Towards Automatically Estimating Porting Effort Between Web Service APIs

Hiranya Jayathilaka, Chandra Krintz, Rich Wolski
Department of Computer Science
Univ. of California, Santa Barbara

***Abstract*—In this paper, we describe a new methodology for automatically quantifying the relative work required for a programmer to port an application from one web API to another, i.e. "porting effort". Our approach defines a simple language (based on Python) with which API developers specify the semantics of API operations, a tool set that consumes and extracts semantic similarity of API operations from annotations expressed in this language, and a metric that facilitates ranking of porting effort for API operation pairs. We evaluate our approach using both randomly generated and real-world web APIs and show that our metric can correctly categorize the relative difficulty that developers associate with porting an application from one API to another.**

***Keywords*-Web APIs; Porting effort; Semantic similarity**

I. Introduction

Web services are used increasingly for the implementation of Internet accessible applications [1], [2], [3]. In this emerging development model, programmers use extant, network accessible services as external modules in their applications (or, indeed, as part of other web services). Building applications from curated web services improves programmer productivity over non-service-oriented methodologies by easing assembly, testing, and maintenance, and by improving the robustness of complex systems through the reuse of software and data components offered by providers "as-a-service" [4]. By composing an application from existing services that encapsulate common yet complicated tasks (data access and analysis, messaging, logging, security, etc.), application developers are able to work at a higher level of abstraction, thereby saving valuable development and debugging time, while at the same time providing scalable access to the core of the application logic via a network-facing service deployment. In addition, the services that an application calls upon are typically maintained and curated by an IT staff which is responsible for assuring service integrity. That is, an application that composes running web services, is able to leverage the deployment and maintenance resources devoted to those services.

A web service itself consists of one or more software components each with a well-defined but separate (in terms of coding and implementation) application programming interface (API) that is network-accessible and facilitates machine-to-machine interoperation. The web service "stack" is responsible for connecting each service implementation to the API code that exposes the web service to users (API consumers).

The growth in the popularity of this approach to application development has introduced several challenges for developers. In particular, because web-service-based applications decouple their service implementations from their APIs, the development and maintenance life cycles for APIs and service implementations are separated. As a result, APIs can change independently of the implementations they serve and new APIs (offering additional features as a superset) emerge frequently for existing services.

For example, commercial service providers experience competitive pressure to add, modify, deprecate, and retire APIs. New APIs are frequently introduced by competitors that are similar in functionality to existing APIs but offer some additional functional and/or business advantage. As a result, many commercial service providers change their APIs in terms of their operations, functionality, and behavior as well as the terms of their licensing, service level agreements (SLAs), and pricing, over time.

There are several notable examples of such commercial API churn. Amazon produced 86 releases of the Amazon EC2 APIs between August 2006 to August 2013 [5]. Eucalyptus, which mirrors the Amazon EC2 API in open source for free, on-premise use of Amazon-compatible services [6], releases its software quarterly to keep pace. Twitter released the version 1.1 of their web API on September 2012 and pulled version 1.0 of out of production on May 2013 [7]. eBay has released 25 versions of its trading API during the year of 2013 [8]. The licensing terms of the Amazon Product Advertising API changed twice between 2011 and 2013 [9].

In this paper, we describe a methodology for quantifying the effort required to port an application from one API to another, or from one API version to another version of the same API. With the possibility of high churn, it is critical that the software development, testing, and maintenance activities be informed by measures of the difficulty associated with "porting" an application to a new API or API version. We refer to this measure, in the abstract, as *porting effort*. Porting effort includes the effort needed to rewrite and refactor code, perform regression testing, and update all impacted test cases, configurations, deployment scripts, and documentation. At present, we know of no simple mechanism for estimating and reasoning about the amount of "work" required to migrate an application between web service APIs. The state of the practice is that, developers simply perform the migration speculatively (possibly losing

their work if the migration fails) or rely on their intuition and experience to obtain this measure.

To address this limitation, we propose a simple but formal mechanism to estimate porting effort for applications that use web APIs. Our approach focuses on semantic similarity of APIs and trades off complexity with efficacy and speed of analysis. Our goal is to capture the functional behavior of API operations (using existing, well-understood techniques) with sufficient accuracy and minimal programmer intervention to rank multiple API alternatives by porting effort.

Note that, in this work, we do not address the research question associated with ensuring that an API and its service implementation remain consistent and conforming to the specification. That is, because the API and implementation are essentially developed and managed as separate software components with separate life cycles, it is possible for their respective functionalities to fail to meet the specification when they are composed. The current state of the practice puts the burden for ensuring API and service implementation conformity on the development and/or maintenance staff. Indeed, the concept of DevOps [10] has been developed, in part, to address this problem organizationally. We believe that future work building on the large body of results from program verification research will address this worthy topic. In this work, however, we look only at the question of compatibility and consistency between APIs themselves and not their respective service implementations.

Initially we focus on a straightforward application of program semantics applied to the APIs of web services. This starting point exposes the working of our mechanism while providing a framework that can make use of more advanced techniques from the program verification research corpus, and semantic web technologies.

To enable this investigation, we define a simple language in terms of a restricted subset of the popular Python programming language. Developers use this language to specify the axiomatic semantics [11] of web APIs (*i.e.* the preconditions and postconditions associated with each operation). Our Python-based specification language is familiar to many developers while facilitating simple static analysis. We use the output of this analysis to measure API similarity via an extended form of the Dice coefficient [12] on the abstract syntax trees of semantic predicates, combined with Hoare's consequence rule [11] applied to API pairs.

We implement our approach and evaluate it using a number of popular APIs for social media login, airline itinerary search, and digital media video search. Our initial results indicate that developers can quantify and reason about the porting effort of migrating their applications to different web API versions and competitive implementations, without performing the porting. Our experimental results also show our approach to be efficient enough to be a practical part of the software engineering process used to develop service-composing applications. In the sections that follow, we overview porting effort and detail our approach. We then describe the empirical evaluation of our approach, discuss the results, and conclude.

II. Evaluating Porting Effort

We start with the hypothesis that application porting effort from one API to another is correlated with the degree to which the two APIs are similar. Two APIs are comparable in terms of porting effort if they are two different versions of the same API or expose same or similar services. API similarity can be syntactic, i.e., two APIs export operations with similar cardinality and data types (schemas) for their inputs and outputs. Alternatively, similarity can be semantic, i.e., two APIs are similar in terms of the functionality and behavior of their syntactically similar operations. In this work, we focus on the latter, i.e. on the semantic similarity of the operations of two APIs that match syntactically. We plan to consider syntactic similarity more comprehensively in future work.

To define a metric for application porting effort from one API to another using the semantics of their operations, we require mechanisms

- with which API developers specify the semantics of API operations (Subsection II-A) [1],
- that automate the consumption and analysis of specified API semantics (Subsection II-B), and
- that use the output from the analysis to construct a measure of porting effort for a pair of APIs (Subsection II-C).

To define each of these mechanisms and the overarching metric, we leverage and assemble extant research advances in a simple, yet new way that enables developers to estimate and rank the effort associated with porting their application to a different version of a web API or to an alternative implementation of an API. For simplicity of discussion, we assume that a pair of APIs under consideration have a *single*, syntactically matching operation. That is, in what follows, we will examine the ability to quantify similarity between individual API operations. As part of our future work we plan to extend the methodology to consider multiple operations in pairs of APIs.

A. Specifying Web API Semantics

The first mechanism of our approach is a specification language that developers can use to document the semantics of the operations in their web APIs. Our goal is to define a language that is simple, familiar, and intuitive to use, and at the same time, enables developers to specify the meaning of an API in a way that is amenable to efficient static analysis for semantic similarity. Toward this end, we leverage popular

[1]We assume that the service provider has verified that the API semantics match those of the service implementation as part of its quality assurance and testing process.

programming language syntax and tooling, and the well researched field of axiomatic semantics.

Our language is a strict subset of the Python programming language. This language choice is inspired by the widespread use of Python, Python's high level of abstraction and available tooling, and by previous works such as JML [13] and Spec# [14] that document program semantics (behavioral interface specifications) using programming language syntax. This latter research and that of others shows that using the syntax of familiar and popular programming languages to document API semantics facilitates programmer creation and editing of semantic specifications [15].

We restrict the Python language in a number of ways to facilitate analysis and to simplify the specification process by API developers. Our language only accepts single-lined Python statements that are free of side effects. We disallow side effects to preclude the consideration of internal service state. We also disallow conditionals, loops, try-catch blocks, class definitions, and function definitions.

Developers use this language to describe the behavior of API operations using axiomatic semantics – preconditions that hold prior to invoking the operation and postconditions that hold after the operation executes. We leverage axiomatic semantics as a first step toward describing and analyzing API operations in a way that reflects porting effort. We plan to consider other successful approaches [16], [17], [18] to do so as part of future work.

Developers refer API request parameters and response parameters using the built-in logical variables `input` and `output`, respectively. For example, for an operation that takes two positive numbers and responds with their sum, the preconditions can be documented using the statements $input.x > 0$ and $input.y > 0$; the postconditions can be documented as $output.sum == input.x + input.y$. These logical variables have been inspired by Hoare logic [11] and separation logic [19] to differentiate precondition values from postcondition values. The use of logical variables also enables expressing postconditions relative to preconditions, that is, postconditions can refer to the pre-state (request state) of an operation.

We do not allow invoking arbitrary functions using our language. This includes the built-in functions of Python as well as any class-level functions that can be invoked as object methods. However, we do support a number of useful predefined, side-effect-free, functions (that we have defined) when invoked as built-in functions (as opposed to object methods). We currently support the functions `len`, `implies`, `forall`, `exists`, `matches`, `datebefore`, and `dateformat`. We illustrate the use of a subset of our built-ins using simple examples below. Our language and built-ins are easily extended if and when more expressive power is required.

- The *password* input must be at least 6 characters long:
  `len(input.password) >= 6`
- All entries in the input list named *scores* must be within the range $[0, 100]$:
  `forall(entry, input.scores, 0 <= entry and entry <= 100)`
- The format of the *publishedDate* output field is *yyyy-MM-dd*:
  `dateformat(output.publishedDate, 'yyyy-MM-dd')`
- If the *country* input field is set to *US*, the *currency* output field will be set to *USD*:
  `implies(input.country == 'US', output.currency = 'USD')`

B. Comparing API Operations Pairwise

We next determine a similarity "score" by comparing the preconditions and postconditions of individual API operations. Throughout the remainder of this paper, we refer to the specified preconditions and postconditions of an API simply as *semantic predicates*. We represent semantic predicates as abstract syntax trees (ASTs).

ASTs fundamentally capture only the syntactic structure of the semantic predicates. As such, their use for semantic analysis may introduce ambiguity and error. Our goal with this work is to quantify the accuracy we are able to achieve in characterizing API similarity quickly, without requiring complex program semantic analysis methods. Using ASTs also have several other advantages such as simplicity, efficiency in the analysis, and the existence of a wide range of program development tools.

To compare a pair of matching API operations, we compute a tree similarity metric on their ASTs. To enable this, we employ a technique that is widely used for software plagiarism detection and source code evolution analysis, called the Dice coefficient [20]. The Dice coefficient has been shown in this past work to accurately extract the semantic similarity of two code fragments. Using the Dice coefficient, we treat each AST as a set of nodes over which we compute set similarity. Specifically, if P_1 and P_2 are two semantic predicates whose ASTs are T_1 and T_2 respectively, we compute the degree of similarity between the predicates P_1 and P_2 by computing the Dice coefficient on T_1 and T_2 as follows.

$$Similarity(< P_1, P_2 >) = Dice(< T_1, T_2 >) \quad (1)$$

$$Dice(< T_1, T_2 >) = \frac{2C}{2C + L + R} \quad (2)$$

C is the number of nodes common to both T_1 and T_2. L is the number of nodes unique to T_1 and R is the number of nodes unique to T_2. This approach enables us to obtain a similarity value between 0 and 1 for any two given semantic predicates, where 0 indicates a total mismatch and 1 indicates a perfect match.

We also apply a trivial transformation on the semantic predicates when performing semantic comparison that

breaks disjunctive and conjunctive predicates into their constituent predicates. This enables our mechanism to handle situations where the same set of predicates have been expressed in two APIs, but in slightly different formats.

Notice that the amount of work necessary to port from one API to another is affected by the number of predicates in each. In particular, the effort to port from a source API with fewer preconditions than the target API is more difficult than porting in the reverse direction.

To illustrate this asymmetry, let M and N be two web APIs where N has more preconditions than M. It is more difficult to port from M to N than from N to M. More preconditions imply that N's input set is more restricted than M. Therefore it cannot support all the inputs that M does. Hence some extra effort has to be put in by the developer to make sure that the application does not pass an unsupported input value to API N. However, by the same argument, porting an application from N to M should be easier. Since M's input set is less restricted than N, the developer does not have to do any extra work in this case.

Notice also that a similar asymmetry exists with respect to postconditions. If an application is to be ported from API S to API T and if T has more postconditions than S, then porting S to T is easier than the other way around. More postconditions help further restrict the output of API T. In other words, T may not produce an output that S does not. Therefore the application should be able to handle all the outputs generated by T, without having to make any code changes. On the other hand, porting from API T to S becomes more difficult, since S might produce an output that T does not.

C. Quantifying Porting Effort Between API Operations

Using the mechanism described in the previous section, we construct a measure of application porting effort using the semantic similarity of two APIs. Suppose S is a source API with the precondition set S_{pre} and the postcondition set S_{post}. Suppose T is a target API with the precondition set T_{pre} and the postcondition set T_{post}. To compute the porting effort from S to T, we first compare each member in S_{pre} against each member in T_{pre}. That is, we calculate the similarity (Dice coefficient) of each predicate pair in $S_{pre} \times T_{pre}$. Then we choose the pairs with the highest similarity, and match each member in S_{pre} to a member in T_{pre}. In other words, for each predicate $x \in S_{pre}$ we assign a predicate $y \in T_{pre}$ such that the similarity of $< x, y >$ is greater than the similarity of any $< x, z >$ where $z \in T_{pre}$ and $y \neq z$. Matched pairs are put into a new set M_{pre}. We also make sure that no member in S_{pre} or T_{pre} is matched to multiple counterparts. That is, whenever we insert a pair $< x, y >$ into M_{pre}, we mark x in S_{pre} and y in T_{pre} so that they cannot be considered for a match again. This way each member in S_{pre} can be matched to a unique member in T_{pre} as long as $|S_{pre}| \leq |T_{pre}|$. But if $|S_{pre}| > |T_{pre}|$ some members of S_{pre} will remain unmatched.

We translate the predicate assignments into a porting effort score by computing $(1-D_i)$ where D_i is the similarity of the pair $i \in M_{pre}$. We add these values up to obtain an initial porting effort score P_{eff1}. Then we consider the remaining unmatched (unmarked) predicates in S_{pre} and T_{pre}. Recall that porting to an API with more preconditions is more difficult than in the reverse direction. To reflect this asymmetry in our methodology, we increase P_{eff1} by 1 for each unmatched predicate in T_{pre}. Unmatched predicates in S_{pre} are ignored. Therefore, we have:

$$P_{eff1}(S,T) = \sum_{i \in M_{pre}} (1 - D_i) + |T_{pre}| - |M_{pre}| \quad (3)$$

We perform a similar computation for postconditions using the sets S_{post} and T_{post}. We compute the similarity of the members of $S_{post} \times T_{post}$ and pick the pairs with the highest similarity to initialize a matching set M_{post}. As a postcondition pair $< x, y >$ inserted to M_{post}, we mark x in S_{post} and y in T_{post} to ensure that no predicate is matched multiple times. Then for each pair $j \in M_{post}$ we compute $(1 - D_j)$ where D_j is the similarity of the pair j, and add these values up to obtain the porting effort score P_{eff2}. We further penalize the porting effort by increasing P_{eff2} by 1 for each unmarked (unmatched) predicate in S_{post}. This adjustment accounts for the greater difficulty associated with porting from an API with more postconditions to one with fewer postconditions.

$$P_{eff2}(S,T) = \sum_{j \in M_{post}} (1 - D_j) + |S_{post}| - |M_{post}| \quad (4)$$

We calculate the final porting effort score by combining the values obtained from previous computations. If $P_{eff}(S,T)$ is the overall porting effort from API S to API T, we have:

$$P_{eff}(S,T) = P_{eff1}(S,T) + P_{eff2}(S,T) \quad (5)$$

Algorithm 1 further illustrates our porting effort evaluation method. $Temp_1$ and $Temp_2$ are map data structures that support storing key-value pairs. The algorithm makes use of following named procedures:

- $map\_store(map, key, value)$ - Stores the given $key-value$ pair in the map.
- $map\_get\_max(map)$ - Returns the key-value pair with the largest value in the map.
- $map\_remove(map, key)$ - Removes the entry with the specified key from the map.
- $mark(set, element)$ - Marks the specified $element$ in the set.
- $unmarked(set)$ - Returns $TRUE$ if the set contains at least one unmarked element and $FALSE$ otherwise.
- $Sim(< x, y >)$ - Returns the similarity (Dice coefficient) of the predicate pair $< x, y >$.

Data: Source API S with predicate sets S_{pre}, S_{post} and Target API T with predicate sets T_{pre}, T_{post}
Result: Porting effort
$M_{pre} \longleftarrow \emptyset, M_{post} \longleftarrow \emptyset$
$P_{eff1} \longleftarrow 0, P_{eff2} \longleftarrow 0$
$Temp_1 \longleftarrow EmptyMap, Temp_2 \longleftarrow EmptyMap$
for $< x, y > \in (S_{pre} \times T_{pre})$ **do**
  $map\_store(Temp_1, < x, y >, Sim(< x, y >))$
end
while $unmarked(S_{pre})$ **and** $unmarked(T_{pre})$ **do**
  $<< x, y >, D_i > \longleftarrow map\_get\_max(Temp_1)$
  $mark(S_{pre}, x)$, $mark(T_{pre}, y)$
  $map\_remove(Temp_1, < x, y >)$
  $M_{pre} \longleftarrow M_{pre} \cup \{< x, y >\}$
  $P_{eff1} \longleftarrow P_{eff1} + (1 - D_i)$
end
$P_{eff1} = P_{eff1} + |T_{pre}| - |M_{pre}|$
for $< x, y > \in (S_{post} \times T_{post})$ **do**
  $map\_store(Temp_2, < x, y >, Sim(< x, y >))$
end
while $unmarked(S_{post})$ **and** $unmarked(T_{post})$ **do**
  $<< x, y >, D_j > \longleftarrow map\_get\_max(Temp_2)$
  $mark(S_{post}, x)$, $mark(T_{post}, y)$
  $map\_remove(Temp_2, < x, y >)$
  $M_{post} \longleftarrow M_{post} \cup \{< x, y >\}$
  $P_{eff2} \longleftarrow P_{eff2} + (1 - D_j)$
end
$P_{eff2} = P_{eff2} + |S_{post}| - |M_{post}|$
return $P_{eff1} + P_{eff2}$

Algorithm 1: Porting effort evaluation algorithm

III. Experimental Results

We have implemented a prototype of our methodology that takes two web API specifications as the input, and outputs the porting effort between them. The input web API specifications are simple JSON strings, similar to Swagger API descriptions [21]. The specifications list all of the operations of the web APIs along with their axiomatic semantics expressed using Python-based specification language described in subsection II-A.

We use this prototype to investigate the properties of our methodology as well as to expose its efficacy. In the first set of experiments, we consider randomly generated API specifications to study various characteristics of our API porting effort metric. We then consider real-world APIs and developer-perceived porting effort, and evaluate the overhead of API similarity mechanism.

A. Randomly Generated APIs

In our first experiment, we randomly generate a population of 100 API specifications. Each specification has a single operation. We compare each API against all others in the population and compute the porting effort between them. We repeat this experiment using different numbers of semantic predicates. We randomly generate the API specifications with 10, 20 and 50 semantic predicates. Our goal with this experiment is to understand how our measure of porting effort changes under these scenarios (*e.g.* to determine the sensitivity of the mechanism to supplied parameters).

Figure 1 shows the cumulative distribution functions (CDFs) of the computed porting effort as a function of the number of predicates per single API operation. A porting effort value of 0 indicates no porting effort. The data shows that the porting effort between API operations increases with the number of semantic predicates. For example, the maximum porting effort observed in APIs with 10 semantic predicates is 17.4. This goes up to 30.1 when the number of predicates is increased to 20. It further increases up to 44.9 when the semantic predicates count is set to 50. Also, when considering the CDFs of the porting effort, 50% of the API operation pairs have 4.3 or less porting effort in the population with 10 semantic predicates. In the population with 20 semantic predicates, 50% of the APIs have 7.1 or less porting effort. In the population with 50 semantic predicates, this limit further increases up to 12.9. This is inline with our experience in which, as the number of semantic predicates increases, the API consumer is forced to adhere to additional restrictions. As such, when porting among different web API operations, the developer has to take more constraints into account and must write more code to reconcile the differences. This results in increased porting effort. Our experimental results suggest that our porting effort evaluation mechanism captures this phenomenon.

It is also interesting to note that our porting effort values are not bounded by any upper limit. The porting effort could be arbitrarily large depending on the number and the complexity of the semantic predicates. We believe that this property of the metric reflects current practice. That is, it is always possible to find or create two new APIs E and F, such that the effort it takes to port an application from E to F is greater than any previously known upper bound. Our porting effort evaluation mechanism captures this property.

B. Publicly Available, Real-World APIs

We next investigate the efficacy of our approach using popular, publicly available web APIs. We list these APIs below. To evaluate our porting effort metric, we have augmented the APIs with semantic specifications manually. To enable this, we carefully analyze the API documentation and examples related to each of these web APIs. Specifically, we identify an important operation from each API set that was present across the set and specify its pre/postconditions using our specification language. Thus, these results pertain the similarity between an individual API operation that is

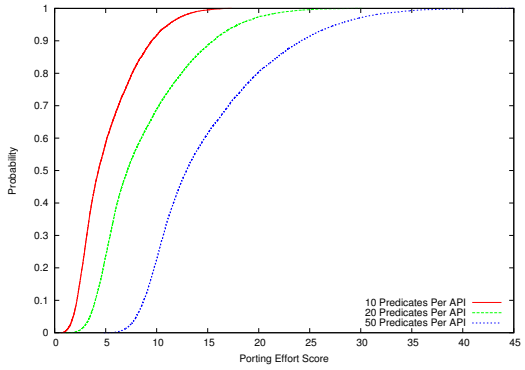

Figure 1. Porting effort CDFs for randomly generated APIs

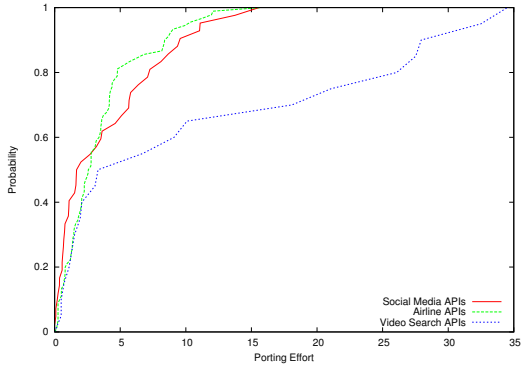

Figure 2. Porting Effort CDFs for real world APIs

common to all APIs in a set (either social media, airline services, or digital media).

- Social media login APIs: Facebook, Google, LlinkedIn, Twitter, Yahoo, Hi5, Amazon
- Airline itinerary search APIs: American Airlines, British Airways, Cathay Pacific, Delta Airlines, Emirates, Etihad, Singapore Airlines, United Airlines, Virgin America
- Digital media video search APIs: Youtube, iTunes, MovieDB, RottenTomatoes, Vimeo

We then compute the porting effort among each pair of APIs within each of the above three categories. We present the CDFs of the results in figure 2.

The data shows that a fairly large proportion of the API pairs have a low porting effort. For instance, in all three populations (social media, airlines and video search), 50% of the pairs have a porting effort of 3.3 or less, a characteristic not present in the data obtained from the randomly generated APIs. This is because, unlike in the randomly generated populations where most APIs are completely unrelated to each other, in real world API populations most APIs can and do have commonalities. For instance, most social media login APIs have similar constraints on username and password. Most airline APIs have similar requirements with respect to specifying departure and arrival cities, travel dates and the number of passengers. Most video search APIs also exhibits similar constraints, in that most APIs at least accept simple text queries to perform keyword-based search. These

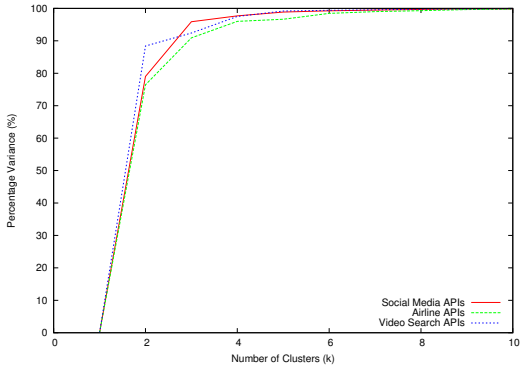

Figure 3. Percentage variance of porting effort

similarities simplify and make application porting between a number of API alternatives practical.

The CDFs of the social media APIs and the airline APIs follow relatively similar trends. However, the CDF of the video search APIs deviates from the other two and reaches a maximum porting effort value close to 35. A closer look at the API specifications showed that social media APIs and the airline APIs are similar in terms of their average semantic predicate count (8.1 and 9.3 respectively). For the video search APIs, the average predicate count is as high as 15.6 thus resulting in an increased porting effort among them. Also, some of the video search APIs have a large number of semantic predicates compared to the others. For instance, Youtube search API has 28 semantic predicates, and the iTunes search API has 30 semantic predicates. Therefore ports that involve these APIs tend to be much more complicated than the others.

C. Categorizing API Porting Difficulty

Given the observed properties of the methodology (particularly for the real-world APIs), one way which we believe it to be useful is in determining categories of difficulty. That is, it should be possible for the methodology to "cluster" API ports into groups that can be ranked in terms of difficulty (*e.g.* is a port "easy" or "hard"?)

To investigate this hypothesis we use k-means clustering to classify the results into two groups (i.e. $k = 2$). Figure 3 shows, for each sample set, the ratio of the variance explained by the categorization to the total variance in the set. Typically, this analysis shows an "elbow" in the curve corresponding to the point at which further categorization adds little explanatory power. In our study, this point of diminishing returns appears at $k = 2$.

Thus, for these API operations, it appears that our methodology should be able to divide pairwise porting effort into two categories: "easy" and "difficult." We then asked two of our lab members (lets call them D_1 and D_2) conversant with web services but not otherwise associated with this project to categorize the porting difficulty of a subset of the porting possibilities in each set as either "easy" or "difficult."

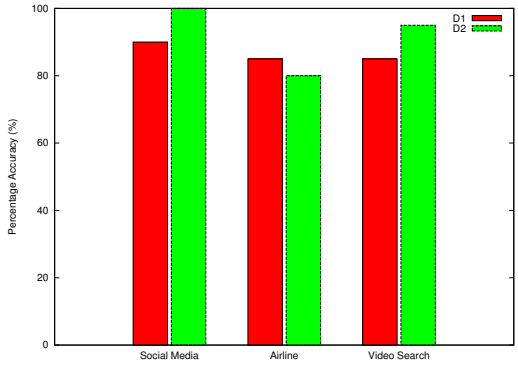

Figure 4. Percentage accuracy of the classification

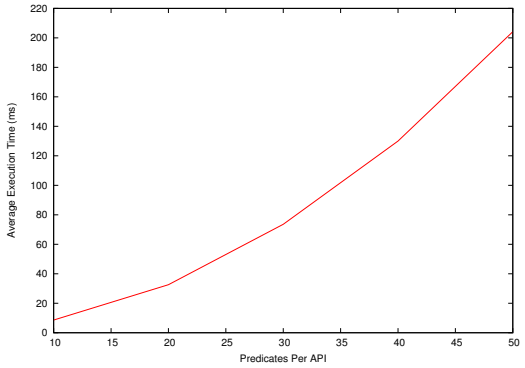

Figure 5. Average execution time of the porting effort evaluation algorithm

We gave these developers three sample sets, each consisting of 5 API specifications, randomly chosen from the above three categories (social media, airlines and video search). We then asked each developer to analyze the API specifications pairwise, and classify all possible pairs into two groups – easy and difficult – depending on the potential complexity of porting an application from one API to another. We also computed the porting effort between these web APIs using our own prototype, and used k-means clustering to classify the results into two groups.

Figure 4 shows the percentage accuracy of the classifications computed using our formal mechanism with respect to the classifications provided by developers D_1 and D_2. We compute percentage accuracy as the ratio of the number of entries classified as the same (*i.e.* agreement between the developer and the prototype) to the total sample size. Developer D_2 and the methodology obtain the same classification (100% accuracy) for the social media API.

D. Overhead

Finally, we evaluate the time overhead associated with computing web API porting effort using our mechanism. We employ our randomly generated set of 100 API specifications and compute the porting effort between each pair of APIs. We measure the time elapsed for all computations and then compute the average time per API pair. We repeat the experiment, varying the total number of semantic predicates in each API specification. We report the average times that we observe in these experiments in figure 5.

For web APIs with 10 semantic predicates, our evaluation method takes less than 10 ms. This increases up to 200 ms when the predicate count is increased to 50. This increase in execution time is due to the pairwise AST comparison operations performed by our algorithm. That is, when computing the porting effort between two APIs, our prototype compares each precondition of the source API against each precondition of the target API. In the same fashion, our prototype compares each postcondition of the source API against each postcondition of the target API. Therefore the number of AST comparisons performed is polynomial in the number of semantic predicates. Hence the average execution time of our algorithm increases polynomially with the increase in semantic predicates. However, for web APIs with 10 semantic predicates, the average execution time is below 10 ms and for web APIs with 20 semantic predicates, the average execution time is well below 50 ms. Since most of the real world web APIs that we have studied to date have a small number of predicates (the max was 30), our approach is not likely to impose a significant time overhead on the development process for applications. If required, the algorithm can be easily parallelized by running the pairwise AST comparisons in parallel to reduce the overhead further.

Overall, our approach produces useful results with a high level of accuracy. The method is efficient and can be easily applied to real world web APIs. The Python-based syntax simplifies documentation and publication of API semantics (relative to semantic ontologies, state machines, and formal logic) by API providers. If an API provider fails to publish API semantics in our language, API consumers (developers) can easily create API specifications on their own by converting the semantics of API operations described in the API documentation into Python statements.

IV. Related Work

Our research leverages and extends significant programming language semantics work. Our method for comparing web API semantics is loosely based on Hoare's rule of consequence [11]. We follow a similar rule when comparing web APIs with unequal number of preconditions or postconditions. Similar arguments have been discussed by Naumann [22] and Olderog [23] in the past. The use of programming language syntax for expressing program semantics and contracts is widely used in JML [13] and Spec# [14]. SPARK language [24] has built-in contract documentation features, where contracts are encoded in the source code as Ada comments. Our work is the first to use documented semantics at development time to reason about web service semantics and porting effort.

Our use of ASTs for comparing the semantics of two APIs is based on [20] which uses AST comparison methods for detecting program clones [20]. Baxter et al introduced the notion of syntactic similarity (based on the Dice coefficient),

as opposed to exact matches, as a more practical means of finding program segments with similar functionality and behavior. Other researchers employ similar methods to detect source code plagiarism [25], to analyze differences between programs written in different languages [26], and to analyze how program code evolves over time [27].

V. Conclusions

In this paper, we investigate a new methodology for automatically quantifying *porting effort*. Our approach defines a simple language based on Python with which API developers specify the semantics of API operations, a tool set that consumes and analyzes specified API semantics, and a new metric, based on a method used to detect code similarity, to estimate porting effort between pairs of API operations. We evaluate a prototype of this approach using randomly generated APIs to measure the sensitivity to the parameters we employ, and using competitive, publicly available APIs to determine its efficacy on real-world APIs. We employ k-means clustering to divide API ports into groups that can be ranked in terms of difficulty and show that the variance is explained by 2 clusters. Finally, we show that computation of our porting effort metric introduces minimal overhead, making it sufficiently practical to include in a developer's tool chain.

This work was funded in part by Google, IBM, NSF grants CNS-0546737, CNS-0905237, CNS-1218808, and NIH grant 1R01EB014877-01.

References

[1] M. Haines and W. Haseman, "Service-Oriented Architecture Adoption Patterns," in *System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on*, 2009, pp. 1–9.

[2] L. An, J. Yan, and L. Tong, "Methodology for web services adoption based on technology adoption theory and business process analyses," *Tsinghua Science & Technology*, vol. 13, no. 3, pp. 383 – 389, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1007021408700610

[3] M. Haines, "Web services as information systems innovation: a theoretical framework for web service technology adoption," in *Web Services, 2004. Proceedings. IEEE International Conference on*, 2004, pp. 11–16.

[4] A. Dan, R. D. Johnson, and T. Carrato, "SOA service reuse by design," in *Proceedings of the 2nd international workshop on Systems development in SOA environments*, ser. SDSOA '08. New York, NY, USA: ACM, 2008, pp. 25–28. [Online]. Available: http://doi.acm.org/10.1145/1370916.1370923

[5] "Release Notes: Amazon Web Services," http://aws.amazon.com/releasenotes/Amazon-EC2, 2013, [Online; accessed 02-Sep-2013].

[6] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus open-source cloud-computing system," in *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*. IEEE, 2009, pp. 124–131.

[7] "Twitter API v1 Retirement: Final Dates," https://dev.twitter.com/blog/api-v1-retirement-final-dates, 2013, [Online; accessed 02-Sep-2013].

[8] "eBay Trading Web Services: Release Notes," http://developer.ebay.com/DevZone/XML/docs/ReleaseNotes.html, 2013, [Online; accessed 02-Sep-2013].

[9] "Product Advertising API," https://affiliate-program.amazon.com/gp/advertising/api/detail/agreement-changes.html, 2013, [Online; accessed 02-Sep-2013].

[10] "DevOps," http://en.wikipedia.org/wiki/DevOps, 2013, [Online; accessed 27-Sep-2013].

[11] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[12] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, 1945.

[13] G. T. Leavens and Y. Cheon, "Design by Contract with JML," 2006.

[14] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: an overview," in *International conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2005.

[15] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson, "Behavioral interface specification languages," *ACM Comput. Surv.*, vol. 44, no. 3, Jun. 2012.

[16] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Vallés, and R. Van de Walle, "Functional descriptions as the bridge between hypermedia APIs and the Semantic Web," in *International Workshop on RESTful Design*, 2012.

[17] Z. Shen and J. Su, "Web service discovery based on behavior signatures," in *Proceedings of the 2005 IEEE International Conference on Services Computing - Volume 01*, ser. SCC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 279–286. [Online]. Available: http://dx.doi.org/10.1109/SCC.2005.107

[18] S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire, "Runtime verification of web service interface contracts," *Computer*, vol. 43, no. 3, Mar. 2010.

[19] J. C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *IEEE Symposium on Logic in Computer Science*, 2002.

[20] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*, 1998, pp. 368–377.

[21] "Swagger: A simple, open standard for describing REST APIs with JSON," https://developers.helloreverb.com/swagger/, 2013, [Online; accessed 27-Sep-2013].

[22] D. A. Naumann, "Calculating Sharp Adaptation Rules," *Information Processing Letters*, vol. 77, p. 2001, 2000.

[23] E. Olderog, "On the notion of expressiveness and the rule of adaptation," *Theoretical Computer Science*, vol. 24, no. 3, 1983.

[24] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[25] B. Cui, J. Li, T. Guo, J. Wang, and D. Ma, "Code Comparison System based on Abstract Syntax Tree," in *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*, 2010, pp. 668–673.

[26] M. Hashimoto and A. Mori, "Diff/TS: A Tool for Fine-Grained Structural Change Analysis," in *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, 2008.

[27] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Workshop on Mining software Repositories*, 2005.