SOAR: Synthesis for Open-Source API Refactoring

Aidan Z.H. Yang
Queen's University Carnegie Mellon University
Kingston, Canada
a.yang@queensu.ca

Abstract

The manual refactoring between APIs is a tedious and error-prone task. 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.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Keywords: software maintenance, program synthesis

ACM Reference Format:
Aidan Z.H. Yang. 2020. SOAR: Synthesis for Open-Source API Refactoring. In *Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '20), November 15–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3426430.3428129

1 Introduction

Modern software development makes heavy use of libraries, frameworks, and associated *application programming interfaces* (APIs). The APIs used by the software can become invalid or inapplicable as the software evolves. APIs themselves may become deprecated or obsolete [1]. 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.

Manual migration between APIs is tedious and error prone. 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 [2], PyTorch [3], and Numpy [4]. Moving between two such libraries often requires significant manual labor as well as domain and library-specific knowledge.

In this work, we present SOAR (Synthesis for Open-source API Refactoring), a novel technique that requires no training data to achieve API migration and refactoring. We focus our approach and evaluation on deep learning and data science APIs. However, we believe the approach will generalize to other APIs with similar properties. 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.

2 Methodology

Figure 1 shows an overview of our approach. SOAR refactors a program from one DL API to another through a pipeline of three main models. We describe the models in the following.

API 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* 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. Specifically, we use the GloVe embedding [5], which are models trained on a large natural language corpus. We train the GloVe model using API method call names scraped from API documentation. To obtain sentence embeddings from individual words, we use a weighting factor to perform a weighted linear combination of word embeddings, which is shown in detail as Equation 1, where $\mathrm{w_j}$ is the GloVe embedding of word x_j:

$$\mathrm{Embedding(x^i)} = \sum_{i=j}^{n} \frac{x_j^i \cdot \mathrm{w_j}}{\sum_{t=0}^{m} x_j^t} \tag{1}$$

Given the representation of two APIs $\mathrm{Rep(x}^i\mathrm{)}$, $\mathrm{Rep(x}^j\mathrm{)}$ in the same space $\mathrm{Rep}(\cdot)$, we compute and rank their similarity with the cosine distance between the representations.

Program synthesis: Our program synthesizer for refactoring of APIs 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

SPLASH Companion '20, November 15–20, 2020, Virtual, USA

ACM ISBN 978-1-4503-8179-6/20/11…$15.00
https://doi.org/10.1145/3426430.3428129

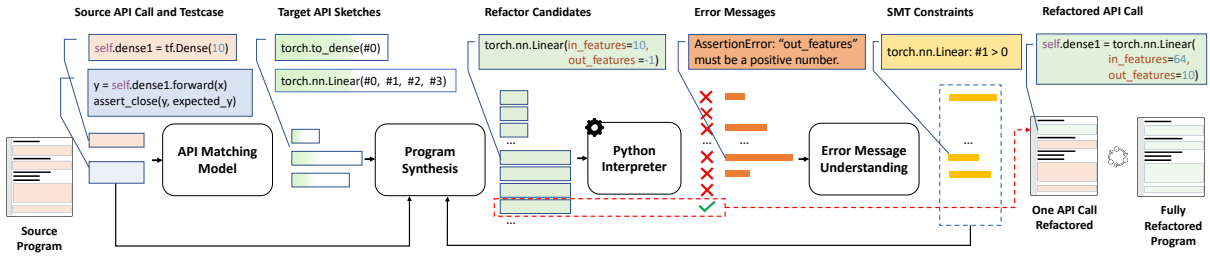

Figure 1. Overview of the SOAR architecture

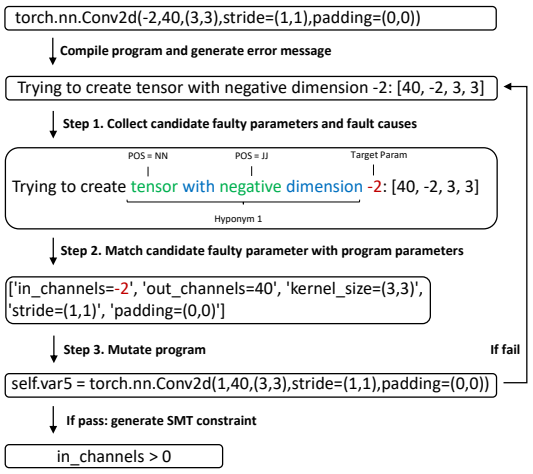

Figure 2. Example error message to SMT constraint pipeline using hyponym 1

with holes) using APIs from the target library $\mathcal{T}$. For each program sketch, we perform program enumeration on the possible completion of the API parameters. 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 the source library $\mathcal{S}$ and the target library $\mathcal{T}$. Otherwise, we continue until we find a complete program that passes all test cases.

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

Error message understanding: We use a combination of extracting hyponymy relations and Word2vec to understand run-time error messages. As outlined in Figure 2, 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 [6]. We propose a set of four lexico-syntatic patterns to identify hyponyms using noun-phrases and regular expressions frequently appearing in machine learning API error messages. We elide detailed explanation on the four hyponyms for space.

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 2, 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). Therefore, the example error message in Figure 2 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-run the program for each mutation. 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 2, 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 2.

3 Results

We collected 20 benchmarks for each of the two migration tasks (i.e, TensorFlow to PyTorch and dplyr to Pandas). In particular, for the TensorFlow to PyTorch task, we gathered 20 neural network programs from tensorflow tutorials [7], off-the-shelf models implemented with TensorFlow [8] or its model zoo [9]. The outputs from SOAR is guaranteed to compile and pass existing test cases. SOAR successfully migrates 16 of the 20 DL models with a mean run-time of 97.23 ± 141.58 seconds, and a median of 14.76 seconds. SOAR can successfully refactor 18/20 of the benchmark set for data wrangling tasks in under 102.50 seconds.

References

[1] J. H. Perkins, "Automatically generating refactorings to support api evolution," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 111–114, 2005.

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

[3] "Pytorch documentation." https://pytorch.org/docs/stable/index.html, july 2020.

[4] 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.

[5] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.

[6] M. A. Hearst, "Automatic acquisition of hyponyms from large text corpora," in *Coling 1992 volume 2: The 15th international conference on computational linguistics*, 1992.

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

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

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