USING SYNTACTIC AND SEMANTIC SIMILARITY OF WEB APIS TO ESTIMATE PORTING EFFORT

Hiranya Jayathilaka, Alexander Pucher, Chandra Krintz, Rich Wolski
Department of Computer Science, UC Santa Barbara, CA, USA
{hiranya,pucher,ckrintz,rich}@cs.ucsb.edu

Abstract

Service Oriented Architecture (SOA) has altered the way programmers develop applications. Instead of using standalone libraries, programmers today often incorporate curated web services, accessed via well-defined interfaces (APIs), as modules in their applications. Web APIs, however, evolve rapidly, making it critical for developers to be able to compare APIs for similarity and estimate the workload associated with "porting" applications to use different or new APIs (or API versions). Unfortunately, today there is no simple automated mechanism for analyzing the similarity between web APIs and reasoning about the porting effort that will be necessary when the web APIs that an application uses change. To address this limitation, we describe an automated methodology for analyzing API similarity and quantifying the porting effort associated with the use of web APIs. Our approach defines a simple type system and a language with which API developers specify the syntactic and semantic features of APIs. We also define algorithms that transform the syntactic and semantic features of APIs into similarity and porting effort information. We evaluate our approach using both randomly generated and real-world APIs and show that our metric captures the relative difficulty that developers associate with porting an application from one API to another.

Keywords: Web services, Web APIs, Porting effort, Syntactic similarity, Semantic similarity, Axiomatic semantics

1. INTRODUCTION

Web services are widely used to implement Internet accessible applications. In this emerging development model, programmers combine extant network accessible services to create new applications. Developing applications out of curated web services improves programmer productivity over non-service-oriented methodologies by simplifying application assembly, testing, maintenance, and by improving the robustness of complex systems through the reuse of software and data components offered by providers "as-a-service". By composing an application from existing services that encapsulate common yet complicated tasks, application developers are able to work at a higher level of abstraction, thereby saving valuable development and debugging time. Moreover, these composed applications leverage the stability and operational experience of their backend API providers.

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

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 and do change independently of the implementations they serve. In particular, new APIs (offering additional features as a superset) emerge frequently for existing services. Commercial service providers respond to competitive pressures by adding, modifying, deprecating, and retiring APIs regularly. Moreover, new APIs are introduced that are similar in functionality to existing APIs but that offer added functional and/or business advantages. Given such "API churn", developers require new tools that help them reason about API similarity and the cost of migrating, i.e. porting, an application from one API to another to adapt to API changes.

Toward this end, we present a new approach that automates the process of evaluating the similarity between two APIs or API versions, and gives developers a way to estimate the "porting effort" required to update an application to use a new API or version. Without such support, developers have only their (error-prone) intuition or must speculatively execute a port to determine its suitability.

Our approach employs simple but formal mechanisms to analyze the similarity and compatibility of web APIs. In particular, we combine techniques that extract syntactic and semantic similarity from API operations. Our syntactic analysis precisely determines the input/output type compatibility between web APIs. Our semantic analysis captures the functional behavior of type-compatible API operations using syntactic structures. We then define a scoring metric that represents porting effort and can be used it to rank API alternatives.

To enable semantic analysis, we define a simple type system for web APIs and a semantic description language based on the popular Python programming language.

Developers use this type system and the semantic description language to document the important syntactic and semantic attributes of web APIs. Our syntactic similarity analysis compares the input and output data types of different APIs and determines if one API can be used to replace another at a syntactic level. The semantic similarity analysis makes use of the axiomatic semantics (i.e. preconditions and postconditions) of web API operations, to measure API similarity via an extended form of the Dice coefficient on the abstract syntax trees of semantic predicates, combined with Hoare's consequence rule applied to API pairs. Use of axiomatic semantics allows service developers to easily document API semantics without delving into the internal implementation details of the web services. This specification language is familiar to many developers while facilitating simple static analysis.

We implement the proposed mechanisms and evaluate them 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 determine the similarity between web APIs and reason about the porting effort of migrating their applications to different web API versions and competitive implementations, without speculatively 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 detail our approach. We then describe the empirical evaluation of our algorithms, discuss the results, and conclude.

2. From API Similarity to Porting Effort

We start with the hypothesis that application porting effort from one API to another is inversely proportional to the degree to which 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 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 propose mechanisms to analyze both syntactic and semantic similarity between web APIs.

The syntactic similarity between APIs provides a simple yet very effective means of establishing design-time or compile-time compatibility of different APIs. That is, if A and B are two syntactically similar APIs (i.e. they consume similar input data types and produce similar output data types), an application written using A can be easily modified and recompiled to use B. In other words, it results in low porting effort from API A to B and vise versa. Note that this notion of syntactic similarity is not too far from the traditional sense of API compatibility often discussed in programming languages and software engineering research. In fact, our algorithm for determining syntactic similarity among web APIs is heavily based on the typical type checking and verification methods used in the above-mentioned research areas.

While syntactic similarity is simple to analyze, when considering the porting effort among web APIs, it often results in insufficient or inconclusive information. To make a sound judgment regarding porting effort one must also consider the semantic similarity between the web APIs involved. This is because it is possible for two APIs to be syntactically identical, while having drastically different semantics. For example consider an API that takes two integers and returns their sum as the output. Now consider another API that also accepts two integers and returns their product as the output. These two APIs have identical input/output data types, but they accomplish very different tasks. Therefore while it is possible to easily rewrite and recompile an application based on the first API to use the latter API, the ported application will not work as expected due to the semantic difference between the two APIs.

To overcome this type of run-time inconsistencies, semantic similarity must be checked among the APIs that are involved in the port. A semantic similarity analysis would indicate very high porting effort between the two example APIs discussed above, while a purely syntactic similarity checker may determine the porting effort to be low which is misleading.

To estimate porting effort between web APIs, we propose a two-phase API similarity analysis. In the first phase APIs are subjected to a syntactic similarity check. This check results in a simple Yes/No answer indicating whether two APIs are compatible with each other or not. If this first step yields the APIs to be syntactically compatible, we proceed to the second phase, where we perform a semantic analysis on the APIs. Our semantic analysis results in a numeric value where higher values indicate higher porting effort (i.e. lesser similarity).

We next detail the syntactic and semantic similarity checking process. While both mechanisms answer the same question (i.e. whether two given web APIs are compatible) the two mechanisms can be studied, implemented and applied independently of each other. We find in this work that the best results are achieved when we apply the two mechanisms in combination.

3. Syntactic Similarity of Web APIs

In this section we overview our approach for establishing syntactic similarity between two web APIs. Syntactic similarity is primarily based on the inputs and outputs of API operations, their cardinality and data types. This is very similar to the notion of API compatibility commonly discussed in programming languages, compilers and software engineering research. Therefore the solution

Data: Source API S with operation set OP_S and Target API T with operation set OP_T
Result: A Boolean value and a set of matching operation pairs

```
M ← ∅
for s ∈ OPs do
  matched ← FALSE
  for t ∈ OPT do
    im ← input_match(s.inputType, t.inputType)
    om ← output_match(s.outputType, t.outputType)
    if im and om then
      OPT ← OPT – { t }
      M ← M ∪ { <s,t> }
      matched ← TRUE
      break
    end
  end
  if not matched then
    return FALSE, ∅
  end
end
return TRUE, M
```

Algorithm 1: Syntactic similarity checking algorithm

we propose is heavily inspired by this already existing research and widely used techniques.

Our algorithm takes two web API descriptions (source API and the target API) as the input and determines whether the calls to source API in an application can be syntactically replaced with calls to the target API. This basically amount to establishing that the target API supports all the operations of the source API. In other words, for each operation in source API, there should be a syntactically matching operation in the target API. The syntactic match (or syntactic compatibility) between two operations can be defined based on the following guidelines:

- Two operations accept identical or compatible input data types.
- Two operations produce identical or compatible output data types.

In order to be able to automatically check for these properties, we need a way to specify the type information regarding API operations in a machine-readable manner. This requires formulating a rich type system that can be used to document the type information regarding web APIs. Most real-world type systems can be used for this purpose. However, in order to maintain language and vendor neutrality, we present the following simple type system for web APIs. This type system has been inspired by several existing type systems used in various cross-language RPC frameworks (e.g. Apache Thrift), and API description languages (e.g. Swagger, WADL, JSON schema). It is not tied to any specific programming language and therefore can be used to describe the web APIs implemented in any real-world language. Our type system consists of three categories of data types:

- Primitive types: boolean, byte, i16 (short), i32 (int), i64 (long), double, string, binary
- Container types: A list or a set of items, where all items are of the same type. A list is ordered and allows duplicates. Set is unordered and does not allow duplicates.
- Complex types: A type that consists of one or more attributes, where each attribute can be of any type.

This simple type system covers most data types encountered in real-world web APIs. It also enables defining high-level data types such as maps and other recursive data structures like lists of lists.

Web APIs often define input and output data fields as optional. To capture this information, we extend our type system with the ability to annotate objects and attributes as "required" or "optional". We assume that the input API descriptions to our analysis contain this information alongside the type information. Other API description languages (e.g. Swagger, WSDL, WADL, JSON Schema) already provide support for such annotations.

Our algorithm for analyzing syntactic similarity between web APIs accepts a source API description and a target API description. For each operation in the source API description, it attempts to find a syntactically compatible operation in the target API. That is, for each source operation it attempts to find a target operation that accepts the same or fewer inputs, and produces the same or additional outputs. To define this notion formally, suppose I_S and O_S are the input and output types of the source API respectively. Similarly, assume that I_T and O_T are the input and output types of the target API. I_S and I_T are syntactically compatible, if I_T contains the same or less attributes as I_S. If I_T includes any attributes that are not present in I_S, they must be annotated as optional to maintain syntactic compatibility among the inputs. Similarly O_S and O_T are syntactically compatible, if O_T contains the same or more attributes as O_S. From an object-oriented programming perspective, I_S and I_T are syntactically compatible if I_T is a more general type (super type) of I_S. Similarly O_S and O_T are syntactically compatible if O_S is a more general type of O_T.

When comparing complex types for syntactic matches, the algorithm may encounter attributes, which in turn are of complex types (due to the recursive nature of the type system). In this case the algorithm must recursively compare the types of the child attributes. For example, assume a source operation, which has a complex input type C_S that contains an attribute A of type T_A. Now suppose there is a target operation, which has a complex input type C_T that also contains an attribute A, but of type $T`_A$. When comparing $C_S$ against $C_T$, the algorithm must recursively compare the types $T_A$ and $T`_A$ for syntactic compatibility.

The algorithm iterates through the target API operations, looking for syntactic matches based on these guidelines. When it finds a matching target operation, the algorithm marks the operation, so that it is not matched with another source API operation. If the algorithm fails to locate a match for at least one source API operation, it returns FALSE to indicate syntactic incompatibility. It returns TRUE only if it can find matches for all source API operations. Algorithm 1 further describes this analysis. The procedures input_match and output_match are recursive functions that take two types as the input, and check for their syntactic compatibility based on the rules described earlier.

Based on the additional information available in the input API descriptions, we can make the syntactic analysis more sophisticated and accurate. For example, in addition to simply comparing the input/output data types, we can also compare the HTTP methods of operations, payload mime types and status codes returned by the APIs. This way, a source operation that consumes a JSON payload sent as a HTTP POST request and produces HTTP 201 responses, will only be matched against a target operation, which also consumes JSON payloads sent as HTTP POST requests and produces HTTP 201 responses in return. Most existing API description languages already capture this additional information regarding API operations, and hence they can easily be included in a syntactic similarity analysis.

4. SEMANTIC SIMILARITY OF WEB APIS

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

- with which API developers specify the semantics of API operations
- that automate the consumption and analysis of specified API semantics, and
- that use the output from the analysis to construct a measure of porting effort for a pair of APIs

To define 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 has 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.

4.1 Specifying 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 that, 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 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 and Spec# 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.

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 to describing the function and behavior of API operations 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 and separation logic 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.

- 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')

Note that most of the above functions are not part of the standard Python programming language. We have added them in our API description language as native constructs. The following examples illustrate how some of the above functions can be used to document API preconditions and postconditions. As seen from the above examples, our Python-based syntax coupled with the built-in functions, can be used to document even the most complex of the API semantics. The language can be easily understood by human developers, and can be easily processed by programs using a simple language parser. New built-in functions can be introduced to extend the language, and enhance its expressive power.

4.2 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).

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. 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.

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

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

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 has 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

Data: Source API S with predicate sets Spre, Spost and Target API T with predicate sets Tpre, Tpost
Result: Porting effort

Mpre ← ∅, Mpost ← ∅
Peff1 ← 0, Peff2 ← 0
Temp1 ← EmptyMap, Temp2 ← EmptyMap

for <x,y> ∈ (S_{pre} x 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> ← map_get_max($Temp_1$)
  mark(S_{pre}, x), mark(T_{pre}, y)
  map_remove($Temp_1$, <x,y>)
  M_{pre} ← M_{pre} ∪ {<x,y>}
  P_{eff1} ← P_{eff1} + (1 – D_i)
end
P_{eff1} ← P_{eff1} + |T_{pre}| - |M_{pre}|

for <x,y> ∈ (S_{post} x 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> ← map_get_max($Temp_2$)
  mark(S_{post}, x), mark(T_{post}, y)
  map_remove($Temp_2$, <x,y>)
  M_{post} ← M_{post} ∪ {<x,y>}
  P_{eff2} ← P_{eff2} + (1 – D_j)
end
P_{eff2} ← P_{eff2} + |S_{post}| - |M_{post}|

return P_{eff1} + P_{eff2}

Algorithm 2: Porting effort evaluation algorithm

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 doesn't 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 doesn't 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 doesn't. Therefore the application should be able to handle all the outputs generated by T, without having to make any code changes. Also, porting from API T to S becomes more difficult, since S might produce an output that T doesn't.

4.3 Quantifying Porting Effort of 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} x 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:

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

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} x 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.

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

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

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

Algorithm 2 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. Otherwise returns FALSE.
- Sim(<x,y>) - Returns the similarity (Dice coefficient) of the predicate pair <x,y>.

5. TWO-PHASE API SIMILARITY ANALYSIS

In this section we combine our syntactic analysis and semantic analysis into a single algorithm. The inputs to the algorithm are two web API descriptions (the source API and the target API), documented using our type system and the Python-based semantic description language. Algorithm outputs a sequence of matching (i.e. syntactically compatible) operation pairs and the porting effort value for each pair. If the algorithm fails to detect any syntactically compatible operation pairs between the source and target API, it simply returns an empty set.

Data: Source API S with operation set OP_S and Target API T with operation set OP_T
Result: Compatible operation pairs with their porting effort

```
Compatible, M ← syntactic_similarity(S,T)
if Compatible then
  R ← ∅
  for <s,t> ∈ M do
    s` ← define_api(s), t` ← define_api(t)
    effort ← semantic_similarity(s`,t`)
    R ← R ∪ { <s,t,effort> }
  end
  return R
end
return ∅
```

Algorithm 3: Two-phased API similarity analysis

The algorithm first performs syntactic similarity analysis on pairs of operations. Each pair consists of one operation from the source API, and one from the target API. We attempt to match each source API operation with a syntactically compatible target API operation. The algorithm returns the empty set and halts if it cannot find a matching target operation for at least one source API operation. The algorithm ensures that each target API operation is matched to at most one source API operation.

If this initial phase of syntactic analysis succeeds in matching all source API operations with target API operations, the algorithm proceeds to the second phase. Here the algorithm performs a semantic analysis on each of the matched operation pairs. Final output of the algorithm is a list of matching operation pairs and their corresponding porting effort values. If the algorithm returns the empty set (in first phase), it implies that a straightforward port between the given source and target APIs is not possible (i.e. at least one of the required operations are not supported by the target API). If the algorithm returns a list of matching operations, the associated porting effort values can be used to estimate the difficultly of the port in practice.

Algorithm 3 illustrates the outline of our two-phase API similarity analysis method. The procedures "syntactic_similarity" and "semantic_similarity" in the listing are functions that invoke algorithm 1 and algorithm 2 respectively. The procedure "define_api" is a helper method that defines a temporary API specification from the operation provided as input. This is there simply because we have defined algorithm 2 to accept two complete API specifications as the input. In a real-world implementation this can be simplified or even avoided if necessary.

6. PROTOTYPE IMPLEMENTATION

We implement the proposed syntactic similarity analysis and the semantic similarity analysis as a command-line tool. This tool is programmed in Python and in total consists of around 750 lines of code. It takes as input two API descriptions documented using an extended form of Swagger. Swagger is a popular JSON-based description language that syntactically describes REST APIs. It uses a type system very similar to the one described in section 3, and also captures individual operation names, HTTP methods, media types of message payloads and error codes. We extend the base Swagger description language by introducing two new JSON attributes to the operation description. These attributes are named "requires" and "ensures" (inspired by JML). Each attribute points to a list of semantic predicates written using our Python-based semantic description language. The "requires" attribute holds the preconditions of the operation, and the "ensures" attribute holds the postconditions. This extension results in a more complete API description that consists of both type information (for syntactic similarity checking) and axiomatic semantics (for semantic similarity checking).

Our decision to base our prototype on the Swagger API description language has been motivated by several reasons. These include simplicity, openness of the standard, widespread adoption in the industry, existence of many tools and libraries to process Swagger descriptions and existence of tools to auto-generate Swagger descriptions from web service codes.

We have kept our prototype very simple and lightweight. In its present state, it does not make use of any third party libraries except for the standard Python modules. Swagger specifications are read from the file system and parsed as JSON strings using Python's native JSON support. Semantic predicates are parsed into their AST representations using Python's built-in "ast" module. This greatly simplifies the implementation, and prevents us from having to write our own grammar rules or parser to process semantic predicates.

6.1 Auto-generating API Specifications

In this section we briefly discuss the issue of auto-generating API descriptions with type information and semantic predicates, so they can be used for the type of analyses described in our work. We believe that the ability to auto-generate details API specifications is crucial for this type of automated analyses and tools to be widely adopted and deployed in the industry. Handcrafting specifications for complex web APIs takes time, can be error prone and can result in various software maintenance complexities in the long run.

As a part of our research, we have implemented tools that can auto-generate Swagger API descriptions from the web services coded in Java (JAX-RS) and Python. The auto-generated specifications list the operations of the APIs, along with their HTTP methods, status codes, mime types and input/output data types. Swagger uses a type system

very similar to the one discussed in section 3, which is serialized into JSON Schema. In case of Java web services, we have implemented a Maven plug-in that gets activated at the compile-time of the source code, which performs static analysis on the code to generate the necessary Swagger API descriptions. It extracts the necessary metadata out of method signatures and JAX-RS annotations and Javadoc comments present in the code. In case of Python (which is not a compiled language), we provide a separate command-line tool that needs to be invoked manually to parse the source and generate the API specifications. This tool also extracts the required metadata from Python method signatures, decorators and docstrings available in the code.

Our tools currently do not facilitate generating API specifications with semantic information. We have left his feature for future work. We intend to utilize the techniques popularized by frameworks such as JML and PyContracts to extract the required axiomatic semantic predicates from source code into the API specifications. That is, the developers will be required to document their source code with the proper axiomatic semantics (using comments and annotations), and the API specification generators will pick up this information from the code. The design by contract research corpus already describes mechanisms that can be used to automatically check and enforce these semantic constraints at run-time, which will ensure that the web service implementations never stray away from their documented semantic contracts.

7. Experimental Results

We have developed our prototype so as to be able to separately evaluate each phase of the analysis. We first consider syntactic similarity analysis and then evaluate semantic analysis in detail. For the latter, 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 our approach.

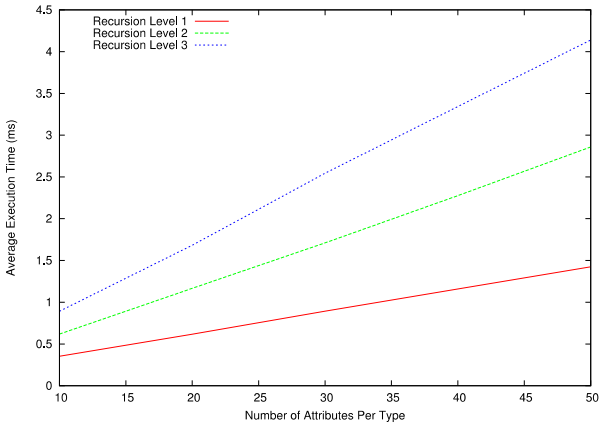

Figure 1: Average execution time of the syntactic analysis

7.1 Syntactic Similarity Results

To evaluate the effectiveness of our syntactic similarity analysis, we take the Swagger specification of an existing test API, and create multiple modified versions of it. Each modified version demonstrates a possible way the input and output data types of an API operation can change in real-world API deployments. Then we run our syntactic similarity analysis algorithm on the original API specification and each of its modified versions, and record the output of the algorithm. In addition to the simple TRUE/FALSE output of the algorithm, our prototype implementation also gives a textual description of the changes it detects between compared API specifications. We record these results in Table I.

Our experimental results show that the proposed syntactic similarity analysis is capable of detecting all possible ways data types of an API operation can change (i.e. addition, removal and modification of type attributes). Further, our prototype is capable of pinpointing the exact differences between input/output types of APIs, when there are incompatibilities among them.

Table I. Syntactic similarity analysis results.

Scenario	Expected Result	Actual Result	Generated Description
Adding a new optional input parameter	*TRUE*	TRUE	None
Adding a new required input parameter	*FALSE*	FALSE	Required input parameter introduced in new API
Removing an input parameter	*TRUE*	TRUE	None
Renaming a required input parameter	*FALSE*	FALSE	Required input parameter introduced in new API
Renaming an optional input parameter	*TRUE*	TRUE	None
Adding a new optional output parameter	*TRUE*	TRUE	None
Adding a new required output parameter	*TRUE*	TRUE	None
Removing an output parameter	*FALSE*	FALSE	No match found for output field
Renaming a required output parameter	*FALSE*	FALSE	No match found for output field
Renaming an optional output parameter	*FALSE*	FALSE	No match found for output field

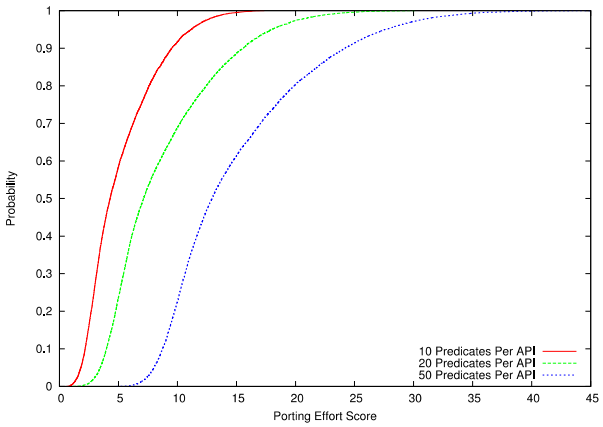

Figure 2: Porting effort CDFs for randomly generated APIs

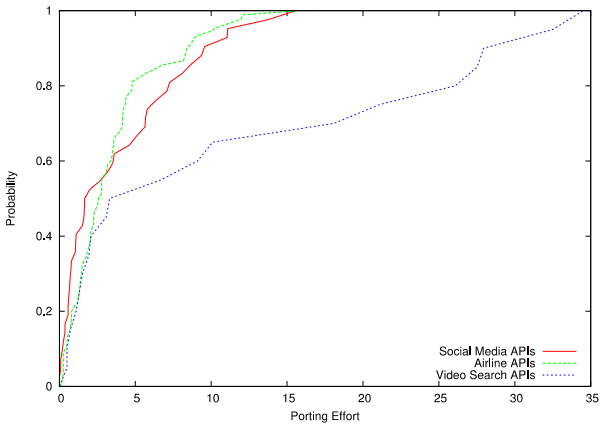

Figure 3: Porting effort CDFs for real-world APIs

Next, we evaluate the performance of syntactic analysis. We handcraft a series of API specifications, with different attribute (parameter) counts per input/output type, and different levels of recursion (nesting of types within a type). We compare each specification against itself using our algorithm 1000 times, and calculate the average execution time of a single run of the algorithm. Figure 1 depicts the results of this experiment.

The data shows that our syntactic similarity analysis scales linearly with the number of attributes available in data types. Also note that the y-axis of Figure 1 is in milliseconds, which implies negligibly small overhead (< 5ms) even in the worst case under consideration (50 attributes per type, with 3 levels of nesting).

7.2 Randomly Generated APIs

In our next experiment, we randomly generate a population of 100 API specifications. Each specification has a single operation. We semantically 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 2 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 metric 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.

7.3 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 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 3.

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 similarities simplify application porting.

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.

7.4 Categorizing API Porting Difficulty

Given this efficacy (particularly for the real-world APIs), we can determine categories of difficulty. That is, we can use 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 4 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 where further categorization adds little explanatory power. In our study, that point of diminishing returns appears at k = 2.

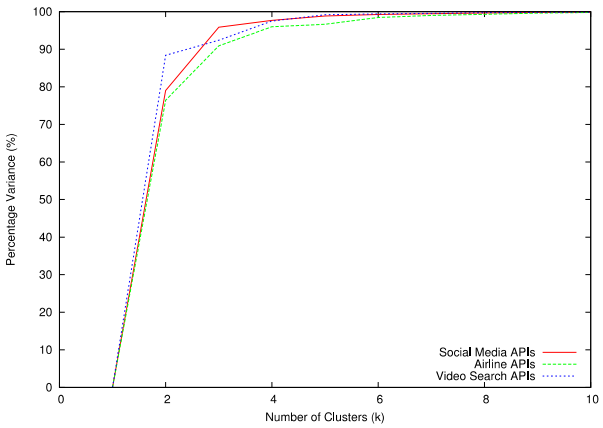

Figure 4: Percentage variance of porting effort

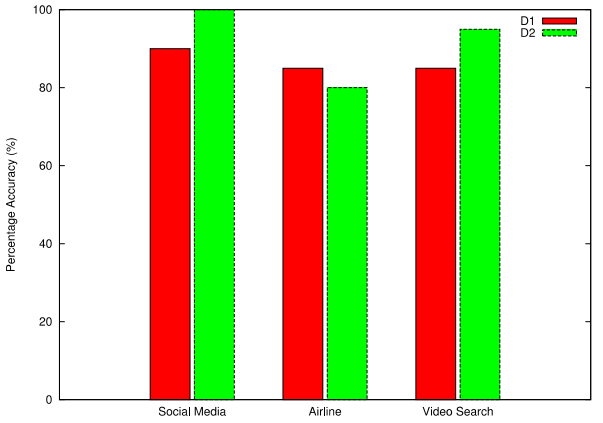

Figure 5: Percentage accuracy of the classification

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".

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 (i.e. k = 2).

Figure 5 shows the percentage accuracy of the classifications computed using our formal mechanism with

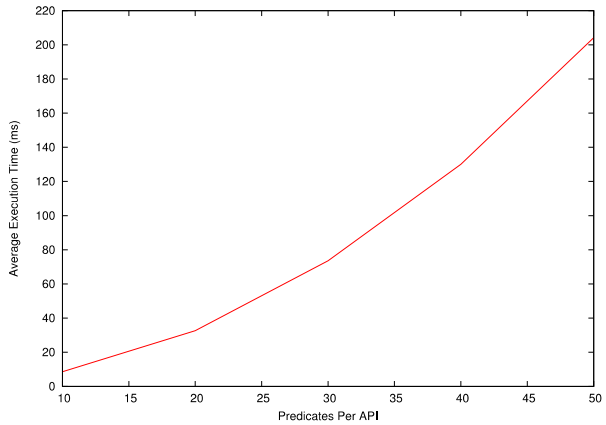

Figure 6: Average execution time of the semantic analysis

respect to the classifications provided by developers D_1 and D_2 respectively.

We compute the percentage accuracy as the ratio of the number of entries classified as the same (i.e. agreement between the developer and the methodology) to the total sample size. In terms of a simple categorization, the agreement is good. Indeed, developer D_2 and the methodology obtained the same classification (100% accuracy) for the social media API operation.

7.5 Overhead of Semantic Analysis

Finally, we the time overhead associated with computing 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 steps 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 6.

For web APIs with 10 semantic predicates, our evaluation method takes less than 10ms. This increases up to 200ms 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 10ms and for web APIs with 20 semantic predicates, the average execution time is well below 50ms. 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.

Compared to the execution time of the syntactic similarity analysis, however, the semantic similarity analysis takes much longer to complete. This indicates that in our two-phased API similarity analysis algorithm, the semantic similarity analysis component is the more expensive and critical element in terms of time complexity.

Overall, our porting effort evaluation method 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 code.

8. Related Work

This paper is an extension of our initial investigations into semantic analysis of web APIs. This work, in general, builds upon and extends research from a number of other areas in computer science. These areas include programming language and web service semantics, analysis and verification.

Static type checking techniques have been in widespread use for decades and make up one of the corner stones of programming languages research. We employ some very traditional and basic type checking mechanisms to implement our syntactic similarity analysis. The proposed input/output type comparison rules have strong roots in existing type checking techniques and object-oriented programming. Our type system has been inspired by a number of other type systems used in cross-language RPC frameworks (e.g. Apache Thrift, Google Protocol Buffers) and syntactic API description languages (e.g. Swagger, JSON Schema, WADL). Like the type systems of cross-language RPC systems, our type system is also not tied to any specific programming language. It facilitates specifying optional and required data fields, much like how most API description languages support annotating data fields as either required or optional.

Our approach of using axiomatic semantics to describe web APIs is rooted in the work of Floyd and Hoare. Floyd modeled computer programs as digraphs where vertices represent program statements and edges represent control flow. Predicates representing correctness conditions are attached to the edges. Hoare introduced the notion of Hoare triples and constructed a formalism for reasoning about program correctness using them. A Hoare triple is a logical

construct of the form P{C}Q where C is a command (an operation) in a program, P is the set of preconditions of C and Q is the set of postconditions of C. We adapted this formalism into our work where we reason about web services by describing their operations along with the respective preconditions and postconditions. Hoare's seminal work on using axiomatic semantics to reason about program correctness excludes side effects and arbitrary procedure calls. In this work, we follow the same approach for semantic predicate description language to facilitate low complexity and thus fast evaluation of API porting effort.

Several researchers have been successful in using axiomatic semantics to reason about the correctness and behavior of software constructs. Hoare himself, along with Wirth showed how axiomatic semantics can be used to describe Pascal programs. Fikes and McGuiness used axiomatic semantics to describe RDF data models. Gegg-Harrison et al introduced ProVIDE, a software development tool that allows the user to establish program correctness via specifying postconditions and then generating the corresponding preconditions. Black used axiomatic semantics to verify the behavior of a secure web server.

Our guidelines for comparing web API semantics are loosely based on Hoare's rule of consequence. The rule of consequence states that if P{C}Q and P'{C'}Q' are two Hoare triples such that P → P' and Q' → Q, then the command C' can be used in any context where the command C can be used. This is because C' has more permissive preconditions and more restrictive postconditions compared to C. We follow a similar rule when comparing web APIs with unequal number of preconditions or postconditions. Naumann and Olderog have made similar arguments.

The use of programming language syntax for expressing program semantics and contracts is a widely used concept. JML uses two primary annotations (requires and ensures) to document the preconditions and postconditions of Java methods using Java syntax. Spec# provides similar functionality for the C# language. SPARK language has built-in contract documentation features, where contracts are encoded in the source code as Ada comments. These technologies use the documented semantics or contracts mostly for verification purposes. That is, they verify whether the program adheres to the given contract at the runtime. We use the documented semantics at the development time to reason about web service semantics and porting effort by applying static analysis methods.

The use of AST representations to compare programs and reason about them is also well researched. Our approach is heavily based on the work of Baxter et al, where they used AST comparison methods for detecting program clones. 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. Cui et al showed how to use AST comparison methods for source code plagiarism detection. They showed that AST comparison based methods are capable of finding a wide range of similarities between different programs. Hashimoto and Mori augmented AST comparison methods with heuristics-driven techniques so that they can be used to efficiently analyze the differences between programs written in a wide range of programming languages. Neamtiu et al used AST comparison methods to track down and analyze how a program code base has evolved over time.

Bianchini et al introduced the notion of semantics-enabled web API selection patterns. One of the selection patterns they discuss is the substitution pattern, which aims at finding a web API that can be used to substitute another API (i.e. porting). They presented a formalism to model and quantify this selection pattern based on semantic ontologies. However, constructing comprehensive semantic ontologies requires a lot of time and manual effort, and therefore such techniques are difficult to apply in practice. Also they mainly focus on semantically annotating the input/output data elements of web APIs, whereas we look at the functional and behavioral traits of the web APIs.

There is a large body of work that uses techniques like process models, state machine models and logic to reason about web service behavior. However these formalisms are aimed at addressing the issues such as discovery, monitoring and verification. Our work deviates from these formal methods, in the sense we attempt to reason about developer experience of different web services, in the sense how much effort a developer has to put in to port an application from one web API to another.

9. CONCLUSIONS

Increasingly, web, mobile, and cloud developers integrate publicly available web services, exposed via well-defined APIs, into their Internet accessible applications. Doing so simplifies and expedites software development, testing, deployment, and management of these applications. Despite these benefits that arise from decoupling of APIs from the implementations they serve, the web service model has also introduced a key challenge for developers: API churn -- constant API evolution (versioning) and emergence of alternative, competitive implementations for the same API. As a result, it is critical that developers be able to efficiently analyze the similarity between APIs and reason about the work required to migrate their applications from one API (or API version) to another.

In this paper, we investigate a new methodology for automatically analyzing API similarity and quantifying application porting effort. Our approach defines a basic recursive type system and a simple language based on Python with which API developers document the syntactic and semantic aspects of API operations. We present algorithms that consume and analyze API features, to automatically determine whether two given APIs are syntactically compatible, and if so, how difficult it is to port an application among them. 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. 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 NSF (0751315, 0905237, and 1218808) and NIH (1R01EB014877-01).

10. REFERENCES

Haines, M., Haseman, W. (2009). Service-Oriented Architecture Adoption Patterns, *42nd Hawaii International Conference on System Sciences (HICSS), 2009*, pp. 1-9.

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

Haines, M. (2004). Web services as information systems innovation: a theoretical framework for web service technology adoption, *Proceedings of International Conference on Web Services, 2004,* pp. 11–16.

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.

Release Notes: Amazon Web Services. (2013). Retrieved September 02, 2013, from http://aws.amazon.com/ releasenotes/Amazon- EC2.

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

Twitter API v1 Retirement: Final Dates. (2013). Retrieved September 02, 2013, from https://dev.twitter.com/blog/ api-v1-retirement-final-dates.

eBay Trading Web Services: Release Notes. (2013). Retrieved September 02, 2013, from http://developer.ebay.com/DevZone/XML/docs/ReleaseNotes.html.

Product Advertising API. (2013). Retrieved September 02, 2013, from https://affiliate-program.amazon.com/gp/ advertising/api/detail/agreement-changes.html.

DevOps. (2013). Retrieved September 02, 2013, from http://en.wikipedia.org/wiki/DevOps.

C. A. R. Hoare. (1969). An axiomatic basis for computer programming, *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

L. R. Dice. (1945). Measures of the amount of ecologic association between species, *Ecology*, vol. 26, no. 3, pp. pp. 297–302, 1945.

G. T. Leavens and Y. Cheon. (2006). Design by Contract with JML. Retrieved September 02, 2013, from http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf.

M. Barnett, K. R. M. Leino, and W. Schulte. (2004). The Spec# programming system: an overview, *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, ser. CASSIS'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 49–69.

J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Muller, and M. Parkinson. (2012). Behavioral interface specification languages, *ACM Comput. Surv.*, vol. 44, no. 3, pp. 16:1–16:58.

R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Valle´s, and R. Van de Walle. (2012). Functional descriptions as the bridge between hypermedia APIs and the Semantic Web, *Proceedings of the Third International Workshop on RESTful Design*, ser. WS-REST '12. New York, NY, USA: ACM, 2012, pp. 33–40.

Z. Shen and J. Su. (2005). Web service discovery based on behavior signatures, *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.

S. Halle, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire. (2010). Runtime verification of web service interface contracts, *Computer*, vol. 43, no. 3, pp. 59–66.

J. C. Reynolds. (2002). Separation Logic: A Logic for Shared Mutable Data Structures, *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74.

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

Swagger: A simple, open standard for describing REST APIs with JSON. (2013). Retrieved September 02, 2013, from https://developers.helloreverb.com/swagger.

D. A. Naumann. (2000). Calculating Sharp Adaptation Rules, *Information Processing Letters*, vol. 77, p. 2001, 2000.

E.-R. Olderog. (1983). On the notion of expressiveness and the rule of adaptation, *Theoretical Computer Science*, vol. 24, no. 3, pp. 337 – 347.

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

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

M. Hashimoto and A. Mori. (2008). Diff/TS: A Tool for Fine-Grained Structural Change Analysis, in *15th Working Conference on Reverse Engineering, 2008. WCRE '08.* 2008, pp. 279–288.

I. Neamtiu, J. S. Foster, and M. Hicks. (2005). Understanding source code evolution using abstract syntax tree matching, *Proceedings of the 2005 international workshop on Mining software repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5.

H. Jayathilaka, C. Krintz, and R. Wolski. (2014). Towards Automatically Estimating Porting Effort between Web Service APIs. In *11th IEEE International Conference on Services Computing (SCC '14)*. 2014, pp. 774–781.

Authors

Hiranya Jayathilaka is a PhD student in the Computer Science Department at UC Santa Barbara (UCSB). His research interests include distributed systems and web/cloud services. Hiranya received a BS degree in Engineering from the Univ. of Moratuwa, Sri Lanka.

Alexander Pucher is a PhD student in Computer Science at UCSB, having received his MS degree from TU Vienna, Austria. His research`h focuses on highly resource-efficient cloud infrastructures and cross-platform interoperability.

Dr. Chandra Krintz is a Professor in the Computer Science Department at UCSB. Her research interests include cloud platforms and programming systems, and she is the progenitor of the open source cloud platform-as-a-service, AppScale. She holds MS and PhD degrees from UC San Diego.

Dr. Rich Wolski is a Professor in the Computer Science Department at UCSB. His research interests include cloud infrastructures and scientific computing, and he is the progenitor of the open source cloud infrastructure-as-a-service (Iaas), Eucalyptus. He holds MS and PhD degrees from UC Davis.