Good morning, ladies and gentlemen. My name is Hiranya and I’m a PhD candidate at UC Santa Barbara. This was a collaborative effort by myself and my advisors Dr. Chandra Krintz and Dr. Rich Wolski. My research at UCSB RACELab focuses on the management, control and governance of web APIs.

Web APIs play a crucial role in our lives. Just think about all the web applications, mobile apps and cloud services that we consume in a day-to-day basis. Think about all the online services you accessed over the last 20 minutes on your laptop or mobile device while listening to the previous speaker. These systems or services may come from many different business domains like search, social networking, online shopping and video streaming; but they all have one thing in common. That is they heavily rely on web APIs to deliver functionality and data to end-users.

Recently, we have witnessed an explosion of web APIs on the Internet. The popular online API registry ProgrammableWeb, reports that over 11,000 APIs have been recorded in their database as of today. And these are just the APIs, whose authors bothered to register them on ProgrammableWeb. So it’s quite possible that these 11,000 APIs are just the tip of a massive iceberg.

What’s even more interesting is that ProgrammableWeb lists many APIs that belong to the same business domain. This implies that any given domain has many comparable APIs that provide similar functionality.

The existence of many similar APIs means that developers should be prepared to port their applications to different APIs as the need arises. Every now and then a new API will emerge that is similar to an API that you already use, but is faster, have more features, or cheaper. And when that happens, developers should consider porting their applications to this new API in order to maintain their applications in a healthy and commercially competitive state.

In addition to having many alternative APIs, individual APIs themselves change over time prompting developers to port their code. Turns out this happens more often than most people realize. Just to give you a few concrete examples:

This level of API churn forces developers to keep porting their applications to newer versions of the APIs.

So, no matter how you look at it, in this new API-based development model, developers always have to bare the burden of porting applications across web APIs. However, that is rather unfortunate, because porting applications is typically very tedious. It can break perfectly good applications by introducing regression bugs. And developers have no way of figuring out how difficult porting a given application is. They often have to learn that while doing the port. This by all means is an engineering nightmare and contrary to all project management and scheduling practices that we know of.

Our goal in this research is to make the developer’s life a little easier by providing them a platform that can be used to analyze and reason about the difficulty or effort associated with porting an application from one web API to another. We want to keep our framework simple and easy to use so that a typical application developer can understand it and apply it to deal with his day-to-day application porting requirements involving many real world APIs. Therefore we intentionally stay clear of complex modeling techniques, formal logic and semantic ontologies. Instead we focus on simple application of existing program analysis methods to restrict and simplify the problem at hand.

We do take a very scientific approach in developing this framework. So before we go any further, lets try to understand and define what exactly we mean by the term “web API”

We are all familiar with the typical client-server interaction model inherent to web services. Basically, you have your typical web service connected to a network listening for incoming requests. Then you have your client, typically a program running on a remote computer or device, which would invoke the service by sending a request over the network. The service will process the request and send a response back. The web service can be broken down into two components as shown in the diagram. There we have the service implementation piece, which contains the code that gets executed when the service receives a request. Then we have the service interface piece which wraps the service implementation and connects it to the network. In addition to network enabling the service, the service interface also helps to abstract out the implementation details of the service, and decouple the service implementation from its consumers.

For the purpose of this presentation, when we mention the term “web API”, we are referring to this interface portion of a web service. Note that web API also consists of code, but it is typically developed separate from the service implementation and follows a different lifecycle.

Now that we have defined what we mean by web APIs, we can go ahead and nail down the exact research problem that we are trying to solve. Our main question for this presentation is…

In order to answer this question, we start with the intuitive idea that,…

… The difficulty of porting an application from API A to API B is dependent on how similar the two APIs are.

Similarity of web APIs can be analyzed from two perspectives. There is syntactic similarity, which is concerned with the input/output data types, message formats and schemas associated with APIs. Then there is semantic similarity which looks at the functional and behavioral traits of web APIs. It turns out checking for syntactic similarity is easy. In fact the programming languages community has already devised methods for effectively dealing with that issue. Given two API descriptions that describe the operations and the schemas of the inputs and outputs, like in WSDL files, it is possible to compare them and establish syntactic similarity using one of many recursive type checking algorithms. Semantic similarity on the other hand is very difficult to analyze using existing methods. There are no widely used tools or techniques that can analyze two API descriptions and conclude in a reasonable amount of time, whether they are semantically compatible.

In this work, we focus on devising a method for reasoning about the application porting effort based on the semantic similarity of web APIs. We do not consider the syntactic similarity in our research, but it is quite possible to enhance the outcome of our tools by combining our approach with already existing syntactic similarity checking methods

Now lets see what our solution looks like at a high-level.

We propose a new language for describing the semantics of web APIs, and a metric that measures the difficulty associated with porting an application from one web API to another. Then we provide an algorithm that computes the porting effort metric from the web APIs documented using our language.

Now lets take a closer look at these different elements, starting from the API semantic description language.

Our language captures the axiomatic semantics of web APIs. That is, it documents the preconditions and postconditions associated with API operations. Axiomatic semantics are great for describing web services, since they allow describing the functionality and behavior of a service without digging into the service implementation, which is generally abstracted away from users and application developers.

To specify these preconditions and postconditions we employ a simple syntax based on the Python programming language. Use of a real programming language to specify API semantics enables API developers to easily document their APIs without having to learn any new tools, languages or service modeling techniques. Also you can use existing Python programming tools and verifiers to write and validate the API semantic descriptions.

In order to keep our analysis simple and efficient, we restrict the allowed Python subset in several ways. Most prominently we disallow all complex constructs such as conditionals, loops, user defined functions, classes and try-catch blocks. We only allow simple, single-lined Python statements that we call semantic predicates. However, we do support a set of built-in functions that can be used to easily express certain complex semantics.

I’d like to refer you to the paper for more details on our semantic description language, but for the sake of completeness let me present a simple example.

<< example >>

Next, lets talk about our algorithm that computes the porting effort values. It takes two API descriptions as the input, the source API and the target API. Source API represents the API that your application is currently using, and target API is the one that you’re planning to port your application into.

Our algorithm starts by converting all the semantic predicates of the APIs into their abstract syntax tree representations. By doing so we reduce the problem of API semantic analysis into a matter of syntax comparison. The rest of the algorithm works by performing a structural analysis on these ASTs. It does limit the level of accuracy obtainable from our algorithm, but in return we get a lot of simplicity and performance. And, as I will show you in the next few slides, our empirical results indicate that our algorithm produces sufficiently accurate results when dealing with real world web APIs, despite this simplification.

Having converted the predicates into ASTs, our algorithm then pairs up source API predicates with target API predicates, and computes a similarity function on them. We use the Dice Coefficient as our similarity function. It is a metric used to quantify the similarity of sets or trees. This similarity value is a number between 0 and 1, where 0 implies no similarity and 1 implies identical ASTs.

Then we convert these similarity values into difficulty values by subtracting them from 1. Next, we aggregate the difficulty values into a single porting effort score by adding them up. We choose the predicate pairs with highest similarity during this aggregation and we make sure each predicate is included in the aggregate score once. Finally, we make some adjustments to the score based on Hoare’s consequence rules. More specifically we increase the porting effort score, if the target API has more preconditions or the source API has more postconditions. If the target API has more preconditions, that implies it accepts a more restricted input set compared to the source API. Therefore porting to such a target API should be more difficult. A similar argument can be made on the source APIs that have more postconditions.

Now lets look at some experimental results obtained using our approach. We did a lot of experiments using randomly generated API descriptions which we have included in the paper. In this presentation I’m going to jump straight into the results obtained using real world APIs. We analyzed a number of real world APIs taken from three categories. For each API, we went through the available API documentation and sample code, and created semantic descriptions using our language. Then we ran our algorithm on pairs of real world APIs, within each category. When we plotted the calculated porting effort values, we get this CDF…

When considering the three curves, we can see that airline APIs and social media APIs follow more or less the same trend. The video search APIs deviates greatly from these two curves and reaches porting effort values as high as 30. The reason behind this behavior is that both airline APIs and social media APIs have similar number of semantic predicates, averaging around 8. Video search API set has an average predicate count of 15 which shifts its CDF curve to the right. We believe this to be a good result. An API with more semantic predicates, imposes more restrictions on the application developers. Therefore the effort required to mediate the semantic differences among APIs is higher, which results in a higher application porting effort.

In our next experiment, we gave three sample API sets to 2 human programmers, and asked them to group API pairs into two categories, based on the potential difficulty of porting applications between them. The developers manually analyzed the provided API descriptions, and also referred to the API documentation where necessary and a did a manual classification of API pairs into the groups “easy” and “hard”.

Then we ran our porting effort calculation algorithm on the same set of APIs, and grouped the obtained scores into two categories using k-means clustering. Next, we evaluated the accuracy of our automated classification with respect to the two manual classifications we received from the two developers.

Here y-axis is the percentage of API classifications obtained from our mechanism, that agree with the baseline classifications we received from the human developers.

Note that in all cases considered, the relative accuracy of our automated mechanism either equals or exceeds 80%. In one case it even reaches 100%.

We understand that this doesn’t qualify as a comprehensive user study from which we can draw firm conclusions. We need to get lot more developers involved in the experiment for that. But before we go there, we wanted to obtain some anecdotal evidence to show that our mechanism reflects the thinking of at least some developers. That was the purpose behind this 2-people study. And results of this small-scale user study is clearly quite promising and encouraging in that regard. As a result, we intend to further expand our study to large groups of programmers in the future.

In conclusion, we presented a new approach for analyzing the effort required to port an application from one web API to another. Our algorithm is simple, practical to implement and fast. It delivers accurate analysis and classification results on a wide range of real world APIs, so that combined with its simplicity and speed makes it suitable to be used as a key component of a developer's toolkit. Our approach uses Python syntax to document the axiomatic semantics of web APIs, which enables service developers to easily specify their API semantics. In instances where the API provider does not provide such specifications, the application developers that consume the APIs can create their own semantic specifications by going through the available API documentation. And finally, we can further improve our porting effort calculation approach by combining it with syntactic similarity checking methods widely used in programming languages research.

And with that I conclude my presentation, and open up this session for questions from the audience.