SOAR: A Synthesis Approach for Data Science API Refactoring

Anonymous Author(s)

***Abstract*—With the growth of the open-source data science community, both the number of data science libraries and the number of versions for the same library are increasing rapidly. To match the evolving APIs from those libraries, open-source organizations often have to exert manual effort to refactor the APIs used in the code base. Moreover, due to the abundance of similar open-source libraries, data scientists working on a certain application may have an abundance of libraries to choose, maintain and migrate between. The manual refactoring between APIs is a tedious and error-prone task. Although recent research efforts were made on performing automatic API refactoring between different languages, previous work relies on statistical learning with collected pairwise training data for the API matching and migration. Using large statistical data for refactoring is not ideal because such training data will not be available for a new library or a new version of the same library. We introduce Synthesis for Open-Source API Refactoring (SOAR), a novel technique that requires no training data to achieve API migration and refactoring. SOAR relies only on the documentation that is readily available at the release of the library to learn API representations and mapping between libraries. Using program synthesis, SOAR automatically computes the correct configuration of arguments to the APIs and any glue code that is required to invoke those APIs. SOAR also uses the error messages from the interpreter when running refactored code to generate logical constraints that can be used to prune the search space. Our empirical evaluation shows that SOAR can successfully refactor 80% of our benchmarks, corresponding to deep learning models with up to 44 layers, with an average run time of 97.23 seconds, and 90%of the benchmark set for data wrangling tasks with an average run time of 17.31 seconds.**

***Index Terms*—software maintenance, program translation, program synthesis**

I. Introduction

Modern software development makes heavy use of libraries, frameworks, and associated *application programming interfaces* (APIs). Libraries provide modular functionality intended for reuse, with prescribing a particular architecture [1], and their widespread use has important productivity advantages [2]. The API for a library defines the interface, or contract, between the (hidden) library implementation of a piece of library functionality, and its client component [3]. Good API selection and maintenance is a key component of modern software engineering [4].

Although ideally API selection and usage could be stable over the course of a software project's lifetime, there are many practical reasons that client code must update the way it uses a given API, or even which API/library it uses for a given set of functionality. Broadly, software may evolve because of a change in the code, the documentation, its properties, or the customer-experienced functionality [5]. The APIs used by the software can become invalid or inapplicable as the software evolves. APIs themselves may become deprecated or obsolete [6]. As a result, to maintain and optimize software that depends on APIs, developers often have to refactor APIs between different versions, or to another API (*i.e., API migration*) altogether.

API migration is a form of software refactoring, a critical software engineering activity that is largely performed manually [7] and is tedious and often error-prone [8]. Migration can be difficult even when migrating between two closely-related APIs that nominally provide the same functionality. For example, consider increasingly popular data science and deep learning libraries, such as TensorFlow [9], PyTorch [10], and Numpy [11]. Moving between two such libraries often requires significant manual labor as well as domain- and library-specific knowledge (we illustrate with an example in Section II); worse, APIs can change, and outdated historical knowledge can exacerbate these challenges.

Fortunately, many popular APIs possess key properties that can inform an automated approach to support migration or evolution. First, open-source APIs are often reasonably well-documented [12]. The quality, quantity, and structure of that documentation can vary widely [13], but as code intended to be called and reused by unrelated client applications, documentation is often key to successful API uptake [13]. Second, unsuccessful API methods often raise exceptions with informative error messages, that developers can use to access stack traces and information that can help them modify a program [14]. We observe that data science API error messages are particularly useful as these error messages often identify how the input data relates to the raised exception. Take for example "*Error in fit[5, 100]: subscript out of bounds*", which is an error message describing an index overflow. From the example error message, we know that either 5 or 100 is out of bounds for the input matrix. Third, although multiple APIs may vary in concrete implementation details, by virtue of solving the same general sets of problems, it is often possible to discretely map between pieces of functionality in each of a source and target API.

We propose SOAR (Synthesis for Open-source API Refactoring), a novel approach that combines natural language processing (NLP) with inductive program synthesis [15] to automatically migrate/refactor between APIs. We focus our approach and evaluation on deep learning and data science APIs. Since there are many APIs targeting these domains, changes and new releases are introduced rapidly (as one example, TensorFlow had 26 releases in 2019 alone), and

switching between them is common and often tricky [16]. Moreover, data scientists and other users of such libraries have broad backgrounds and are not always classically trained programmers, and thus could particularly benefit from tool support to assist them in these tasks [8]. However, we believe the approach will generalize to other APIs with similar properties (see detailed discussion in Section IV).

Given a program that uses a given source API, SOAR's central proposition is to use NLP models learned over available API documentation and error messages to inform program synthesis to replace all source API calls with corresponding calls taken from the target API. SOAR starts by using existing documentation for the source and target libraries to build an *API matching model*, which finds likely replacement calls for each API call in the source program.

However, simply finding the right function in the corresponding target API is not enough, since the new function must be called with the correct arguments, and function specifications may vary between libraries. SOAR uses *inductive program synthesis* to construct the full target method call in a way that replicates the original source behavior. This synthesis step may be further informed by specifications inferred, again, from the API documentation. During the program synthesis enumeration procedure, a potential migrated call may throw an error when tested. In these situations, SOAR uses an *error message understanding* model that again uses NLP techniques to analyze error messages and generate logical constraints to prune the search space of the synthesis task.

To the best of our knowledge, SOAR is the first refactoring tool that incorporates program synthesis and machine learning tools for refactoring, and is a significant improvement over the prior state of the art. SOAR maps programs between different APIs using only readily-available documentation. It does not require manual migration mappings [17] or a history of previous migrations or refactorings in other software projects [18]–[21]. Indeed, SOAR does not require training data at all, and is thus applicable for migrations to a new library or newer version of the same library shortly after release. We demonstrate that SOAR is versatile in Section IV, using it to migrate between two deep learning libraries (*i.e.,* TensorFlow to PyTorch) in the same programming language (*i.e.,* Python) and between two data manipulation libraries (*i.e.,* dplyr to pandas) in two different programming languages (*i.e.,* R and Python). Prior techniques either specialize exclusively in supporting cross-language migration (*e.g.,* StaMiner [19]), or do not support it at all. Because SOAR uses synthesis, when it succeeds the produced code is guaranteed to compile and pass existing test cases for the original source code.

In summary, our main contributions are:

1) We propose SOAR, a novel approach based on NLP and synthesis for automatic API refactoring, focusing on (but not limited to) deep learning and data science tasks.
2) SOAR requires no training data, and its output is guaranteed to compile and pass existing test cases. Instead of using training data from prior programs, SOAR leverages API documentation and program error messages to generate logical constraints to prune the program enumeration search space.
3) We evaluate SOAR on two library migration tasks (*i.e.,* TensorFlow to PyTorch and dplyr to pandas) to demonstrate its effectiveness. Our results show that SOAR can successfully migrate 80% of neural network programs composed by 3 to 44 layers in with an average time of 97.23 seconds. And for dplyr to pandas migration, 90% of benchmarks are solved on average in 17.31 seconds.
4) With ablation studies, we also evaluate how each part of SOAR impacts its performance. We show that the use of specifications from API documents and learning from error messages are largely helpful for the synthesis process. We also show how different API matching methods perform on the two migration tasks.
5) We release the SOAR implementation for the two migration tasks mentioned above. We also release the documentation and benchmark tests we use in this work to facilitate future research on this direction.

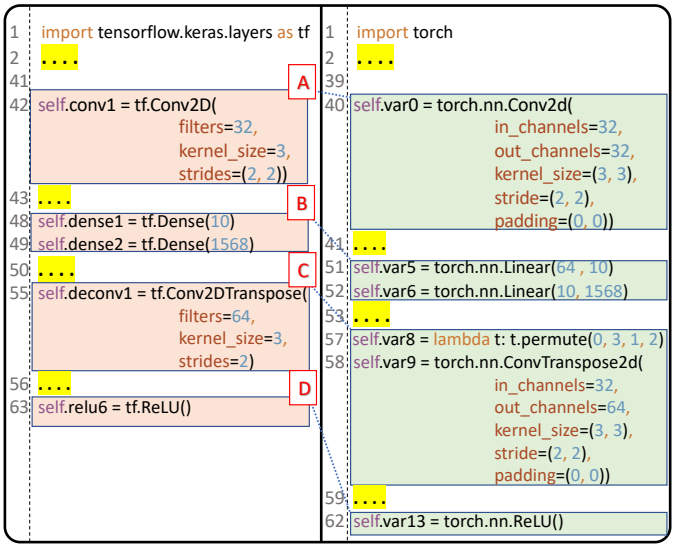

Fig. 1: An example of how SOAR refactors a program written with TensorFlow (left) to using PyTorch (right). Note that the whole program consists of 15 APIs calls to TensorFlow, though we only show four blocks of them (*i.e.,* A, B, C and D) for brevity. SOAR can migrate the full program in 161 seconds.

The remainder of this paper is organized as follows: Section II presents a motivating example that illustrates the challenges of manual API refactoring. In section III, we describe our approach to automatic API migration. Section IV presents our empirical evaluation and analysis of results. Next, we discuss our current approach and limitations in section V. Finally, we conclude with an overview of related work in section VI and conclusions in section VII.

II. Motivating Example

We illustrate some of the difficulties of manual API refactoring via example. Consider the TensorFlow code snippet on the left-hand-side of Figure 1. The program being refactored

shows an autoencoder program [22] written using the TensorFlow API; the goal is to migrate this code to use the PyTorch API. An autoencoder is a type of neural network that is trained to copy its input to its output. Specifically in this example, the autoencoder tries to compress an image with an encoder and then the decoder will try to restore the original image.

The example in Figure 1 shows only a portion of the program, for didactic purposes. To build the first layer of the encoder, function `Conv2D` is called, which constructs a convolution layer that can be applied to 2D images. After further (elided) activation and convolution layers, it calls `Dense` to output a latent representation of the input image. Decoding this output follows roughly the same procedure as the encoding, but using `Conv2DTranspose` instead of `Conv2D`. The function `ReLu` appears in both the encoder (not shown) and decoder, initializes a type of activation layer to ensure non-linearity of the neural network.

The example of deep learning library code and translation in Figure 1 illustrates several of the core challenges in refactoring open-source APIs, as well as opportunities to inform an automated approach. First, the names of function calls implementing similar functionality may be very similar or even identical (such as those in blocks A, C, and D), or completely different (*e.g.,* `Dense` versus `Linear` in block B). If a developer were performing this migration manually, they might reference the API documentation. For example, the TensorFlow documentation describes the `Conv2D` class as a "2D convolution layer (*e.g.,* spatial convolution over images)" [23]; the corresponding PyTorch documentation for the `Conv2d` call describes it similarly, as a "2D convolution over an input signal composed of several input planes" [24]. Here, the function names map well, but when this does not happen, it is more challenging to connect the documentation.

Even when we know which function to use, however, calls that implement the same functionality can require different types, parameters, parameter names, and even the parameter values may be different between them. This is true for the majority of the calls in our example (see those in blocks A, B, and C). Note for example that the `Conv2D` functions take different parameters in each of the two libraries. There is overlap between them — both include `kernel_size`, and `stride` and `strides` clearly correspond — but even the in-common parameters are not in the same argument position between the two calls (`strides` is the third parameter in TensorFlow but `stride` is the fourth in PyTorch). Sometimes, some or all of the arguments to a call in the source API can be copied directly to the call in the target API (see the calls in blocks A, C); other times, correct arguments must be inferred (such as the first parameter to `Linear` in block B). Finally, in other situations, no single function in the target API can match the semantics of a call from the source API, requiring instead a one-to-many mapping (as we see in converting the `Conv2DTranspose` call in block C).

In the next section, we show how SOAR addresses these challenges with each of its components.

Algorithm 1 SYNTHESIZER($\mathcal{I}, \mathcal{S}, \mathcal{T}, \mathcal{C}$)

Input: $\mathcal{I}$: existing program, $\mathcal{S}$: source library, $\mathcal{T}$: target library, $\mathcal{C}$: test cases
Output: $\mathcal{O}$: refactored program

```
r⃗ : API mapping = MAPAPI(T, S)
O = {}
for each l ∈ I do
    O = O ∪ REFACTORLINE(l, T, C, r⃗)
end for
```

III. REFACTORING ALGORITHM

This section describes SOAR, our approach for automatic API migration. We begin with a high-level overview of the method (Section III-A) before providing more detail on individual components (Section III-B; III-C; III-D).

A. Overview

Figure 2 shows an overview of the SOAR architecture, while Algorithm 1 provides an algorithmic view. SOAR takes as input a program $\mathcal{I}$ consisting of a sequence of API calls from a source library $\mathcal{S}$, the source ($\mathcal{S}$) and target ($\mathcal{T}$) libraries and their corresponding documention, and a set of existing test cases ($\mathcal{C}$). Since the user wants to refactor code from $\mathcal{S}$ to $\mathcal{T}$, we assume that the user already has test cases for $\mathcal{I}$ that can be reused to check if the refactored code ($\mathcal{O}$) has the same functional behavior has the original code ($\mathcal{I}$). Refactoring proceeds one line at a time in $\mathcal{I}$, finding/constructing an equivalent snippet of code (composed by one or more lines) that uses APIs of the target library $\mathcal{T}$; the composition of all these translated lines comprises the output $\mathcal{O}$.

For each API call in the input program, the first problem either a developer or a tool must face is to identify methods in the target API that implement the same functionality (*i.e.,* for a given set of input parameters, the target API call must generate the same output). SOAR uses an *API matching model* to identify target API calls. This model is built using NLP techniques that analyze the provided API documentation for each call, and provides a mapping ($\vec{r}$ in Algorithm 1) that computes the similarity between each target API function and each potential source API function. SOAR uses this to find the most likely replacement methods in the target API for each source API call in the input program. We provide additional detail in Section III-B.

Given a potential match call in the target API, the next step is to determine *how* to call it, in terms of providing the correct parameters, in the correct order, of the correct type. SOAR uses program synthesis to automatically write the refactored API call, using the provided test cases to define the expected behavior of the synthesized code and its constituent parts. The synthesis process can be assisted with additional automated analysis of API documentation, which often provides key information about each parameter, namely (1) whether it is required or optional, (2) its type, (3) its default value (if applicable), and (4) constraints between arguments,

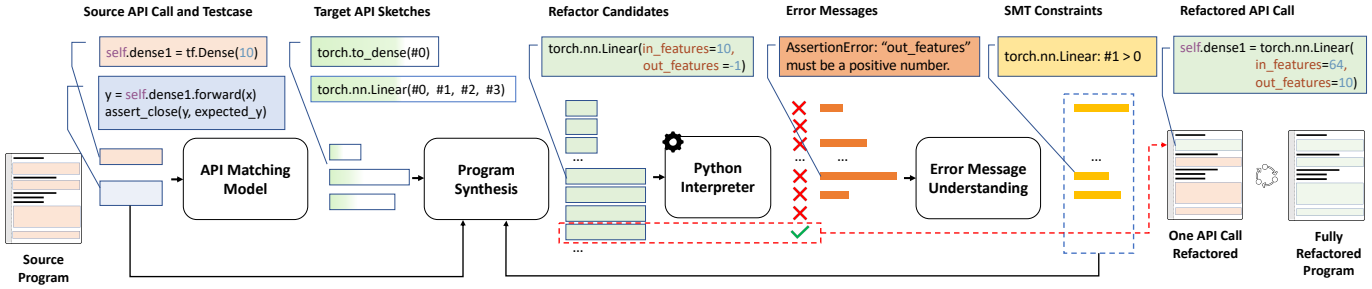

Fig. 2: Overview of SOAR's architecture.

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **padding_mode** (*string, optional*) – `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

Fig. 3: Description of the program parameters in `torch.nn.Conv2d` documentation [24].

input and output (*e.g.,* input and output tensor shapes). Figure 3 shows a snippet of the descriptions of all parameters for `torch.nn.Conv2d`. For example, the parameter `stride` is optional; it takes type `int` or `tuple`, and its default value is 1. Analysis of this documentation can produce a *specification constraint* for the `stride` parameter, assisting the program synthesis task. Section III-C describes the synthesis step.

Given a potential rewrite in the target API, a natural step for a developer would be to run the refactored code on test inputs. Unsuccessful runs can be quite informative, because many APIs (especially in the deep learning and data science domains) provide error messages that can be very helpful for debugging. SOAR simulates the manual debugging process by first adapting the input whole-program test cases to test partially refactored code, and then extracting both syntactic and semantic information from any error messages observed when running them. SOAR uses this information to add new constraints to the iterative synthesis process (Section III-D).

After migrating all calls in the source API to the target API such that all input tests pass, SOAR outputs a fully refactored program. Subsequent sections provide additional detail on the previously described steps.

B. API Representation Learning and Matching

The first step in migrating a call in a source API is to identify candidate replacement calls in the target API with similar semantics. The *API matching model* supports this task by analyzing the prose documentation associated with each call in each API, and computing similarity scores between all API pairs. At a high level, this model embeds each API method call in a source and target library into the same continuous high-dimensional space, and then computes similarity between two calls in terms of the distance between them in that space. We explored two ways on obtaining API representation: TF-IDF (term frequency – inverse document frequency) [25] and pretrained word embeddings [26].

TF-IDF. The intuition behind TF-IDF is to find the most *representative* words rather than the most frequent words in a sentence. Normalizing by the inverse-document-frequency lowers the weights of common keywords that are less informative, such as *torch*, *tensorflow* and those stop words in natural language such as *the* or *this*.

Specifically, we first derive a bag-of-words representation $\mathbf{x^i}$ from a description of an API call after some stemming of the words with the Snowball Stemmer [27]. $\mathbf{x^i} = [x_1^i, x_2^i, ..., x_n^i]$ where x_j^i denotes the frequency with which word x_j appeared in the sentence $\mathbf{x^i}$, and n is the size of the vocabulary from the descriptions of all APIs we are trying to embed. A TF-IDF representation of the call is computed as Equation 1:

$$\text{TF-IDF}(\mathbf{x^i}) = [\frac{x_1^i}{\sum_{t=0}^{m} x_1^t}, \frac{x_2^i}{\sum_{t=0}^{m} x_2^t}, ..., \frac{x_n^i}{\sum_{t=0}^{m} x_n^t}] \quad (1)$$

However, the major downside of TF-IDF is that it does not encode the similarities between words themselves. For example, consider two hypothetical call descriptions: (1) *Remove the last item of the collection*, and (2) *Delete one element from the end of the list*. They are semantically similar but since they have minimal overlapping words, a TF-IDF representation method would not recognize these two API calls as similar.

Tfidf-GloVe. We can fix this problem by adding the use of pretrained word embeddings. Specifically, we use the GloVe embedding [26], which is trained on a very large natural language corpus and learns to embed similar words closer in the embedding space.

To obtain sentence embeddings from individual words, we perform a weighted average of the word embeddings and use the TF-IDF scores of individual words as weight factors. It is a simple yet effective method to obtain sentence embedding for downstream tasks, as noted by previous work [28], [29]. This is shown in detail as Equation 2, where $\mathbf{w_j}$ is the vector encoding the GloVe embedding of word x_j:

$$\text{Embedding}(\mathbf{x^i}) = \sum_{i=j}^{n} \frac{x_j^i \cdot \mathbf{w_j}}{\sum_{t=0}^{m} x_j^t} \quad (2)$$

Algorithm 2 REFACTORLINE($l, \mathcal{T}, \mathcal{C}, \vec{r}$)

Input: l: line of code from $\mathcal{I}$, $\mathcal{T}$: target library, $\mathcal{C}$: test cases, $\vec{r}$: ranked list of API matchings
Output: $\mathcal{R}$: refactored snippet

```
1: O = {}
2: for each l' ∈ r⃗ do
3:     s⃗ = GENERATESKETCHES(l', T)
4:     for each s ∈ s⃗ do
5:         R = FILLSKETCH(s)
6:         if PASSTESTS(R, C) then
7:             return R
8:         end if
9:     end for
10: end for
```

By including the GloVe embedding, word similarity is preserved; by including the TF-IDF terms, the influence of embeddings of common words is greatly reduced. However, GloVe is trained with Common Crawl [30] which contains raw webpages, which is a mismatch from our domain of textual data (*i.e.,* data science and programming). This causes a lot of OOV (out-of-vocabulary) problems.

API matching. Given the representation of two APIs $\text{Rep}(\mathbf{x}^i)$, $\text{Rep}(\mathbf{x}^j)$ in the same space $\text{Rep}(\cdot)$, we compute their similarity with cosine distance:

$$\text{sim}(\text{Rep}(\mathbf{x}^i), \text{Rep}(\mathbf{x}^j)) = \frac{\text{Rep}(\mathbf{x}^i) \cdot \text{Rep}(\mathbf{x}^j)}{|\text{Rep}(\mathbf{x}^i)||\text{Rep}(\mathbf{x}^j)|} \quad (3)$$

For computational efficiency, we pre-compute the similarity matrix between the APIs across the source and target library. So we will be able to query the most similar API for the synthesizer to synthesize its parameters on the fly.

C. Program Synthesis

Given the input test cases and the API matching model providing a ranked list $\vec{r}$ of APIs in the target library, the synthesis model automatically constructs new, equivalent code, of one or more lines, that uses APIs of the target library $\mathcal{T}$. The refactored program $\mathcal{O}$ has the same functionality as input program $\mathcal{I}$, and passes the same set of tests $\mathcal{C}$.

To refactor each line of the existing program $\mathcal{I}$, we use techniques of programming-by-example (PBE) synthesis [31]. PBE is a common approach for program synthesis, where the synthesizer takes as specification a set of input-output examples and automatically finds a program that satisfies those examples. In the context of program refactoring, our examples correspond to the test cases for the existing code. In this paper, we restrict ourselves to straight-line code where each line returns an object that can be tested. With these assumptions, we can automatically generate new test cases for each line k of program $\mathcal{I}$. This can be done by using the input of the existing tests, running them, and using the output of line k as a new test case for the program composed by lines 1 to k.

Our program synthesizer for refactoring of APIs is presented in Algorithm 2 and it is based on two main ideas: (i) program sketching, and (ii) program enumeration. For each line l in program $\mathcal{I}$, we start by enumerating a program sketch (*i.e.,* program with holes) using APIs from the target library $\mathcal{T}$ (line 3). For each program sketch, we perform program enumeration on the possible completion of the API parameters (line 5). For each complete program, we run the test cases for the program up to line l. If all test cases succeed, then we found a correct mapping for line l between libraries $\mathcal{S}$ and $\mathcal{T}$ (line 6). Otherwise, we continue until we find a complete program that passes all test cases.

Program Sketching. Program sketching is a well-known technique for program synthesis [32] where the programmer provides a sketch of a program and the program synthesizer automatically fills the holes in this sketch such that it satisfies a given specification. We refactor *one line* of program $\mathcal{I}$ at each time. Our first step is to use the ranked list of APIs to create a program sketch where the parameters are unknown. For instance, consider the first layer from the motivating example that shows the network for an autoencoder using TensorFlow:

```
tf.keras.layers.Conv2D
(filters=32, kernel_size=3,strides=(2, 2))
```

A possible sketch for this call using PyTorch is:

```
torch.nn.Conv2d
(#1,#2,(#3,#4),stride=(#5,#6),padding=(#7,#8))
```

Where holes `#i` have to be filled with a specific value for the APIs to be equivalent. This approach works for *one-to-one* mappings but would not support common *one-to-many* mappings where the parameters often need to be transformed before being used in the new API. This is the case of the previous API where a reshaping operation must be performed before calling the PyTorch API. To support this common behavior, we include in our program sketch *one* API from the target library $\mathcal{T}$ and common reshaping APIs (*e.g.,* permute, long).

The sketch that corresponds to the refactoring solution of the `Conv2D` API from TensorFlow uses a reshaping API before calling the `Conv2d` API from PyTorch:

```
lambda x: x.permute(#9,#10,#11,#12)
torch.nn.Conv2d
(#1,#2,(#3,#4), stride=(#5,#6),padding=(#7,#8))
```

Using Occam's razor principle, our program synthesizer enumerates program sketches of size 1 and iteratively increases the size of the synthesized program up to a specified limit.

Program Enumeration. For each program sketch $\mathcal{P}$, our program synthesizer enumerates all possible completions for each hole. Since each hole has a given type, we only want to enumerate well-typed programs. We encode the enumeration of well-typed programs into a Satisfiability Modulo Theories (SMT) problem using a combination of Boolean logic and Linear Integer Arithmetic (LIA). This encoding is similar to other approaches that use SMT-based enumeration for program synthesis [33], [34] and encodes the following properties:

- Each hole contains exactly one parameter;
- Each hole only contains parameters of the correct type.

A satisfying assignment to the SMT formula can be translated into a complete program. The types for each hole can be

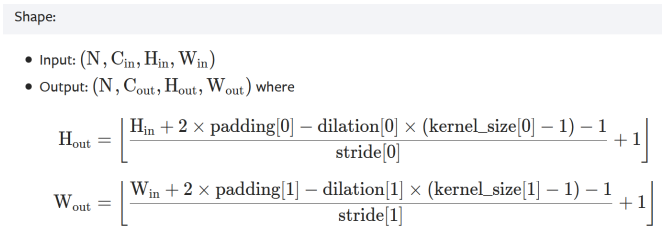

Fig. 4: Relationship between the parameters of `Conv2d` API described in PyTorch documentation [24].

determined by extracting this information from documentation, by performing static analysis, or by having this information manually annotated in the APIs. The available parameters and their respective types can be extracted automatically from the parameters used in the k-th line of program $\mathcal{I}$ and by any default parameters that can be used in the API from $\mathcal{T}$ that appears in the program sketch $\mathcal{P}$. For instance, for the `Conv2d` example presented in this section, we consider as possible values for the holes, the values that appear in the existing code (32, 3, 2) and default values for integer parameters (-1, 0, 1, 2, 3) that are automatically extracted from documentation.

Encoding the enumeration of well-typed programs in SMT has the advantage of making it easier to add additional logical constraints that can prune the search space.

Specification Constraints. As we described in Section III-A, API documentation often provides additional useful information about parameters to function calls, including type and default values. For each considered API call, we scrape/process the associated documentation to extract these properties and encode them as SMT constraints to further limit the synthesizer search space.

Additionally, some APIs have complex relationships between parameters which if encoded into SMT may reduce the search space considerably. For instance, Figure 4 shows the relationship between the different parameters for the `Conv2d` API described in PyTorch documentation. For APIs with these kinds of shape constraints, we can encode these relationships into SMT to further prune the number of feasible completions. When we use these relationships in our experiments, we encode them manually (a one-time cost for an actual SOAR user or API maintainer), but we observe that in many cases they could be automatically extracted from documentation.

Besides these specification constraints, we can also further prune the search space by using the error messages provided by the Python interpreter, as we discuss in the next section.

D. Error Message Understanding

We use a combination of extracting hyponymy relations and Word2vec [35] to understand run-time error messages. As outlined in Figure 5, our SMT constraint generation method consists of three steps.

Step 1: Extract hyponymy relation candidates from error messages. We perform an automatic extraction of customized hyponyms on each error message. Hyponyms are specific lexical relations that are expressed in well-known ways [36].

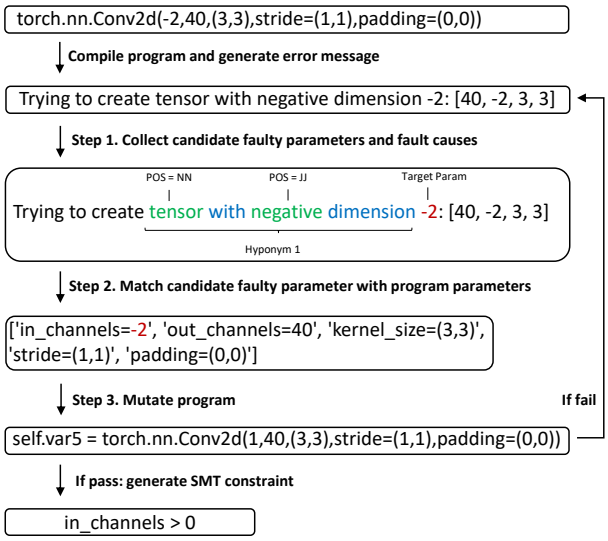

Fig. 5: Example error message to SMT constraint pipeline using hyponym 1.

In encoding a set of lexico-syntactic patterns that are easily recognizable (*i.e.,* hyponyms), we avoid the necessity for semantic extraction of a wide-range of error message text. We then use the collected hyponyms to map the error message to a single faulty parameter, and output a SMT constraint based on the faulty parameter.

Prior work on text parsing uses Tregex, which is a utility developed by Levy and Andrew for matching patterns in constituent trees [37]. For example, Evans *et al.* evaluated the performance of Tregex on privacy policies [38]. However, DL API compilation error messages are domain specific. Sumida *et al.* used the hierarchical layout of Wikipedia articles to identify hyponymy relations [39]. Similarly to Wikipedia documents, DL API compilation error messages are more consistent and organized than normal, natural language, documents. Therefore, we follow the approach of extracting hyponymy relations based on the hierarchical layout of a string.

We propose a set of four lexico-syntatic patterns to identify hyponyms using *noun-phrases* (NP) and regular expressions frequently appearing in machine learning API error messages. Table I shows the four hyponyms. If we identify any of the four lexico-syntatic patterns within an error message, we tag the error message with a hyponym type. As shown in Figure 5, we identify hyponym 1 in error message "*Trying to create tensor with negative dimension...*".

Step 2: Identify candidate faulty parameters and constraints. Step 2 uses different keywords based on the result of step 1 to identify the faulty parameter. As shown in Figure 5, an error message with hyponym 1 is likely to have the POS=JJ word as a parameter constraint (*i.e.,* word "*negative*"). Based on the fault cause candidate, we then store all negative numbers as candidate faulty parameters (*e.g.,* [40, -2, 3, 3] has -2 as the only faulty parameter). We then vectorize the candidate faulty parameter name (*i.e.,* -2) and find the program parameter name with the closest vectorized distance. As shown

TABLE I: The four hyponyms in the error message understanding model

Type	NP	Example error messages	Identified hyponym
1	$\{\langle Noun\rangle * \langle Preposition\rangle\langle Adjective\rangle?\langle Noun\rangle\}$	'Trying to create tensor with negative dimension -1: [-1, 100, -1, -1]'	tensor with negative dimension
2	$\{\langle Noun\rangle\langle Cardinal\_number\rangle\}$	'embedding(): argument weight (position 1) must be Tensor, not int'	position 1
3	$\{\langle Coordinating\_conjunction\rangle\langle Verb\rangle\langle Adjective\rangle\langle Noun\rangle\}$	'Expected 3-dimensional input for 3-dimensional weight [2, 2, 3], but got 4-dimensional input of size [100, 50, 40, 1] instead'	but got 4-dimensional input
4	$\{\langle Verb\rangle\langle Adverb\rangle\langle Verb\_past\_participle\rangle\}$	'non-positive stride is not supported'	is not supported

in Figure 5, the parameter "$in\_channels = -2$" has the nearest vectorized distance to the candidate faulty parameter -2. Based on the fault cause, we generate a candidate constraint. The example error message in Figure 5 has only one candidate constraint: "$in\_channels >= 0$".

Step 3: Mutate program. To validate the candidate faulty parameters and constraints, we mutate each faulty parameter according to each faulty parameter and constraints pair. We then re-compile the program for each mutation. If the error message remains the same, we discard the faulty parameter and constraint pair as a candidate. If the program passes, or if the error message changes, we store the faulty parameter and constraint pair as an SMT constraint. As shown in Figure 5, the API call mutator mutates the second parameter ("$in\_channels = -2$") to a non-negative number. The mutator first attempts "$in\_channels = 0$" and it encounters a different error message. From the new error message, we mutate this parameter to "$in\_channels = 1$" and observe no further errors. Therefore, we refine our previous constraint to be "$in\_channels > 0$", and store it as the final SMT constraint for the program in Figure 5.

IV. Evaluation

We evaluate our approach by answering the following research questions:

Q1. How effective is SOAR at migrating neural network programs between different libraries?
Q2. How does each component of SOAR impact its perfomance?
Q3. Is SOAR generalizable to domains besides deep learning library migration?

A. Benchmarks and experimental setup

We collected 20 benchmarks for each of the two migration tasks. In particular, for the TensorFlow to PyTorch task, we gathered 20 neural network programs from tensorflow tutorials [40], off-the-shelf models implemented with TensorFlow [41] or its model zoo [42]. This set of benchmarks includes: Autoencoders for image and textual data, classic feed-forward image classification networks (*i.e.,* the VGG family, AlexNet, LeNet, etc), convolutional network for text, among others. The average number of layers in our benchmark set is 11.80 ± 11.52, whereas the median is 8. Our largest benchmark is the VGG19 network which contains 44 layers.

For the domain of table transformations, we collected 20 benchmarks from Kaggle [43], a popular website for data science. The programs in the benchmark set have an average of 3.05 ± 1.07 lines of code, and a median of 3 lines. Although the programs considered for this task are relatively small compared to the deep learning benchmarks, they are still relevant for data wrangling tasks as shown by previous program synthesis approaches [44].

All results presented in this section were obtained using an Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz, with 64GB of RAM, running Debian GNU/Linux 10, and a time limit of 3600 seconds. To evaluate the impact of each component in SOAR, we run four versions of the tool. SOAR with TF-IDF (SOAR w/ TF-IDF) and SOAR with tfidf-GloVe (SOAR w/ Tfidf-GloVe) to evaluate the impact of API representation learning methods. SOAR without specification constraints (SOAR w/o Specs.) and SOAR without error message understanding (SOAR w/o Err. Msg.) to evaluate the impact of these components on the performance of SOAR.

B. Implementation

The SOAR implementation integrates several technologies. Scrapy [45], a Python web-scraping framework, is used to collect documentation for the four libraries in our experiments. To enumerate programs in the synthesis step, we use the Z3 SMT solver [46]. For each target program call parameter, we extract an answer for the four parameter questions in Section III-A and generate corresponding SMT constraints. In both API matching model and the error message understanding model, the GloVe word embeddings [26] are used as an off-the-shelf representation of words. For the four libraries appearing in our two evaluation migration tasks, we use TensorFlow 2.0.0, PyTorch 1.4.0, dplyr 1.0.1 (with R 4.0.0) and pandas 1.0.1, though our proposed method and associated implementation do not rely on specific versions. We provided anonymized code and data to support review[1] and we will provide a full, unblinded replication package post-review.

C. Q1: SOAR effectiveness

Table II shows how long it takes to migrate each of the deep learning models from TensorFlow to PyTorch, using the various approaches. Our best approach (shown as SOAR) successfully migrates 16 of the 20 DL models with a mean

[1]https://figshare.com/s/ec38b01e057bda4b8c78

TABLE II: Execution time for the deep learning library migration task in each of the 20 benchmarks.

	SOAR	SOAR w/o Specs.	SOAR w/o Err. Msg.
conv_pool_softmax(4L)	**1.60**	23.02	14.35
img_classifier(8L)	**12.82**	336.00	65.66
three_linear(3L)	3.18	**2.34**	21.07
embed_conv1d_linear(5L)	**5.27**	123.85	16.90
word_autoencoder(3L)	1.81	**1.46**	2.64
gan_discriminator(8L)	**12.80**	timeout	252.20
two_conv(4L)	16.69	timeout	**15.09**
img_autoencoder(11L)	**160.97**	391.09	487.54
alexnet(20L)	425.22	timeout	**66.13**
gan_generator(9L)	**412.47**	timeout	timeout
lenet(13L)	**280.91**	timeout	timeout
tutorial(10L)	**6.04**	timeout	58.29
conv_for_text(11L)	**9.04**	timeout	32.29
vgg11(28L)	**40.83**	timeout	132.67
vgg16(38L)	**82.05**	timeout	139.27
vgg19(44L)	**83.99**	timeout	189.90
densenet_main1(5L)	timeout	timeout	timeout
densenet_main2(3L)	timeout	timeout	timeout
densenet_conv_block(6L)	timeout	timeout	timeout
densenet_trans_block(3L)	timeout	timeout	timeout

TABLE III: Execution time and average API ranking for each of the 20 benchmarks using TF-IDF and GloVe models.

	SOAR w/ TF-IDF		SOAR w/ Tfidf-GloVe	
	Time(s)	Avg. Ranking	Time(s)	Avg. Ranking
conv_pool_softmax(4L)	1.60	1.0	**1.56**	1.0
img_classifier(8L)	**12.82**	2.8	31.04	2.8
three_linear(3L)	3.18	8.0	7.70	8.0
embed_conv1d_linear(5L)	**5.27**	2.4	7.75	2.4
word_autoencoder(3L)	1.81	1.0	**1.52**	1.0
gan_discriminator(8L)	**12.80**	2.8	37.01	2.8
two_conv(4L)	16.69	1.0	**13.75**	1.0
img_autoencoder(11L)	**160.97**	1.9	166.34	2.0
alexnet(20L)	**425.22**	2.1	428.42	2.1
gan_generator(9L)	**412.47**	2.0	1892.86	2.0
lenet(13L)	**280.91**	**4.3**	timeout	89.1
tutorial(10L)	**6.04**	**2.3**	21.31	2.4
conv_for_text(11L)	**9.04**	2.3	14.08	2.3
vgg11(28L)	**40.83**	1.8	73.92	1.8
vgg16(38L)	**82.05**	1.6	114.41	1.6
vgg19(44L)	**83.99**	1.5	114.98	1.5
densenet_main1(5L)	timeout	**172.8**	timeout	285.4
densenet_main2(3L)	timeout	**16.0**	timeout	387.5
densenet_conv_block(6L)	timeout	**293.3**	timeout	612.7
densenet_trans_block(3L)	timeout	**291.0**	timeout	480.0

run-time of 97.23 ± 141.58 seconds, and a median of 14.76 seconds. The average number of lines in the 16 benchmarks that we successfully migrate is 13.6 ± 12.14, whereas the average number of lines in the output programs is 18.56 ± 16.40. The reason the number of synthesized lines is higher than those in the original benchmarks is that we frequently do one-to-many mappings. In fact, 15 out of the 16 require at least one mapping that is one-to-many. In the 16 benchmarks, SOAR tests on average 4414.18 ± 5676 refactor candidates (i.e. program fragments tested for each mapping), and it needs to test a median 2111 candidates before migrating each benchmark. The reason 4 benchmarks timeout is that in each of these benchmarks there is at least one API in the benchmark that has a poor ranking (*i.e.,* not in the top 200).

D. Q2: performance of each SOAR component

We perform an ablation study to understand the effectiveness of several features in the SOAR design.

Embeddings. In Table III, we show results of SOAR using different API representation learning methods, namely TF-IDF and tfidf-GloVe, as described in Section III. We can see that for these tasks of TensorFlow to PyTorch migration, using TF-IDF-based API matching model works better than adding pre-trained GloVe embeddings. We believe this is because similar APIs are often named with same words(*e.g.,* `Conv2DTranspose` vs. `ConvTranspose2d`) or even identical name (*e.g.,* the APIs of creating a Rectified Linear Unit are both named as `ReLU(...)`), for TensorFlow and PyTorch. Thus simple word matching method like TF-IDF is suffice for API matching purposes. However, things are different for the second task we consider (see Section IV-E for more details).

Another interesting result worth noticing is that although the synthesis time differs for the two approaches, the average rankings are quite similar for most of the benchmarks. The reason is that despite the average rankings of correct target APIs being similar, the incorrect APIs ranked by the model before the correct one is different, and the time it takes to rule out those incorrect APIs varies greatly, determined largely by the number of parameters required for that API.

Error Message Understanding. As shown in Table II, SOAR performs significantly better when using the error message understanding model. We can observe that without this component, two of the benchmarks that SOAR could solve would timeout at the 1 hour mark. For the 14 benchmarks it still manages to solve, the synthesis time increases on average $4.66\times$. The number of performed evaluations also increase substantially for each benchmark. For the 16 benchmarks that SOAR successfully migrates, we evaluate an average of 43319.63 ± 61259.62 refactor candidates without the error message understanding model. This corresponds to a $9.81\times$ increase in the number of necessary evaluations when compared to the full SOAR method. In summary, we can significantly reduce the search space by interpreting error messages.

Specifications Constraints. In Table II, we also show the impact of specification constraints that describe the relationship between different parameters of a given API (see Section III-C for details). Even though, we only have these complex specifications for the 7 most common APIs, the impact on performance is significant. Without these specification we can only solve 6 out of 20 benchmarks. Relating the arguments of the APIs helps SOAR to significantly reduce the number of argument combinations that it needs to enumerate.

E. Q3: SOAR generalizability.

Our experiments so far concern deep learning library migration in Python. To study the generality of our proposed SOAR, we applied SOAR to another task of migrating from dplyr, a data manipulation package for R, to pandas, a Python library with similar functionality. Fig. 7 shows how the two API matching methods perform in this domain. While with Tfidf-GloVe, 30% of the correct APIs are ranked among the top 5, saving lots of evaluations for the synthesizer, none of the correct APIs are ranked by the TF-IDF-based matcher as

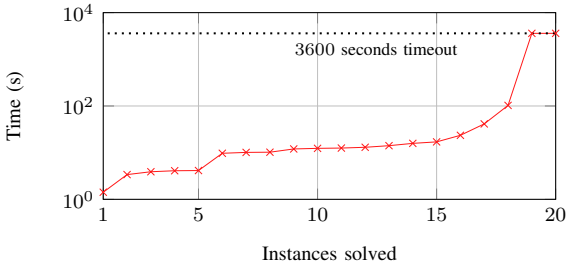

Fig. 6: Execution time for each benchmark of the dplyr-to-pandas task with a timeout of 3600 seconds.

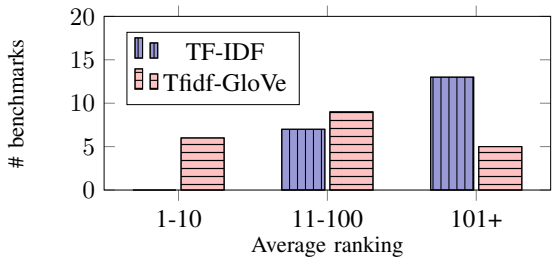

Fig. 7: Average ranking of the APIs for each of the 20 dplyr-to-pandas benchmarks.

its first 5 choices. Worse, nearly half of those are ranked above 100, making the synthesis time almost prohibitively long. We believe this is because the lexical overlap between the names of similar APIs in those two libraries is much smaller compared to the deep learning migration task. For example, dplyr's `arrange` and panda's `sort_values` provide the same functionality (they both sort the rows by a given column), but the function names are different. In this way, Tfidf-GloVe can take advantage of the pretrained embeddings to explore the similarities between APIs beyond simple TF-IDF matching.

In Figure 6, we show the time it takes to migrate each of the 20 benchmarks with a timeout of 3600 seconds when using word embeddings. We solve 18 out of 20 collected benchmarks in under 102.5 seconds. The average run time for 18 benchmarks is 17.31 ± 22.59 seconds and a median of 12.19 seconds. Note that for this task we did not consider error messages, nor specifications since we wanted to test how a basic version of SOAR would behave in a new domain. Moreover, for this domain, all the refactored benchmarks only used one-to-one mappings since no additional reshaping was needed before invoking pandas APIs. Even with these conditions, we show that we are able to successfully refactor code for a new domain across different languages.

V. Limitations and Discussion

Here we discuss the main limitations of our method and possible challenges for extending SOAR's ability to refactor new APIs, even potentially beyond the domain of data science.

Benchmarks. Our evaluation of SOAR uses benchmarks from well-known deep learning tutorials and architectures. However, they are all feed-forward networks, effectively sequences of API calls where the output of the current layer is the input of the next layer. There may be more applications that share this feature, but support for more complex structure is likely necessary to adapt to other domains.

Additionally, and naturally, the APIs in the benchmarks we collected may be biased and not reflect the set of APIs developers actually use. To assess this risk, we checked the degree to which the APIs used in our benchmarks appear to be widely used on other open-source repositories on GitHub. To do this, we collected the top 1015 starred repositories that have TensorFlow as a topic tag, which contains over 8 million lines of code and over 500K TensorFlow API calls. We found that 76% of the 1000+ repositories use API calls included in our benchmarks at least once, which validates some representativeness of our collected benchmarks.

Automatic testability. One benefit of the data science/scientific computing domain is that much of the input, output, and underlying methods are typically well-defined. As a result, it's particularly easy to test and verify the correctness of individually migrated calls, which can be processed in sequence. There may be other types of libraries that share these types of characteristics, like string manipulation or image processing libraries, whose intermediate outputs are strings/images. We also assume user-provided tests. Given the migration task, it is reasonable to assume the user has tests (the code must be sufficiently mature to justify migrating, after all), but a more general solution might benefit from automatically generating tests, which would both alleviate the input burden on the user and, potentially, reduce the risks of overfitting. In our current implementation, we moreover use the provided tests to construct smaller test cases for each mapping. This is particularly easy in this domain, because data science and deep learning API calls are often functional in their paradigm. Adapting the technique to other paradigms would require more complex test slicing or generation to support synthesis.

Correctness. Since we evaluate our migration tasks using test cases, it is always possible for our approach to overfit to these tests. However, this threat can be mitigated if the user provides a sufficiently robust test set that has provides enough coverage. Additionally, code written to different APIs may be functionally equivalent, but demonstrate different performance characteristics, which we do not evaluate. However, this fact is one reason users might find SOAR useful in the first place: a desire to migrate code from one library to another that is more performant for the given use case.

Error message understanding. The error message understanding model is built on four domain specific lexico-syntatic patterns, which we identify as hyponyms when they appear in an error message. We propose the hyponyms based on the specific syntax of DL API error messages, thus take non-trivial human effort to make it generalize to error messages that appear when calling APIs from libraries of other domains. However, we believe the idea of program mutation (Step 3 of Fig. 5) is still widely applicable for the purpose of generating SMT constraints when dealing with error messages.

Synthesis. Our approach supports one-to-many mappings but it restricts the mapping to *one* API of the target library and one or more reshaping APIs. However, this could be extended to include *many* APIs of the target library at the cost of slower synthesis times. An additional challenge is to support many-to-one or many-to-many mappings since this would require extending our synthesis algorithm. However, even with the current limitations, our experimental results show that the current approach can solve a diverse number of benchmarks.

Overall, we focus our design and evaluation on deep learning and data science libraries. These libraries have properties that render them well-suited to our task in terms of common programming paradigms, and norms, such as in the API documentation. However, we believe this is also a particularly useful domain to support, given the field's popularity and how quickly it moves, how often new libraries are released or updated, as well as the wide variety of skill sets and backgrounds present in the developers who write data science or deep learning code. Automation of migration and refactoring in this domain is very minimal, and we design SOAR as a step towards better tool support for this diverse and highly active developer population.

VI. Related Work

A. Automatic Migration

Existing work on automatic API migration uses example-based migration techniques. Lamothe *et al.* [47] proposed an approach that automatically learns API migration patterns using code examples and identified 83 API migration patterns out of 125 distinct Android APIs. Fazzini *et al.* [48] proposed APIMigrator, which learns from how developers from existing apps migrate APIs and uses differential testing to check validity of the migration. They were able to achieve 85% of the API usages in 15 apps, and validated 68% of those migrations. Meditor [49] mines open source repositories and extracts migration related code changes to automatically migrate APIs. Meditor was able to correctly migrate 218 out of 225 test cases. Unlike prior API migration tools, SOAR can migrate code without existing code examples.

SOAR also relates to automatic migration on APIs between different programming languages. Zhong *et al.* proposed MAM [50] and mined 25,805 unique API mapping relations of APIs between Java and C# with 80% accuracy. Nguyen *et al.* proposed StaMiner [19], which is a data-driven approach that statistically learns the mappings of APIs between Java and C#. Bui *et al.* [20] used a large sets of programs as input and generated numeric vector representations of the programs to adapt generative adversarial networks (GAN). Bui *et al.* then identified the cross-language API mappings via nearest-neighbors queries in the aligned vector spaces. Again these methods largely rely on existing training data, such as MAM and StaMiner [19], [50] mine mappings from parallel equivalent code from two languages (Java and C#), where SOAR only leverages the documentation for migration.

B. Program Synthesis

Program synthesis has been used to automate tasks in many different domains, such as, string manipulations [51], table transformations [44], SQL queries [52], and synthesis of Java functions [53]. However, its usage for program refactoring is scarce. ReSynth [54] uses program synthesis for refactoring of Java code by providing an interactive environment to programmers, where they indicate the desired transformation with examples of changes. Our approach differs from ReSynth since we do not require the user to provide a partially refactored code. Since our problem domain is API migration, it is unlikely that the user knows all the required APIs from the target library and can perform these edits.

NLP can be used to synthesize programs directly from natural language [51], [52] or to guide the search of the program synthesizer [55], [56]. For instance, NLP has been used to synthesize tasks related to repetitive text editing [51], SQL queries [52], and synthesis of regular expressions [55]. One can also combine input-output examples with a user-provided natural description to have a stronger specification and achieve better performance [55], [56]. Our approach follows this trend of work where we combine NLP to guide the program synthesizer with input-output examples that provide stronger guarantees in the synthesized code. However, instead of using a natural description provided by the user, our approach uses documentation from libraries to guide the search.

Using error messages from the compiler or interpreter is not common in program synthesis. The most relevant approach to ours is the one from Guo *et al.* [57] where they use type error information to refine polymorphic types when synthesizing Haskel code. In contrast, SOAR uses error messages from the interpreter not to refine the type information but to restrict the domain of the parameters and to prune the search space.

VII. Conclusions

API selection and maintenance is an important and difficult task for software development. To match evolving software, developers often have to manually refactor APIs, which is a tedious and error-prone job. We proposed SOAR to take advantage of API documentation and error messages as a rich sources of information intended for developers. We It uses natural language processing and program synthesis to automatically write refactored API calls. It is particularly well-suited for data science or deep learning library refactoring, a prevalent use case in modern development where tool support is positioned to have particular impact. SOAR collects information from both API documentation and error messages to generate logical constraints that can be used to limit the synthesizer search space. Unlike prior approaches to automatic API migration, SOAR requires no training data, and its output is guaranteed to compile and pass existing tests. Our empirical evaluation shows that SOAR can successfully refactor 16/20 of our benchmarks for the deep learning domain with an average time of 97.23 seconds, and 18/20 of the benchmark set for data wrangling tasks with an average time of 17.31 seconds.

REFERENCES

[1] C. Jaspan and J. Aldrich, "Checking framework interactions with relationships," in *ECOOP*, vol. 5653 of *Lecture Notes in Computer Science*, pp. 27–51, Springer, 2009.

[2] C. R. De Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, "How a good software practice thwarts collaboration: the multiple roles of apis in software development," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, pp. 221–230, 2004.

[3] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013.

[4] C. R. de Souza and D. F. Redmiles, "On the roles of apis in the coordination of collaborative software development," *Computer Supported Cooperative Work (CSCW)*, vol. 18, no. 5-6, p. 445, 2009.

[5] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of software maintenance and evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[6] J. H. Perkins, "Automatically generating refactorings to support API evolution," in *PASTE*, pp. 111–114, ACM, 2005.

[7] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *SIGSOFT FSE*, p. 50, ACM, 2012.

[8] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "Data scientists in software teams: State of the art and challenges," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1024–1038, 2017.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[10] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[11] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in science & engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[12] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *ASE*, pp. 307–318, IEEE Computer Society, 2009.

[13] G. Uddin and M. P. Robillard, "How API documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[14] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," in *CHI*, pp. 1019–1028, ACM, 2010.

[15] S. Gulwani, O. Polozov, R. Singh, *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[16] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *ASE*, pp. 810–822, IEEE, 2019.

[17] O. Meqdadi and S. Aljawarneh, "Bug types fixed by api-migration: a case study," in *DATA*, pp. 2:1–2:7, ACM, 2018.

[18] I. Savga, M. Rudolf, and S. Goetz, "Comeback!: a refactoring-based tool for binary-compatible framework upgrade," in *ICSE Companion*, pp. 941–942, ACM, 2008.

[19] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining API usage mappings for code migration," in *ASE*, pp. 457–468, ACM, 2014.

[20] N. D. Q. Bui, Y. Yu, and L. Jiang, "SAR: learning cross-language API mappings with little knowledge," in *ESEC/SIGSOFT FSE*, pp. 796–806, ACM, 2019.

[21] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deepam: Migrate apis with multi-modal sequence to sequence learning," *arXiv preprint arXiv:1704.07734*, 2017.

[22] "Intro to autoencoders : Tensorflow core." https://www.tensorflow.org/tutorials/generative/autoencoder, August 2020.

[23] "Api documentation : Tensorflow core v2.2.0." https://www.tensorflow.org/api_docs/index.html, August 2020.

[24] "Pytorch conv2d api documentation." https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html, August 2020.

[25] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.

[26] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *EMNLP*, pp. 1532–1543, ACL, 2014.

[27] M. F. Porter, "Snowball: A language for stemming algorithms," 2001.

[28] C. S. Perone, R. Silveira, and T. S. Paula, "Evaluation of sentence embeddings in downstream and linguistic probing tasks," *arXiv preprint arXiv:1806.06259*, 2018.

[29] S. Arora, Y. Liang, and T. Ma, "A simple but tough-to-beat baseline for sentence embeddings," in *ICLR (Poster)*, OpenReview.net, 2017.

[30] "Common crawl." https://commoncrawl.org/, August 2020.

[31] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Found. Trends Program. Lang.*, vol. 4, no. 1-2, pp. 1–119, 2017.

[32] A. Solar-Lezama, "The sketching approach to program synthesis," in *APLAS*, vol. 5904 of *Lecture Notes in Computer Science*, pp. 4–13, Springer, 2009.

[33] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho, "Encodings for enumeration-based program synthesis," in *CP*, vol. 11802 of *Lecture Notes in Computer Science*, pp. 583–599, Springer, 2019.

[34] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig, "Trinity: An extensible synthesis framework for data science," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 1914–1917, 2019.

[35] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, pp. 3111–3119, 2013.

[36] M. A. Hearst, "Automatic acquisition of hyponyms from large text corpora," in *COLING*, pp. 539–545, 1992.

[37] R. Levy and G. Andrew, "Tregex and tsurgeon: tools for querying and manipulating tree data structures.," in *LREC*, pp. 2231–2234, Citeseer, 2006.

[38] M. C. Evans, J. Bhatia, S. Wadkar, and T. D. Breaux, "An evaluation of constituency-based hyponymy extraction from privacy policies," in *RE*, pp. 312–321, IEEE Computer Society, 2017.

[39] A. Sumida and K. Torisawa, "Hacking wikipedia for hyponymy relation acquisition," in *IJCNLP*, pp. 883–888, The Association for Computer Linguistics, 2008.

[40] "Tensorflow tutorial." https://www.tensorflow.org/tutorials, August 2020.

[41] "Tensorflow applications." https://www.tensorflow.org/apidocs/python/tf/keras/applications, August 2020.

[42] "Tensorflow models." https://github.com/tensorflow/models, August 2020.

[43] "Kaggle." https://www.kaggle.com, August 2020.

[44] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," in *PLDI*, pp. 422–436, ACM, 2017.

[45] "Scrapy: A fast and powerful scraping and web crawling framework." https://scrapy.org/, August 2020.

[46] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.

[47] M. Lamothe, W. Shang, and T.-H. Chen, "A4: Automatically assisting android api migrations using code examples," *arXiv preprint arXiv:1812.04894*, 2018.

[48] M. Fazzini, Q. Xin, and A. Orso, "APIMigrator: An API-Usage Migration Tool for Android Apps," in *MOBILESoft@ICSE*, IEEE / ACM, 2020.

[49] S. Xu, Z. Dong, and N. Meng, "Meditor: inference and application of API migration edits," in *ICPC*, pp. 335–346, IEEE / ACM, 2019.

[50] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," in *ICSE (1)*, pp. 195–204, ACM, 2010.

[51] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, "Program synthesis using natural language," in *ICSE*, pp. 345–356, ACM, 2016.

[52] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "Sqlizer: query synthesis from natural language," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 63:1–63:26, 2017.

[53] K. Shi, J. Steinhardt, and P. Liang, "Frangel: component-based synthesis with control structures," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 73:1–73:29, 2019.

[54] V. Raychev, M. Schäfer, M. Sridharan, and M. T. Vechev, "Refactoring with synthesis," in *OOPSLA*, pp. 339–354, ACM, 2013.

[55] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig, "Multi-modal synthesis of regular expressions," in *PLDI*, pp. 487–502, ACM, 2020.

[56] Y. Chen, R. Martins, and Y. Feng, "Maximal multi-layer specification synthesis," in *ESEC/SIGSOFT FSE*, pp. 602–612, ACM, 2019.
[57] Z. Guo, M. James, D. Justo, J. Zhou, Z. Wang, R. Jhala, and N. Polikarpova, "Program synthesis by type-guided abstraction refinement," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 12:1–12:28, 2020.