries we evaluate most often contain loops as their primary control flow; branch coverage is less difficult to achieve over looping code than over conditionals.

These results provide confidence that the synthesized candidates behave equivalently to the target program with respect to inputs that exercise the complete control flow in the candidates.

Inside the Black Box. In many cases, we had the source code for libraries, making it possible to directly compare our synthesized programs to the original code by "looking inside" the black-box. These programs were compiled to LLVM IR and used as input to the Match and Migrate phases as if they had in fact been synthesized. We did not identify any meaningful divergence in results; we achieved similar per-library branch coverage, and the compiled IR for synthesized and handwritten implementations was almost identical in most cases. No behavioral differences were observed.

7.3 RQ3: Accuracy of Match

We assessed if the constraint descriptions of every synthesized library implementation was able to match the inlined implementation of the same library in application code (**L→L**). Match is able

Table 3: Migration opportunities discovered in each application, broken down by the category of the source context (source library calls L or user code C).

Application	Migrations	Category		
		L→L'	C→L'	L+C→L'
ffmpeg	655	629	24	2
texinfo	431	413	1	17
xrdp	274	269	0	5
coreutils	649	633	16	0
gems	107	46	61	0
darknet	40	7	13	20
caffepresso	24	24	0	0
nanvix	16	0	16	0
etr	49	4	45	0
androidfs	2	0	2	0
Total	**2,247**	**2,025**	**178**	**44**

to successfully identify every instance of inlined code across all the applications we evaluated; the number of inlined instances for each application is given in Table 2. This is because the same code is inlined at each site, and because inlining does not change the structure of the code from which the constraint description was generated.

As well as being able to successfully identify inlined calls, Match is able to identify locations in the application code where equivalent functionality to a library function is implemented, **C→L** (number of instances shown in Table 2). We performed a manual search for further instances not discovered by Match based on these results. A combination of several techniques was used to perform this search: we used handwritten CAnDL constraints for significantly abstracted versions of each function to guide an initial search, as well as textual similarity and heuristic exploration of the code.

For example, where a re-implementation of one string-processing function was discovered, we searched by hand for similar re-implementations that were not discovered by Match. For a region to be classified as a re-implementation, we required that on well-formed inputs (i.e. not accounting for "exceptional" control flow), the region performs the same task as the original function.

No further instances of this kind were identified by this search, confirming with reasonable certainty that there were no false negatives from Match (though no technique can verify this formally). The constraints generated by Match were specific enough that none of the application code matches represented false positives.

Running the CAnDL solver takes additional time during compilation; approximately the same as compilation itself for each pattern to be searched for [20]. This time is not a bottleneck when using M^3 practically.

7.4 RQ4: Accuracy of Migrate

For every synthesized target library function, we assessed in how many cases the generated constraints for that function matched application code that was not originally a call to that function. This quantifies the number of possible migrations enabled by M^3. Table 3 gives the total number of migrations found in each application, as well as a breakdown into three categories:

- Replacement of a source function with a semantically identical target function from a different library (**L→L'**).
- Identification and replacement of redundant application code that could be better expressed as a target library function call (**C→L'**).
- Replacement of code that *combines* a source library call and handwritten code with a target function (**L+C→L'**).

The most common migrations were **L→L'**, where two libraries implemented the same function (for example, delimited string copying or a vector dot product). Some functions did not produce migration opportunities, even though they could be inlined and matched. memcpy is an example of this; applications like ffmpeg and xrdp that frequently perform buffer copies show far fewer migrations than inlined matches.

Note that the category **C→L'** corresponds exactly to the number of matches in user code (**C→L**) quoted in Table 2. This is because any matching instance of a function in application code represents a migration opportunity; there is no original function whose matches we are not interested in.

These results demonstrate that M^3 is able to successfully identify distinct classes of migration (other tools are often limited to one of these classes only, and **L+C→L'** migrations generally require expert knowledge to express). The migrations we identify are useful and would be difficult to identify with existing tools.

7.5 Threats to Validity

We find M^3 is able to identify and perform a large number of useful migrations using real-world applications and libraries, in contexts not well served by existing tools. The primary threats to *internal validity* are: (1) The fragment vocabulary used by Model is a limiting factor; the variety of programs that can be synthesized depends on this vocabulary. However, this is a limitation shared by all sketching program synthesizers. (2) Our CAnDL constraint generation is not formally verified; we rely on testing with different library functions to check constraints always match their source programs. (3) We check the correctness of the synthesized implementations against target functions using test inputs that achieve branch coverage. Proving total correctness is known to be challenging [15], especially when the target source code is not visible.

The main threat to *external validity* lies in the subject libraries chosen and the restriction to two problem domains: string processing and mathematical operations. These domains have also been targeted by other migration tools and we used these to facilitate comparison. Our synthesis technique is not restricted to these domains and we will apply our techniques to other domains in the future; extending the vocabulary of fragments to include more expressive computations will allow us to scale synthesis to more complex APIs and functions.

8 RELATED WORK

8.1 Library Migration

(Semi-)automatic rewriting of application code to use new libraries has been well studied, particularly for Java and other object-oriented

languages [55, 59]. Robillard et al. [40] partition migration techniques into 3 sub-areas: library upgrade [16, 56], API evolution [13, 44] and library migration [31, 54, 61]. Many schemes rely on a large corpus of programs using the old and new libraries, frequently focusing on change logs [51]. This ongoing need is highlighted by Alrubaye and Mkaouer [2], whose work aims to automatically identify key changes that produce a migration.

8.1.1 Automatic. Different levels of abstraction delineate automatic approaches. Similarity of text description has been used to map old to new APIs [36], while others [30, 32, 38] use a syntactic view of programs to build a learned vector-space encoding [28] for migration given an initial parallel mapping. Although the embedding-based approach taken by API2Vec [32] is flexible, the resulting ambiguity is in fact a hindrance when performing migrations. More recent work attempts to generate mappings between APIs based on their usage [7].

Other work [57, 58] goes beyond simple replacement of library API calls. Xu et al. [57] use syntactic program differencing and program dependency analysis to target actual edits and replacements. Although it is a syntactic rather than semantic approach, they are able to add new code to help migration of libraries. EdSynth [58] synthesizes candidate API calls to fill partial program using information from test executions and method constraints. Unlike our synthesis approach, their work requires white-box information on candidate methods, exact locations to insert API calls, and a user-provided test suite to serve as a correctness specification.

Closer to our aim of not relying on prior API mapping examples is the approach taken by Bui [6]. It uses GANs to generate initial migrations (seeds) rather than using human knowledge to do so [5]. To achieve this, it makes the assumption that use of APIs when migrating remains roughly the same. It has significantly lower precision than our approach, relies on lexical similarity and cannot perform **C→L'** migrations. Other work uses specific semantic knowledge of functions to perform refactoring with semantic guarantees [45].

8.1.2 Expert-Driven. A different approach to API migration is to use expert knowledge to encode migration patterns by hand, then compile them to a searchable representation to perform migrations. This approach is taken by tools like ReFaster (Java) [53] and No-Brainer (C / C++) [43]; they permit complex migrations but require experts to create the migration patterns initially. Similarly, IDL [21] implements migrations of computational "idioms" to target heterogeneous computing platforms. The underlying code search mechanism for M^3 (CAnDL [20]) can be used to implement this style of migration tool in a portable way. M^3 extends prior work by adding a *learning* phase that creates migration patterns automatically.

8.2 Program Synthesis

M^3 uses program synthesis as a technique for modeling the behavior of library functions. We give a brief overview of related work in synthesis.

Prior work in imperative synthesis frequently focuses on straight-line code [23, 42] or has to make special provision for control-flow [22]. Simpl overcomes this problem by assuming a partial program is already provided (such as a loop structure) [47]. Other work aims to complete suggested sketches [49] of programs to provide programmer abstraction and auto-parallelization [18]

Type signatures and information are often used to direct program synthesis, most commonly for functional programs [34, 35]. Other work uses extended type information as a means of accessing heterogeneous accelerators [11]. Our work considers a much wider, more diverse class of libraries and applications without additional annotation.

Others have used neural components to improve the performance of an existing synthesizer. For example, both DeepCoder [3] and PCCoder [62] aim to learn from input-output examples; both require fixed-size inputs and outputs and use a small DSL to generate training examples. Learned programs are limited to list processing tasks; the DSLs targeted by these (and similar implementations such as SketchAdapt [33]) also rely on high level primitive including (for example) primitives to tokenize strings or perform list manipulations.

Operating under the assumption of a black-box API means that many existing approaches in program synthesis do not apply or fail to generalize to our context [8, 9]. By using a black-box oracle we are able to avoid issues of generalization across datasets [25, 37].

9 CONCLUSION

In this paper we have proposed a novel API migration problem that matches real-world problem contexts. Our approach, M^3, uses the behavior of library functions to discover migrations without expert knowledge, changelogs, or access to the library's source code.

We successfully applied this approach to 7 large, widely-used libraries and were able to successfully synthesize nearly 40% of their functions. We discovered over 7,000 instances of these functions in 10 well-known C/C++ applications, and were able to discover a number of missed optimizations and bugs.

Using constraint-based search for API migration allows for the *semantics* of the code in question to be accounted for, rather than just the contexts in which they appear; this results in more precise migrations. Future work applying these methods to more domains is likely to be interesting.

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics.

REFERENCES

[1] 2020. CWE-126: Buffer Over-read. https://cwe.mitre.org/data/definitions/126.html

[2] Hussein Alrubaye and Mohamed Wiem Mkaouer. 2018. Automating the Detection of Third-Party Java Library Migration at the Function Level. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON '18)*. IBM Corp., Riverton, NJ, USA, 60–71.

[3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. (Nov. 2016). https://arxiv.org/abs/1611.01989v2

[4] Clark Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer International Publishing, Cham, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[5] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: learning cross-language API mappings with little knowledge. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 796–806.

[6] Nghi D. Q. Bui. 2019. Towards Zero Knowledge Learning for Cross Language API Mappings. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE Press, 123–125. https://doi.org/10.1109/ICSE-Companion.2019.00054

[7] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. 2019. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering* (2019).

[8] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2019. Multi-Modal Synthesis of Regular Expressions. (Aug. 2019).

[9] Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal Multi-Layer Specification Synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 602–612. https://doi.org/10.1145/3338906.3338951

[10] Fabien Coelho and François Irigoin. 2013. API compilation for image hardware accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–25.

[11] B. Collie, P. Ginsbach, and M. F. P. O'Boyle. 2019. Type-Directed Program Synthesis and Constraint Generation for Library Portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 55–67. https://doi.org/10.1109/PACT.2019.00013

[12] Bruce Collie and Michael O'Boyle. [n.d.]. Retrofitting Symbolic Holes to LLVM IR. ([n. d.]). arXiv:2006.05875 [cs] http://arxiv.org/abs/2006.05875

[13] Barthélémy Dagenais and Martin P. Robillard. 2008. Recommending Adaptive Changes for Framework Evolution. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 481–490. https://doi.org/10.1145/1368088.1368154

[14] Cristina David and Daniel Kroening. 2017. Program Synthesis: Challenges and Opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (Oct. 2017), 20150403. https://doi.org/10.1098/rsta.2015.0403

[15] Yves Deville and Kung-Kiu Lau. 1994. Logic program synthesis. *The Journal of Logic Programming* 19 (1994), 321–350.

[16] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Springer-Verlag, Berlin, Heidelberg, 404–428. https://doi.org/10.1007/11785477_24

[17] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 6059–6068.

[18] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. 2017. Gradual Synthesis for Static Parallelization of Single-pass Array-processing Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 572–585. https://doi.org/10.1145/3062341.3062382

[19] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

[20] Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. 2018. CAnDL: A Domain Specific Language for Compiler Analysis. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3178372.3179515

[21] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 139–153. https://doi.org/10.1145/3173162.3173182

[22] 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. https://doi.org/10.1145/1926385.1926423

[23] 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. https://doi.org/10.1145/1993498.1993506

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

[25] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. *arXiv:1804.01186 [cs]* (Sept. 2018). arXiv:1804.01186 [cs]

[26] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. 2010. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conference*.

[27] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization*. Ph.D. Dissertation. Computer Science Dept., University of Illinois at Urbana-Champaign.

[28] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13)*. Curran Associates Inc., USA, 3111–3119. http://dl.acm.org/citation.cfm?id=2999792.2999959

[29] André Miranda and João Pimentel. 2018. On the Use of Package Managers by the C++ Open-Source Community. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. Association for Computing Machinery, Pau, France, 1483–1491. https://doi.org/10.1145/3167132.3167290

[30] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 511–522.

[31] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-Based Approach to API Usage Adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 302–321. https://doi.org/10.1145/1869459.1869486

[32] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 438–449.

[33] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to Infer Program Sketches. In *International Conference on Machine Learning*. Long Beach, CA, USA, 4861–4870.

[34] Peter-Michael Osera. 2019. Constraint-Based Type-Directed Program Synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2019)*. ACM, New York, NY, USA, 64–76. https://doi.org/10.1145/3331554.3342608

[35] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

[36] Rahul Pandita, Raoul Praful Jetley, Sithu D Sudarsan, and Laurie Williams. 2015. Discovering likely mappings between apis using text mining. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 231–240.

[37] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis. *arXiv:1611.01855 [cs]* (Nov. 2016). arXiv:1611.01855 [cs]

[38] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen. 2017. Statistical Migration of API Usages. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 47–50. https://doi.org/10.1109/ICSE-C.2017.17

[39] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 72:1–72:30. https://doi.org/10.1145/3290385

[40] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering* 39, 5 (May 2013), 613–637. https://doi.org/10.1109/TSE.2012.63

[41] Christopher D. Rosin. 2018. Stepping Stones to Inductive Synthesis of Low-Level Looping Programs. *arXiv:1811.10665 [cs]* (Nov. 2018). arXiv:1811.10665 [cs]

[42] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *arXiv:1711.04422 [cs]* (Nov. 2017). arXiv:1711.04422 [cs] http://arxiv.org/abs/1711.04422

[43] Valeriy Savchenko, Konstantin Sorokin, Georgiy Pankratenko, Sergey Markov, Alexander Spiridonov, Ilia Alexandrov, Alexander Volkov, and Kwangwon Sun. 2019. Nobrainer: An Example-Driven Framework for C/C++ Code Transformations. In *Perspectives of System Informatics (Lecture Notes in Computer Science)*. Springer International Publishing, Cham, 140–155. https://doi.org/10.1007/978-3-030-37487-7_12

[44] T. Schäfer, J. Jonas, and M. Mezini. 2008. Mining Framework Usage Changes from Instantiation Code. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 471–480. https://doi.org/10.1145/1368088.1368153

[45] A. Shaw, D. Doggett, and M. Hafiz. 2014. Automatically Fixing C Buffer Overflows Using Program Transformations. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 124–135. https://doi.org/10.1109/DSN.2014.25

[46] Richard Shin, Neel Kant, Kavi Gupta, Christopher Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. 2019. Synthetic Datasets for Neural Program Synthesis. *arXiv:1912.12345 [cs, stat]* (Dec. 2019). arXiv:1912.12345 [cs, stat]

[47] Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis (Lecture Notes in Computer Science)*, Francesco Ranzato (Ed.). Springer International Publishing, Freiburg, Germany, 364–381.

[48] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3

[49] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (2013), 475–495.

[50] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. 2020. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 104–115.

[51] C. Teyton, J. Falleri, and X. Blanc. 2013. Automatic Discovery of Function Mappings between Similar Libraries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 192–201. https://doi.org/10.1109/WCRE.2013.6671294

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

[53] Louis Wasserman. 2013. Scalable, Example-Based Refactorings with Refaster. In *Workshop on Refactoring Tools*.

[54] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim. 2010. AURA: A Hybrid Approach to Identify Framework Evolution. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 325–334. https://doi.org/10.1145/1806799.1806848

[55] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 138–147.

[56] Z. Xing and E. Stroulia. 2007. API-Evolution Support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33, 12 (Dec. 2007), 818–836. https://doi.org/10.1109/TSE.2007.70747

[57] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 335–346.

[58] Zijiang Yang, Jinru Hua, Kaiyuan Wang, and Sarfraz Khurshid. 2018. EdSynth: Synthesizing API sequences with conditionals and loops. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 161–171.

[59] Syed Sajjad Hussain Zaidi. 2019. *Library Migration: A Retrospective Analysis and Tool*. Master's thesis. Science.

[60] Michał Zalewski. 2020. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[61] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API Mapping for Language Migration. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 195–204. https://doi.org/10.1145/1806799.1806831

[62] Amit Zohar and Lior Wolf. 2018. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., USA, 2098–2107.