M^3: Black-Box Library Migration

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

Philip Ginsbach
philip.ginsbach@ed.ac.uk
University of Edinburgh
Edinburgh, United Kingdom

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

ABSTRACT

Managing library migration is a challenging problem for developers. It is not always clear when libraries are compatible with their code, and even when a target library is identified, correctly migrating to a new API is difficult.

This paper proposes a scheme to address this issue: M^3. It uses program synthesis to model the behavior of API functions in code targeted for migration. These synthesized models are translated to searchable constraint descriptions and used to discover matching instances in user programs. Finally, matches are tested and validated before reporting opportunities to the user.

We evaluate our approach against 7 well-known libraries from varied application domains, learning correct implementations for 62 functions. We discover over 6,000 instances of these functions in imperative user code, leading to 1,700 correct migration opportunities. Our approach is integrated with standard compiler tooling, and we used this integration to evaluate its application to significant existing C/C++ libraries with over 1MLoC.

KEYWORDS

library migration, program synthesis, constraint programs

ACM Reference Format:

Bruce Collie, Philip Ginsbach, and Michael O'Boyle. 2020. M^3: Black-Box Library Migration. In *Proceedings of ICSE '20: International Conference on Software Engineering (ICSE '20).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Libraries are a fundamental feature of software development. They allow the sharing of common code, separating concerns and reducing 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: library upgrade [13], API evolution [19] and library migration [4]. Although there is great diversity in approaches, prior work largely focuses on using examples from previous migrations to learn how to migrate well-structured object-oriented programs. What happens if there are no prior examples?

ICSE '20, May 23–29, 2020, Seoul, South Korea

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

1.1 M^3: Model, Match and Migrate

In this paper we focus on black-box library migration where we do not have access to commit logs or even either the old or new library source code; an extremely challenging task [5]. We investigate and evaluate a novel approach to reconstructing the underlying source code. Given a library function, our approach attempts to automatically *model* its behavior using program synthesis. It then uses compiler-based constraint analysis to *match* existing code and old function calls with new ones. Once the user confirms the match, we can *migrate* the code. 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.

While not requiring library vendors to release their source code is highly desirable, it relies on the ability to synthesize programs in a reasonable time. Given that there is an unbounded space of potential programs and that we must also determine function equivalence (which is undecidable in general), this is clearly a difficult problem.

In this paper we examine the use of small user-provided property annotations to API type signatures. These enable the modeling and matching of library APIs from a diverse range of libraries to code from different domains in a reasonable time. A similar approach is taken in [6], targeting the problem of exploiting heterogeneous hardware accelerators to improve the performance of scientific applications.

Across 7 libraries we synthesize 62 functions in total; 84% in under 20 seconds. We accurately match them to over 6,000 old library calls across 9 applications and migrate 1,700 of them. One key feature is that we rarely synthesize incorrect implementations; our approach is able to achieve over 90% sensitivity and successfully rejects false positive migrations using dynamic testing.

2 OVERVIEW

In this section we first give an a high-level summary of the M^3 workflow. As our compiler based approach operates on LLVM programs, we briefly describe its structure. This is followed by a small example of how our approach works in practice.

2.1 Summary of Approach

Figure 1 shows the flow of data through M^3. It takes as input a target program, along with specifications for old and new library APIs. The end result is a modified program that references the new library rather than the old one. We highlight the three distinct phases of this workflow: **Model**, **Match** and **Migrate**.

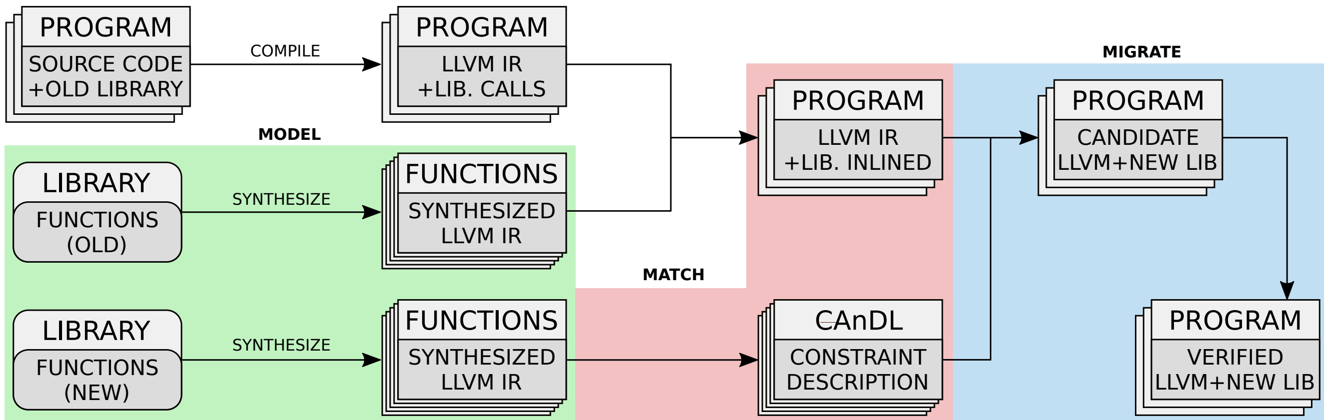


Figure 1: A summary of the M^3 workflow. Models for library functions are synthesized and generalized into constraint descriptions, which are then used to search compiled user code for potential migrations.

```
define i64 @sum(i64* %array, i64 %N) {
entry:
  %do_loop = icmp sgt i64 %N, 0
  br i1 %do_loop, label %loop, label %exit
loop:
  %it = phi i64 [%nit,%loop], [0,%entry]
  %sum = phi i64 [%nsum,%loop], [0,%entry]
  %addr = getelementptr i64,
          i64* %array, i64 %it
  %elem = load i64, i64* %addr, align 8
  %nsum = add i64 %elem, %sum
  %nit = add i64 %it, 1
  %end = icmp eq i64 %nit, %N
  br i1 %end, label %exit, label %loop
exit:
  %ret = phi i64 [0,%entry], [%nsum,%loop]
  ret i64 %ret
}
```

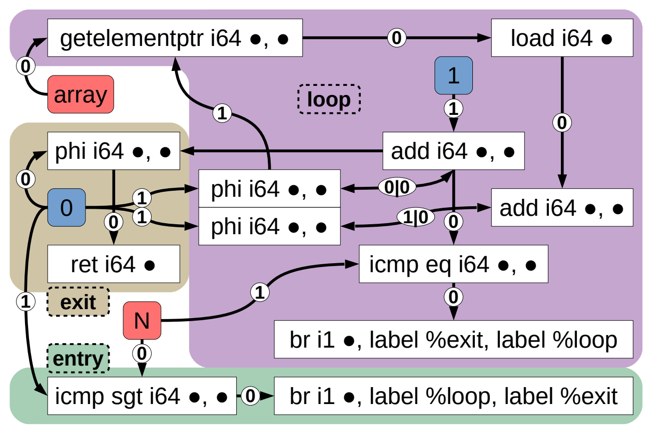


Figure 2: Example of LLVM IR code and the corresponding data-flow graph. Shown on the left is the textual representation of a simple C function compiled to LLVM, and shown on the right is its DFG. Basic blocks are highlighted regions, and nodes represent single instructions. Edges correspond to the argument relation.

2.1.1 Model. We assume that the source code for the old and new libraries is not available, as is often the case in practice for real-world libraries. The first phase of M^3 is Model: the synthesis of LLVM programs equivalent to the behavior of both the old and new libraries. We generate many different inputs and record the corresponding outputs from the library function; these pairs specify correct behavior and drive our program synthesis.

2.1.2 Match. Synthesized implementations of library functions are used in two distinct ways. First, we inline synthesized implementations of the old API. This allows all or part of the old API to be matched to calls to the new API. Secondly, we use a synthesized version of the new API as a basis for matching. We generate a set of constraints that describe the synthesized LLVM program. These are then used to search the inlined program for matches.

2.1.3 Migrate. Once matches are found, we verify whether or not they are correct. First, we check by testing the match with automatically generated inputs to eliminate false positives. We then ask the user to confirm that the match is correct. Then, we can migrate by replacing code with the call to the new API.

2.1.4 LLVM. In this paper we use the LLVM compiler framework [21] as the means of representing and reasoning about programs. Programs in LLVM IR are written in single static assignment (SSA) form, which means that every variable is assigned a value exactly once and never reassigned. Control flow is limited to linear sequences of instructions (basic blocks) with branches between them. When the value of a variable is control-flow dependent, ϕ nodes can be used to select different values depending on the preceding basic block. The result of compiling the small C program in Figure 4

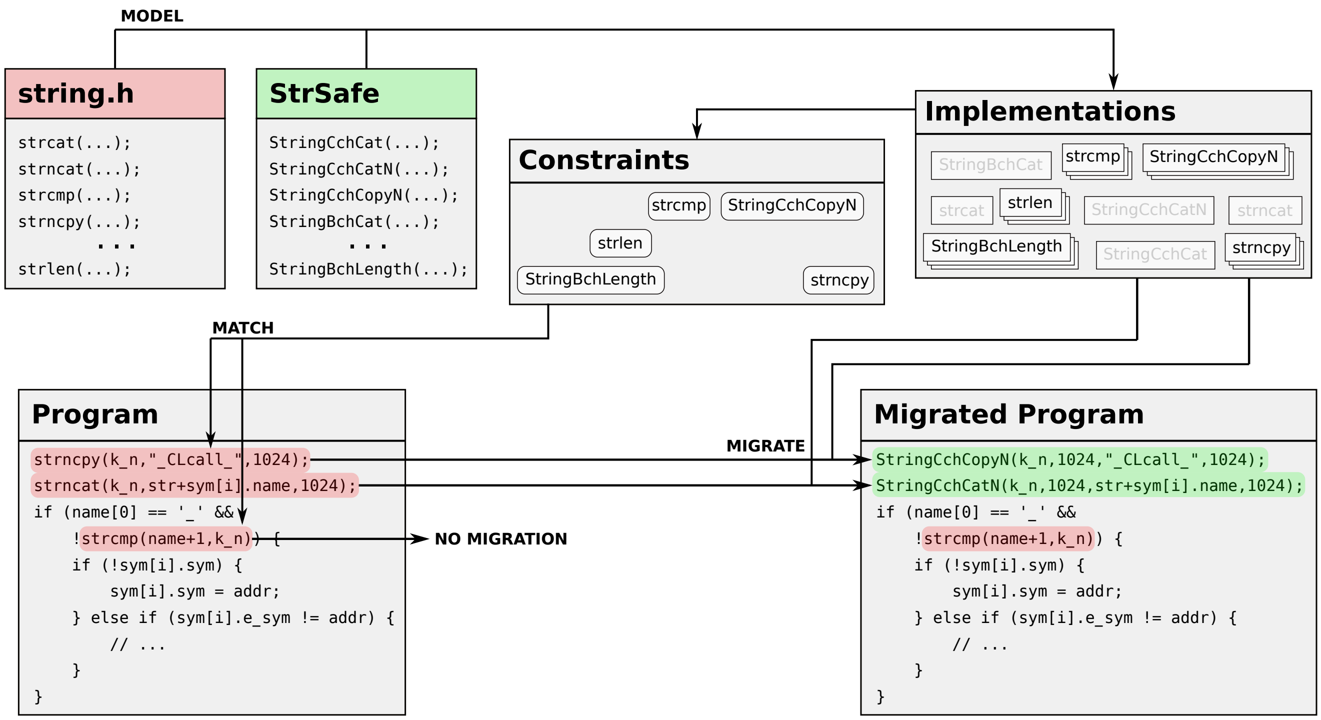


Figure 3: Worked example showing how string processing functions can be migrated used M^3. Implementations for the two libraries are synthesized where possible, converted to constraints, then used to discover matches. Finally, the discovered matches are used to migrate between libraries, where a migration is possible.

to LLVM is shown on the left of Figure 2. As well as this textual representation, the SSA form permits a graph representation, shown on the left. M^3 works on this graph-semantic representation rather than a token or syntactic level [40, 43].

```c
int64_t sum(int64_t *array, int64_t N) {
  int64_t sum = 0;
  for(int64_t i = 0; i < N; ++i)
    sum += array[i];
  return sum;
}
```

Figure 4: C function to sum the elements of an array.

2.2 Example

As a motivating example, consider the program on the left of Figure 3. String functions such as `strcpy` and `strncpy` are often misused by developers, resulting in potential buffer overflows. There are a number of libraries available that perform the same operations in a safer way (e.g. StrSafe [20]). Using this library is a valuable transformation to make to user code; this section describes how M^3 can achieve this migration.

The first steps in M^3 occur ahead of time, before any user code is compiled. At the top of Figure 3 we see two libraries containing functions and their API signatures. The compiler author uses the synthesis tool (**Model**) to produce inlinable implementations of library functions (**Implementation**). In this case, it successfully synthesizes (among others) LLVM implementations of the functions `strncpy` and `StringCchCopyN`.

We then translate these synthesized programs into **Constraints**, which are used to search the user's **Program** for matches in two separate stages. First, we detect references to old uses of `string.h`. If there is a corresponding function in the new library `StrSafe`, we inline for later migration. If there is no corresponding function, we do not modify the program as is the case for `strcmp`. Second, we search the modified code for matches with the new library. When we find a match (i.e. `strncpy` and `strncat`), we replace them with calls to the new library. These are checked for validity by testing the old and new code using input-output examples. If these give the same behavior, the user is asked to confirm to give the **Migrated Program**.

This is an example of how M^3 is able to achieve a useful API migration, allowing user code to use safer variants of known-unsafe string functions. However, our approach is general and can be used to perform migrations on large applications across a wide range of problem domains.

3 MODEL

This section gives an overview of Model, the first stage of our approach. Figure 5 shows a step-by-step illustration of our process, the details of which are given in this section.

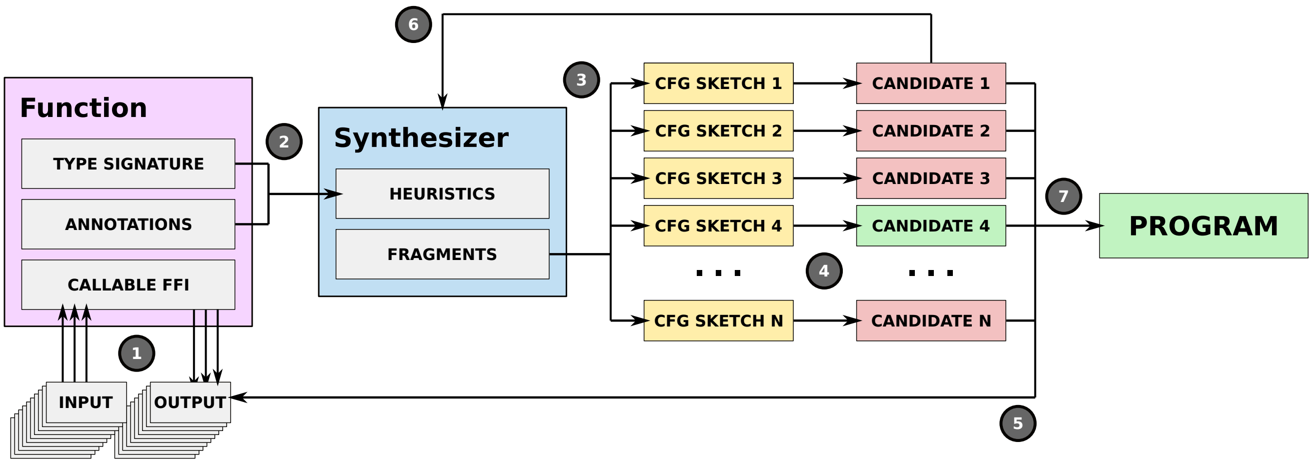


Figure 5: A summary of Model, the program synthesis component of M^3. Shown is the full synthesis process, beginning with the generation of input-output examples, then instantiating control-flow structures based on synthesis heuristics, and finally instantiating these into full programs which are checked for correctness.

3.1 Program Synthesis

The task of automatically generating a correct program given a specification is a widely-studied research problem. Many approaches exist that specify correctness differently or target distinct program representations.

Model falls broadly into the area of *programming by example*. A synthesis problem is stated by providing a set of input-output pairs; a correct solution to the problem is one that provides the correct output for each input.

Additionally, we use *component-based sketching* program synthesis. This approach builds up partial structure from smaller components. These structures are instantiated with instructions to produce solutions. The synthesis process amounts to two different search problems: for the correct structure, and then for the correct instantiation of that structure to yield a solution.

3.2 Specification

In Figure 5, **Function** shows the specification for our synthesis problem. We use three pieces of information about a library function: its type signature, a way to make calls to it, and some optional additional annotations that can be used to optimize the search process.

Examples. At stage ①, we generate many instances of input data. We pass each of these to the function and collect the resulting output value. The collected examples are used to check whether or not a candidate solution is correct. We operate under the assumption that a large number of generated examples capture the behavior completely, and have found this to be true for all the functions we synthesize.

Annotations. As well as the type signature, we permit additional specification of each function to be made in the form of annotations. These annotations express semantic knowledge about a function's interface that a programmer would know in order to use the function correctly (e.g. the relationship between a pointer variable and its size). These annotations are optional but serve to optimize the synthesis process. They can be readily obtained from library documentation, and do not require knowledge of internal behavior.

3.3 Sketches

Our program synthesis comprises two distinct phases. The first of these is to assemble a set of potential program structures that may represent a solution. We do this by sampling from a set of abstract components, then composing the sampled components together. At stage ③, the synthesizer has generated a number of potential control flow sketches by doing this.

Components. The synthesizer implements a collection of parameterized, compositional recipes for constructing regions of LLVM code. Composing a collection of these components together results in a partial program structure. These structures do not perform any computation; they must be further instantiated with instructions to do so. An example of a simple component is a loop, parameterized on the loop upper bound and an array to iterate over. Some components represent concrete control flow structures, while others represent more abstract ideas such as modifying the compilation of child components.

3.4 Heuristics

The synthesizer populates an initial set of program components by matching rule-based heuristics against the type signature and optional annotations of a library function (stage ②) using a simple logic programming engine. Each heuristic specifies when a particular component should be added to the initial set (e.g. considering a loop when the size of a pointer is known). A listing of the most important heuristics used by the synthesizer in this paper is given in Figure 6.

```
size(X, N) and
type(N, Int)
  => loop(X, N)
```

(a) Iteration with known size.

```
size(X, N) and
size(Y, N)
and type(N, Int)
  => zipLoop(X, Y, N)
```

(b) Joint iteration with known size.

```
output(X) and
not size(X, _)
  => singleStore(X)
```

(c) Store to scalar output.

```
output(X) and
size(X, N)
  => outputLoop(X, N)
```

(d) Store to array elements.

```
isPointer(X) and
not size(X, _)
  => affineAccess(X)
```

(e) Affine access to array.

```
type(X, Char) and
isPointer(X)
  => delimiterLoop(0, X)
```

(f) Delimited iteration.

```
enum(P, C0, ...) and
type(P, T)
  => switch(T, P, C0, ...)
```

(g) Enumeration switch-case.

```
isPointer(X) and
isPointer(Y)
  => fuse(X, Y)
```

(h) Attempt to fuse loops.

Figure 6: Synthesis heuristics used in Model. These rules specify combinations of properties that suggest a particular control flow structure should be included in candidate programs. When no annotations are used, the rules fall back to only checking types.

3.5 Search

Once a set of initial program components has been assembled using the synthesis heuristics, a search process begins to generate candidate programs. Starting from the set of possible program components, a set of possible compositions is sampled (stage ③ in Figure 5). Each of these represents a potential structure for a solution to take.

Instructions. To create programs that perform meaningful computation, we sample LLVM instructions and add them to the structure (at points defined by the idividual components. This produces a candidate program from a program structure (stage ④); these stages are iterated until a solution is found (stage ⑥).

3.6 Correctness

For each candidate program generated during the synthesis process, the final step of the is to decide whether or not it is correct (stage ⑤). We do this by using the IO examples generated at the start of synthesis; if the candidate program and the library function produce identical output for each input, then the candidate is correct and returned to the user (stage ⑦).

4 MATCH

In this section, we explain how we detect user code that matches synthesized implementations. To enable this, we translate the LLVM IR into a description suitable for automatic solvers; we use CAnDL [16], an constraint language to describe LLVM code structure. Then, we use an SMT solver to search for matching code.

4.1 Translating LLVM to Constraints

Our goal is to search the user's program to find code which matches functions synthesized from the new API. To achieve robust matching, we use a specialized existing constraint solver based on the CAnDL language.

Figure 7 shows how CAnDL describes an LLVM program. We can see that 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, and 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 code satisfying the constraints.

Our constraint descriptions are built from the data flow graph representation shown in Figure 2, 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).

This approach results in a constraint program that searches for exact sub-graph matches in user code, but is often too specific. Therefore, we use post-processing to turn this baseline constraint program into a more powerful matching engine.

4.2 Post-Processing Constraints

In order to allow for more effective detection of matches in user programs, the generated constraint programs are enhanced with a post-processing stage. The aim is to generate a more robust matching; this effectively requires a careful weakening of the constraints. 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

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

(a) Fragment of LLVM code extracted from the function in Figure 2.

```
Constraint Generated
( opcode{iter}     = phi
& opcode{sum}      = phi
& opcode{addr}     = gep
& opcode{elem}     = load
& opcode{nsum}   = add
& opcode{niter} = add
& ir_type{0}       = literal
& ir_type{1}       = literal
& ir_type{array}   = argument
& {niter}        = {iter}.arg[0]
& {0}              = {iter}.arg[1]
& {nsum}         = {sum}.arg[0]
& {0}              = {sum}.arg[1]
& {array}          = {addr}.arg[0]
& {iter}           = {addr}.arg[1]
& {addr}           = {elem}.arg[0]
& {elem}           = {new_sum}.arg[0]
& {sum}            = {new_sum}.arg[1]
& {iter}           = {new_iter}.arg[0]
& {1}              = {new_iter}.arg[1])
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 7: LLVM code sample and its corresponding CAnDL constraints, as generated by Match.

between permuted versions. Finally, we remove instructions that correspond only to compiler-specific code generation idioms.

5 MIGRATE

The final step in our approach is to perform API migration given a set of matched function instances in application code.

Available Migrations. Possible migrations are identified by using the same constraint search process as Match, but applied to application code where the original library calls have been fully inlined. This means that the constraints for similar functions in other libraries will match the inlined code; when this is the case we suggest a possible migration to the user.

Testing. We use an input-output based testing approach similar to that used in the synthesis process to validate suggested migrations. For each potential migration, we test the original code with randomly-generated IO examples, then compare the results to the new function. If they differ, then the migration is not valid and is discarded. This defends against potential false positive matches resulting from our constraint search process.

Algorithm 1 Emit Simple Constraint Description

```
function EmitConstraints(V,E)
    emit("Constraint Generated (")
    first ← true
    for v in V do
        if first then
            first ← false
        else
            emit("&")
        end if
        if op(v) = parameter then
            emit("ir_type", name(v), " = argument")
        else if op(v) = const then
            emit("ir_type", name(v), " = literal")
        else
            emit("opcode", name(v), " = ", op(v))
        end if
    end for
    for n, a, b in E do
        emit(name(a), " = ", name(b), ".args[", n, "]")
    end for
    emit(") End")
end function
```

6 EXPERIMENTAL DESIGN

To evaluate the success of the three components of M^3 with respect to the problem of API migration, we identify three research questions:

- **(RQ1)** Can program synthesis be used effectively to learn the behavior of black-box library functions?
- **(RQ2)** Given synthesized implementations for library functions, can similar instances in application code be accurately discovered?
- **(RQ3)** Given instances of user code that match the constraints generated from a library function, can API migrations be correctly implemented?

6.1 Evaluation Corpora

Applications. We selected 9 widely-used applications to evaluate our approach against; they are listed in Table 1. Each application is written in C or C++, and together they cover a wide range of problem domains: `ffmpeg` is a general-purpose video processing toolkit; `xrdp` is an open-source remote desktop protocol; `gems` is a reference collection of graphics algorithms; `etr` is a 3D game; `darknet` [32] and `caffepresso` [18] are deep learning frameworks; `Nanvix` [30] is a lightweight operating system; `androidfs` is a module from the Android filesystem; and `coreutils` are standard Unix utilities.

Libraries. We selected 7 libraries to target for migration (see Table 2 for details).

Table 1: Application source code used for evaluation.

Software	Description	LoC
`ffmpeg`	Media processing	1,061,655
`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

Table 2: Library APIs used for evaluation.

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

6.2 (RQ1)

To evaluate the effectiveness of program synthesis as a way of learning library programs, we considered several different metrics:
Coverage: We define the synthesis coverage of a library to be the proportion of that library's functions to be correctly synthesized.
Correctness: We manually checked whether the synthesiszed functions are in fact equivalent to the relevant library function.
Synthesis Difficulty We measured the mean number of candidates that are evaluated before a correct solution is found. We also record the number of lines of code for each function and the number of annotations used.

We prepared each library's functions for the synthesizer by collecting the type signatures together with references to locations in a shared library callable by the synthesizer. We applied property annotations where they could be clearly extracted from documentation. and synthesized 5 implementations of each library function. In cases where the synthesizer ran for longer than 1 day, we recorded a failure to synthesize that function.

6.3 (RQ2)

To evaluate the succcess of Match discovering instances in user code we used the following metrics:
Instances Discovered The number of locations in user code that match the generated constraints for a particular function; this metric is a good estimate for the possible number of migrations in a code base.
Precision and Sensitivity Even if a large number of instances are discovered, an inaccurate matching process is likely to lead to errors in the migrated user code; we therefore measure the precision and sensitivity of the matching.

We evaluated these metrics as follows: Synthesized functions were converted to constraint descriptions, which were then searched for in each of the applications in Table 1. We recorded the total number of reported matches for each function, then examined the application code by hand in order to discover possible matches not identified by the generated constraints, and examined reported matches to verify that they were in fact correct. Errors were recorded as false negatives or false positives respectively.

6.4 (RQ3)

To evaluate Migrate, we examined the number of matches from (RQ2) that can be correctly migrated to a different library. Our metric for this is the number of migrations, organised by original and replacement library.

We measured this by inlining the synthesized definition for each library function wherever a correct match had been identified. Then, we re-ran the constraint search for the other libraries. Potential migrations were tested, and the number of correct migrations after testing reported.

7 RESULTS

7.1 (RQ1) Results

Coverage. Table 3 shows the proportion of each library's API we were able to synthesize correctly. As expected, we could not synthesize every function from each library; some functions performed control flow not expressible using our library of heuristics, and some had type signatures incompatible with the synthesizer. We show an adjusted coverage metric that does not consider functions whose type signatures could not be expressed by the synthesizer.

Of the functions with expressible type signatures, we were able to synthesize implementations for nearly 40%. This represents a significant proportion of each library's behavior—in our worst performing case (BLAS), we are able to synthesize nearly 20% of all functions in the library.

The cases with expressible type signatures that could not be synthesized can be split into three groups: those limited by control flow idioms, data flow instruction sampling and synthesis complexity respectively. An example of each is:

Control Flow Our terminated-loop components could not express the loops needed to implement string concatenation (e.g. in `strcat`).

Instruction Sampling Algorithms such as lexicographic comparison of vectors were not synthesized as we did not sample boolean instructions in data flow.

Complexity Functions such as `strsm` simply have too complex a structure for our implementation, with many nested conditional tests and loops.

Despite these limitations, the collection of control flow structures and instruction sampling methods used in Model were able to synthesize a significant proportion of the functions considered.

Table 3: Synthesis coverage and adjusted coverage results for each evaluated library API.

Library	Functions	Coverage	Adj. Coverage
string.h	22	27.3%	37.5%
StrSafe	20	30.0%	60.0%
glm	61	26.2%	36.4%
mathfu	37	29.7%	44.0%
BLAS	31	19.4%	19.4%
Ti DSP	23	34.8%	34.8%
ARM DSP	38	31.6%	31.6%
Total	**232**	**28.4%**	**37.7%**

Program synthesis over an entire library API is a challenging problem; the programs that we were able to synthesize compare favorably in terms of complexity and synthesis time to other recent work in synthesis [34].

Correctness and Realism. Of the functions we were able to synthesize, all but one yielded solutions judged by a human expert to be correct. The incorrect implementation was the `memmove` function from `string.h`; in the presence of aliased arguments, the implementations produced incorrect behavior. Our testing framework did not provide a method for supplying aliased arguments to candidate functions; doing so is interesting future work.

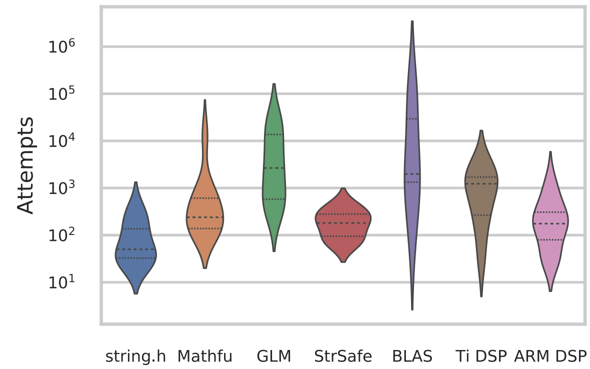
While otherwise all the functions were behaviorally correct, we identified that in some cases, the control-flow structure synthesized was unintuitive. For example, we found a solution to the vector addition function in the ARM DSP library that contained multiple unnecessary nested loops. The presence of unintuitive synthesized solutions did not affect our subsequent results.

Difficulty. Figure 8a 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 using fewer than 10^5 candidates.

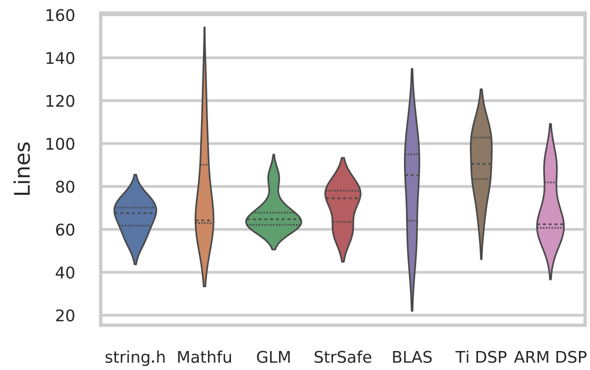
We were able to evaluate around 1,000 candidates per second on a desktop-class machine, meaning that almost every successful synthesis could be achieved in under 2 minutes. Only two functions from the `BLAS` library took more than 10^7 seconds to synthesize, leading to a long-tailed distibution. These 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.

The distributions in Figure 8b show how many lines of LLVM IR code were synthesized for correct solutions in each library. Most synthesized functions used between 40–100 lines, with some longer functions. We found in some cases that the synthesized code was shorter than the original library source; a side effect of our synthesis process is distilling out algorithmic intent from implementation details.

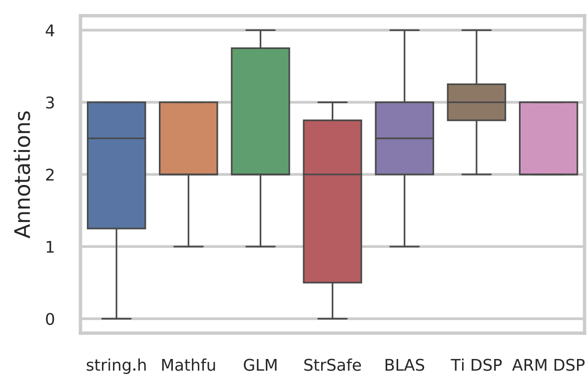
Use of Annotations. Annotations could all be readily obtained from library documentation; other work report easily extracting more complex properties about pointer aliasing using the same approach [22]. We show the number of annotations used for each library's



(a) Distribution of the mean number of candidate programs evaluated before a correct solution is found for each library API.



(b) Distribution of the number of lines of code present in synthesized solutions for each library API.



(c) The number of property annotations required to synthesize functions from each library.

Figure 8: Results for Model, showing distributions over each library API targeted.

Table 4: Match results, showing the number of function instances discovered in application code for each library, where N_C is instances discovered as handwritten code, and N_L is instances discovered as existing library calls. Additionally, the precision and sensitivity achieved when discovering functions from each library are reported.

Application	Library	N_C	N_L	Prec.	Sens.
ffmpeg	string.h	0	4,976	100.0%	100.0%
	GLM	7	0	100.0%	87.5%
	Mathfu	3	0	100.0%	100.0%
	BLAS	3	0	100.0%	100.0%
	ARM DSP	7	0	100.0%	87.5%
	Ti DSP	4	0	100.0%	100.0%
xrdp	string.h	0	686	100.0%	100.0%
coreutils	string.h	10	623	100.0%	100.0%
	StrSafe	6	0	100.0%	100.0%
gems	string.h	0	46	100.0%	100.0%
	GLM	13	0	100.0%	100.0%
	Mathfu	22	0	100.0%	88.0%
	ARM DSP	20	0	100.0%	87.0%
	Ti DSP	6	0	100.0%	100.0%
darknet	string.h	0	123	100.0%	100.0%
	BLAS	5	5	100.0%	100.0%
	GLM	1	0	100.0%	100.0%
	Mathfu	2	0	100.0%	100.0%
	ARM DSP	3	0	100.0%	100.0%
	Ti DSP	2	0	100.0%	100.0%
caffepresso	string.h	0	180	100.0%	100.0%
	Ti DSP	0	9	100.0%	100.0%
nanvix	string.h	10	0	100.0%	100.0%
	StrSafe	6	0	100.0%	100.0%
etr	string.h	0	4	100.0%	100.0%
	GLM	8	0	100.0%	100.0%
	Mathfu	21	0	100.0%	91.3%
	ARM DSP	11	0	100.0%	84.6%
	Ti DSP	5	0	100.0%	71.4%
androidfs	string.h	1	0	100.0%	100.0%
	StrSafe	1	0	100.0%	100.0%
Total		**177**	**6,648**		

functions in Figure 8c. No function used more than 4, and some used none at all. Comparing to Figure 8a, we can also see a rough correlation between synthesis difficulty and the number of annotations used: more complex functions have more semantic properties to annotate.

7.2 (RQ2) Results

Table 4 summarises our results from using Match to discover instances of synthesized functions in application code. The number of discovered instances is split into two parts: those discovered by matching against handwritten application code (N_C), and those discovered by identifying existing library calls (N_L). We observe that $N_C << N_L$; this is because application code is typically factored to avoid repetition. The number of call sites using this code is much greater in each case.

Accuracy. Our constraint generation and subsequent matching is very accurate. Testing matched regions of code for correctness using input-output examples eliminates any potential false positives; we did not suffer any loss of precision using this approach. However, in a small number of cases we suffered from false negatives. This occurred in cases where application code was hand-optimized heavily enough that the constraint solver was unable to understand it. This is a limitation of any IR-based search and is not specific to our method.

In total, we discovered 177 instances of hand-written application code that matched the semantics of a library function, as well as 6,648 cases where user code called a library function we were able to synthesize with Model. These results provide a large corpus of user code with known semantics that can be used to drive API migration.

7.3 (RQ3) Results

The results we obtained by using our Migrate methodology on the matched code results in Table 4 are shown in Table 5. From the 6,825 instances of matched code in applications, we were able to generate 1,798 valid migration opportunities. This is because not every match result corresponds to a migration. For example, the `memcpy` function can be easily synthesized and discovered, but no migration to a safer implementation is needed.

We found that the migrations reported at this stage were accurate; testing both the results from Match and the potential migrations with input-output examples meant that migrations were correct with a high degree of confidence.

By using Migrate, we were able to generate a number of different classes of migration: in 7 of the 9 applications evaluated, we discovered instances where handwritten code could be migrated to an existing library. Additionally, in every application we also identified cases where one uses of one library can be migrated to another.

Summary. Our results from using M^3 are positive. We achieve nearly 40% coverage across complex real-world library APIs and programs can be synthesized in minutes in most cases. Using constraint generation, we are able to discover over 6,000 instances of these functions in user code with high precision and sensitivity. These matched instances lead to over 1,700 correct migrations, either from handwritten code to a library, or from library to library.

8 RELATED WORK

Library Migration. (Semi-)automatic rewriting of application code to use new libraries has been well studied, particularly for Java and other object-oriented languages [41, 44]. In [33], migration techniques are partitioned into 3 sub-areas: library upgrade [14, 42], API evolution [11, 35] and library migration [25, 40, 45].

Most schemes rely on a large corpus of programs using the old and new libraries, frequently focusing on change logs [39]. In [2] each commit is examined to determine when one library is added

Table 5: Migrations discovered in user code, grouped by their source and target library. Additionally, the total number of migrations for each user application is given.

Application	Migration			Instances
ffmpeg	string.h	→	StrSafe	629
	internal	→	GLM	7
		→	Mathfu	3
		→	BLAS	3
		→	ARM DSP	7
		→	Ti DSP	4
xrdp	string.h	→	StrSafe	269
coreutils	string.h	→	StrSafe	633
	internal	→	string.h	10
		→	StrSafe	6
gems	string.h	→	StrSafe	46
	internal	→	GLM	13
		→	Mathfu	22
		→	ARM DSP	20
		→	Ti DSP	6
darknet	internal	→	BLAS	5
		→	ARM DSP	3
		→	Ti DSP	2
		→	Mathfu	2
	BLAS	→	ARM DSP	3
		→	Ti DSP	2
		→	Mathfu	2
caffepresso	Ti DSP	→	ARM DSP	9
		→	GLM	6
		→	Mathfu	3
		→	BLAS	6
nanvix	internal	→	string.h	10
		→	StrSafe	6
etr	string.h	→	StrSafe	4
	internal	→	GLM	8
		→	Mathfu	21
		→	ARM DSP	20
		→	Ti DSP	6
androidfs	internal	→	string.h	1
		→	StrSafe	1
Total				**1,798**

or removed; similar functions are automatically determined and are reported as potentially related candidates for migration.

The ongoing need for commit examples is highlighted in [3]. This paper provides a clean set of commit examples from a program corpus. It automatically finds the code change segments which are a basis for migration techniques. In both [26, 31], migration uses a NLP inspired approach based on a syntactic view of programs and builds a Word2Vec [24] model that can migrate a Java program to C# given an initial parallel mapping of APIs. More recently, [43] goes beyond simple replacement of library API calls and uses syntactic program differencing and program dependency analysis. It targets actual edits and replacements rather than just making suggestions. Although it is a syntactic rather than semantic approach, it is able to add new code to help migration of libraries.

Closer to our aim of not relying on commit logs is [5]. It uses GANs to generate the commit seeds rather than using human knowledge to do so. 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 refactor. Other work uses specific semantic knowledge of functions to perform refactoring with semantic guarantees [36].

Program Synthesis. Program synthesis has long been investigated in both the programming language and machine learning communities. In [38], synthesis is used to verify that an implementation meets a specification. Other work has looked at completing sketches [37] of programs to provide programmer abstraction and auto-parallelization [15]. Type signatures and information are often used to direct program synthesis, most commonly for functional programs [27, 28]. In [7] extended type information is suggested as a means of improving program synthesis, while in [6] a similar approach is used as a means of accessing heterogeneous accelerators for scientific applications. Our work considers a much wider, more diverse class of libraries and applications.

The machine learning community has long studied programming by example and input-output based program synthesis. Recent work has examined both induction with a learned latent version of the program and generation which uses a language model to generate programs [1, 29]. Such learned language models have been used for compiler-based benchmark generation [10], prediction of optimizations [9] and compiler fuzzing [8].

Constraint Analysis. Constraints in compilers are frequently deployed for program analysis [12]. They allow complex detection to be encapsulated in a high level description, and exploit advances made in SMT solvers. In [6, 17], constraints are used to detect areas of code that could be translated into libraries or DSLs such as Halide to give improved performance. A similar objective is considered in [23], which uses program lifting to detect and verify that code can be translated to Halide.

9 CONCLUSION

In this paper we have tackled the challenging task of *black-box* library migration, without access to the source code or semantics of either library. We investigated whether program synthesis can model the source and target libraries, and how constraints derived from synthesized programs can be used for library migration.

M^3 was successfully applied to a large corpus of existing imperative code, synthesizing 62 different functions across a range of problem domains, and using these implementations to discover over 1,700 library migrations. One interesting side-effect identified is the ability to automatically refactor library-free code to use new libraries.

Future work will investigate more powerful synthesis techniques removing the need for type information. The synthesized programs could also be used as a corpus generation methodology for alternative changelog-based approaches.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[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] 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*. IBM Corp., 60–71.

[4] Ittai Balaban, Frank Tip, and Robert Fuhrer. 2005. Refactoring support for class library migration. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 265–279.

[5] 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

[6] Bruce Collie, Philip Ginsbach, and Michael F. P. O'Boyle. 2019. Type-Directed Program Synthesis and Constraint Generation for Library Portability. (Aug. 2019). https://arxiv.org/abs/1908.04546

[7] Bruce Collie and Michael O'Boyle. 2019. Augmenting Type Signatures for Program Synthesis. *arXiv:1907.05649 [cs]* (July 2019). arXiv:cs/1907.05649

[8] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing Through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 95–105. https://doi.org/10.1145/3213846.3213848

[9] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 219–232. https://doi.org/10.1109/PACT.2017.24

[10] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 86–99.

[11] 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

[12] Steven Dawson, Coimbatore R Ramakrishnan, and David S Warren. 1996. Practical program analysis using general purpose logic programming systemsâĂŤa case study. In *ACM SIGPLAN Notices*, Vol. 31. ACM, 117–126.

[13] Jens Dietrich, Kamil Jezek, and Premek Brada. 2014. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 64–73.

[14] 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

[15] 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 (PLDI 2017)*. ACM, New York, NY, USA, 572–585. https://doi.org/10.1145/3062341.3062382

[16] 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

[17] 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

[18] G. Hegde, Siddhartha, N. Ramasamy, and N. Kapre. 2016. CaffePresso: An Optimized Library for Deep Learning on Embedded Accelerator-Based Platforms. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. 1–10. https://doi.org/10.1145/2968455.2968511

[19] Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE, 274–283.

[20] jwmsft. [n.d.]. About Strsafe.h - Windows Applications. https://docs.microsoft.com/en-us/windows/win32/menurc/strsafe-ovw

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

[22] Dhruv C. Makwana and Neelakantan R. Krishnaswami. 2019. NumLin: Linear Types for Linear Algebra. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 14:1–14:25. https://doi.org/10.4230/LIPIcs.ECOOP.2019.14

[23] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. 2015. Helium: Lifting High-Performance Stencil Kernels from Stripped X86 Binaries to Halide DSL Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 391–402. https://doi.org/10.1145/2737924.2737974

[24] 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

[25] 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

[26] 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.

[27] 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

[28] 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

[29] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis. (Nov. 2016). https://arxiv.org/abs/1611.01855v1

[30] P. H. d M. M. Penna, M. B. Castro, H. C. d Freitas, J. Méhaut, and J. Caram. 2017. Using the Nanvix Operating System in Undergraduate Operating System Courses. In *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*. 193–198. https://doi.org/10.1109/SBESC.2017.33

[31] 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

[32] Joseph Redmon. 2013. Darknet: Open Source Neural Networks in C. (2013).

[33] 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

[34] Christopher D. Rosin. 2018. Stepping Stones to Inductive Synthesis of Low-Level Looping Programs. *arXiv:1811.10665 [cs]* (Nov. 2018). arXiv:cs/1811.10665 http://arxiv.org/abs/1811.10665

[35] 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

[36] 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

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

[38] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2010. From program verification to program synthesis. In *ACM Sigplan Notices*, Vol. 45. ACM, 313–326.

[39] 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

[40] 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

[41] 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.

[42] 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

[43] 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.

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

[45] 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