Developing Systems for API Governance

Chandra Krintz, Hiranya Jayathilaka, Stratos Dimopoulos, Alexander Pucher, Rich Wolski, and Tevfik Bultan
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA, USA
{ckrintz,hiranya,stratos,pucher,rich,bultan}@cs.ucsb.edu

UCSB Computer Science Technical Report Number 2013-06

***Abstract*—As scalable information technology evolves to a more cloud-like model, *digital assets* (code, data and software environments) that increasingly form the basis of research and education require curation as web-accessible services. "Service-izing" digital assets consists of encapsulating assets in software that exposes them to web and mobile applications via well-defined, network accessible, application programming interfaces (APIs). The stability, maintenance, and lifecycle of these APIs is critical to the utility of the digital assets they serve. Our work focuses on the development methodologies and technologies for *API governance* – policy, implementation, and deployment functions for IT management of APIs at scale. This paper presents our view of API governance in a technology landscape that is trending towards reliance on web services. It also outlines some early results generated by our investigation of a prototype we are developing for implementing API governance at scale.**

I. Introduction

With the advent of cloud computing, digital assets (code, data, and software environments) and not infrastructure are becoming the resources that must capture scientific, research, and educational investment. The science and education communities use well-developed Information Technology (IT) practices to manage the infrastructure in the data centers they manage today. With high-quality, low-cost infrastructure available from public clouds, IT management must increase its focus on the maintenance, protection, and lifecycle control of the digital assets – the code, software environments, and data – that comprise the "computational" component of any scientific or educational endeavor.

Cloud computing has had the same effect on commercial IT practices and as a result new systems and methodologies for managing digital assets "as a Service," such as Hadoop [1], DevOps [2] and NoSQL [3] continue to proliferate. As a result, previously successful software and IT approaches (*e.g.* Service-oriented Architecture, web services, machine virtualization) are enjoying a renaissance of utility.

In particular, the IT industry has embraced the web services technologies over the last decade [4], [5], [6]. A web service is a software component with a well-defined interface, which can be accessed over a network, and facilitates interoperable machine-to-machine interactions. The interface portion of a web service is often termed a web Application Programming Interface or a web API. It is the API that both defines and controls what operations can be performed on each asset, by whom, and under what conditions.

APIs also decouple the implementation of this access functionality from the technologies that are used to manage and store the assets. That is, while the assets may remain the same, the technologies used to serve and implement them can change, particularly as technological advances reduce implementation costs. APIs must preserve user access to the assets when the underlying technologies change. Thus, the *lifecycle* of the API follows the lifecycle of its assets and *not* the lifecycle of the surrounding technologies which typically change at a more rapid pace.

Finally, APIs in the modern application environment must provide standardized network-facing access so that the widest possible variety of applications and devices can access their digital assets. They must also support availability guarantees and fault management strategies associated with the assets and the implementing technologies. It is the combination of standardized, continuously available, networked access that will enable purely digital science to scale.

Thus, APIs provide three functionalities that are critical for the management of digital assets and artifacts. They

- *implement control* over the assets, both in terms of operations and access control,
- *protect asset lifecycle* from technological changes driven by advances and/or economics, and
- *enable scale* through standardized, networked connectivity, and fault management.

Because of these functions, implementing and managing APIs can be more important than either the digital assets or the technologies that underly them.

However, despite the primacy of APIs in the new digital IT environment little technology has yet been developed to implement *API governance* – combined policy, implementation, and deployment control – in a research and educational context. Good commercial technologies exist for managing digital assets and also for developing both hardware and software necessary to implement digital assets (including the necessary APIs). A few technologies [7], [8] are emerging for packaging and cataloging APIs. However, technologies for providing stewardship of APIs through all phases of governance are rare.

Our work focuses on the development of methodologies and systems for implementing API governance. In addition to typical API management features (e.g. cataloging, search,

deployment support, etc.) we believe a system for API governance must include the following capabilities.

- *Change Control* – When API changes are necessary, the extent of the effects of the change must be predictable and implemented in a uniform, consistent way. If changes need to be rolled back, the return to previous functionality is likewise consistent, complete, and managed. This requires development of efficient automated change-impact analysis techniques that can determine the potential effects of a proposed change.
- *Policy Specification and Analysis* – Since APIs are gateways to digital assets, they should allow only the authorized clients access these resources. API governance requires development of mechanisms for specification of access control policies, and analysis and runtime enforcement of these policies.
- *Consistent Policy Implementation* – Policies governing the use of digital assets and/or their APIs are implemented consistently across all assets regardless of the constituent technologies that are used to implement the assets themselves.
- *Implementation Portability* – API implementation is decoupled from the implementation of the digital assets. As technologies evolve or, more problematically, devolve when they sunset, API integrity must be maintained across different implementations.
- *Monitoring and Auditing* – API governance must include a unified approach to monitoring and auditing API activity. A unified approach is particularly important when digital assets make heavy use of open source as many research and educational environments do today. Using runtime monitoring, erroneous or malicious behaviors can be identified and resolved by appropriate exception handling mechanisms. This is necessary since clients that interact with an API are typically not controlled by the organization that governs that API.

In the remainder of this paper, we describe our initial investigations of API governance for curated digital assets accessed via web services. In particular, we discuss a new system for automatically determining the similarity between different web service APIs and the degree to which that similarity correlates with programmer porting effort.

Determining API similarity is a critical component of API governance for two reasons. First, as new technologies emerge, they may require enhancements or extensions to existing APIs. Determining the degree to which a new technological implementation supports and existing API and/or the extent of the effort that will be needed to port applications to a new API are critical aspects of change control and, hence, API governance.

II. Web Services

Today, a large number of mobile, web and cloud applications are developed using web services as building blocks. We expect this software engineering trend to continue. Thus, we believe that in the future, web services (and not programming language packaging) will provide the basis for software modularity. There are several benefits to developing applications based on web services. These include increased code reuse, increased software maintainability and improved software reliability via using well-tested code [9].

Because web service applications must be rewritten when an API changes, the lifecycle of an API is typically longer than the infrastructure or service implementation to which it serves as an interface. Insulating applications from technological change in the software and hardware infrastructure is a key advantage of the web service approach.

However, APIs are not immutable. For example, there have been 86 releases of the Amazon EC2 service from August 2006 to August 2013 [10]. Twitter released the version 1.1 of their web API on September 2012, and pulled the version 1.0 of out of production on May 2013 [11]. eBay has released 13 versions of their trading API during the first two quarters of year 2013 alone [12]. The licensing terms of the Amazon Product Advertising API have changed twice between the years of 2011 and 2013 [13].

Porting an application from one web API to another is a cumbersome activity. This process is further complicated because no simple mechanism exists for evaluating the amount of effort needed to port applications between web APIs. In most situations, the only way to evaluate the complexity of a port is to actually do it, thereby incurring the development cost as a way of determining what it will be. Often, as an alternative, the IT organization must rely on the intuition of developers to estimate the porting effort.

III. Methodology

We propose a formal and automated mechanism for evaluating the porting effort of applications from one web API to another. Application porting effort can be analyzed from two perspectives:

- *Syntactic similarity* – Similarity of the inputs and outputs of web APIs
- *Semantic similarity* – Functional and behavioral similarity of web APIs

Of the two, checking for syntactic similarity is a solved problem. Given a machine-readable description of the inputs and outputs of the APIs, static analysis methods can be used to verify whether two web services APIs are syntactically compatible. However, checking the semantic compatibility of web APIs is complicated. Existing semantic matchmaking methods solve this problem using semantic ontologies, process models or state machine models, all of which are complex, laborious and potentially computationally intensive. We address this problem using much more efficient techniques that are simple to implement.

Our approach uses axiomatic semantics to describe the functionality and behavior of web APIs. We document the semantics in a machine-readable manner using a Python subset specifically designed to capture web service API characteristics. We keep our approach simple by disallowing complex

programming constructs (e.g. loops, functions etc.), and restricting the subset to a side-effect-free programming model when documenting API semantics. Then we derive abstract syntax tree (AST) representations [14] of semantic predicates expressed in our language to compare and reason about the semantic similarity of different web APIs. We use a Dice coefficient [15] based AST similarity algorithm and Hoare's consequence rule [16] to compute a porting effort score for any two given web APIs.

IV. Initial Findings

To establish the practicality of our model, we have implemented a prototype of the proposed porting effort evaluation mechanism. Using this prototype, we studied the resolution characteristics of the methodology using randomly generated APIs where we could control the porting "distance" *a priori*. We also analyzed the porting effort associated with a number of web APIs from popular e-commerce and social networking venues. Our experimental results indicate that the proposed mechanism is efficient, and delivers accurate results under most circumstances. We have further tested the validity of our approach by comparing the results computed by our formal mechanism with the results provided by some human developers when manually analyzing several web APIs.

Unsurprisingly, the initial experiments performed using randomly generated web API specifications show that the porting effort between APIs tends to increase with the number of semantic predicates. As the number of semantic predicates increases, the API consumer (*e.g.* the developer of the application consuming the API) is put under more and more restrictions. Therefore, when porting among different web APIs, the developer has to take more constraints into account, and do more patchwork to reconcile the differences among these restrictions. These activities increase the porting effort, and the experimental results suggest that our porting effort evaluation mechanism captures this phenomenon well.

When considering real world web API sets, we see that a fairly large proportion of the API pairs within a set of APIs ostensibly serving the same function have a low porting effort. For example, we looked at three API populations: social media, airline e-commerce, and video search. In each of them, 50% of the pairs have a low porting effort score, a modal characteristic not present in the data obtained from the randomly generated APIs. Again, this result is confirmational with respect to intuition. In API populations that serve similar web services most APIs have a lot in common with each other. For example, 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 have some constraints in common, in the sense most APIs at least accept simple text queries to perform keyword-based search. These similarities explain how current mobile and desktop applications currently manage multiple APIs in e-commerce and social computing settings.

Finally, we asked student developers who were in the process of building applications to access some of these venues for their respective scoring of the difficulty associated with porting from on API to another. We then applied statistical clustering to both the scoring results generated by our methodology and to the results given to us by the developers. The clustering results show good registration between our methodology and human perception of porting effort with respect to categorizing a particular port as "hard" or "easy."

V. Future Work

The proposed mechanism provides a way to reason about the application porting effort between web APIs at a semantic level. Currently it doesn't take the syntactic compatibility of the web APIs into account. That is, it doesn't consider the compatibility of web service requests (inputs) and responses (outputs) when computing the porting effort. We believe that adding this capability will make our mechanism even more powerful and useful. Most web services use data types such as XML and JSON for receiving inputs and sending outputs. This uniformity enables modeling the web service inputs and outputs using a structured data model consisting of a simple type system. Once such a data model has been formulated, it is a rather well-understood task to analyze the syntactic compatibility of the web APIs by employing static analysis methods. We are already in the process of exploring this possibility.

We are also beginning to study the other aspects of API governance. In particular, we are interested in developing methodologies for policy-based change control and auditing capabilities to manage APIs in large-scale deployments. Cloud computing, as both a research and commercial discipline, has developed a number of new approaches for managing "soft" resources in highly scalable settings. We plan to leverage many of these developments to develop API governance methods for scalable systems.

We believe that our approach to semantic specification of APIs in machine readable form can be extended to specify access control policies. Using a restricted form of an existing programming language (such as Python) provides two benefits: 1) It does not require the users to learn a new language for policy specification, 2) It allows us to appropriately restrict the language to enable automated analysis. Using this approach, we plan to develop automated techniques for policy analysis, such as checking if a policy is stronger or weaker than another policy, and change-impact analyses that identify the effects of a policy change. We plan to implement these analyses by translating the analysis questions to satisfiability queries in decidable theories and then using automated decision procedures (such as SAT-Solvers or SMT-Solvers). Focusing on API usage and using a restricted policy language will enable us to develop a scalable static analysis framework for API policies.

VI. Conclusion

APIs have emerged as a key component of the modern digital economy and we believe the scientific community

will want to leverage the technological developments that this primacy is engendering. However, even though APIs are the longest-lived and most expensive software artifacts, little research has yet focused on what is necessary to implement good IT governance of them.

Our work is taking an initial step towards the development of a system for implementing API governance. We have focused on the problem of determining API similarity which we measure as the effort necessary to port an application from one API to another. Our initial results are promising, leading us to conclude that API governance is worthy of on-going investigation.

REFERENCES

[1] "Hadoop MapReduce," "http://hadoop.apache.org/".

[2] "DevOps," "http://en.wikipedia.org/wiki/DevOps".

[3] "NoSQL," "http://en.wikipedia.org/wiki/NoSQL".

[4] 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.

[5] 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

[6] 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.

[7] "Mashery," "http://www.mashery.com/".

[8] "Layer7," "http://www.layer7tech.com/".

[9] 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

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

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

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

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

[14] 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.

[15] W. B. Frakes, "Stemming algorithms." 1992.

[16] 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