Towards software analysis as a service

Ghezzi, G; Gall, H C


Posted at the Zurich Open Repository and Archive, University of Zurich. http://www.zora.uzh.ch

Towards software analysis as a service

Abstract

Throughout the years software engineers have come up with a myriad of specialized tools and techniques that focus on a certain type of analysis, such as metrics extraction, evolution tracking, co-change detection, bug prediction, all the way up to social network analysis of team dynamics. However, easy and straight forward synergies between these analyses/tools rarely exist because of their stand-alone nature, their platform dependence, their different input and output formats and the variety of systems to analyze. This significantly hampers their usage and reduces their acceptance by other researchers and software companies. To overcome this problem we propose a distributed and collaborative software analysis platform to enable a seamless interoperability of software analysis tools across platform, geographical and organizational boundaries. In particular, we devise software analysis tools as services that can be accessed and composed over the Internet. These distributed services shall be widely accessible through a software analysis broker where organizations and research groups can register and share their tools. To enable (semi)-automatic use and composition of these tools, they are classified and mapped into a software analysis taxonomy and adhere to specific meta-models and ontologies for their category of analysis.
Towards Software Analysis as a Service

Giacomo Ghezzi and Harald C. Gall
s.e.a.l. – software evolution and architecture lab
University of Zurich, Department of Informatics, Switzerland
{ghezzi, gall}@ifi.uzh.ch

Abstract

Throughout the years software engineers have come up with a myriad of specialized tools and techniques that focus on a certain type of analysis, such as metrics extraction, evolution tracking, co-change detection, bug prediction, all the way up to social network analysis of team dynamics. However, easy and straightforward synergies between these analyses/tools rarely exist because of their stand-alone nature, their platform dependence, their different input and output formats and the variety of systems to analyze. This significantly hampers their usage and reduces their acceptance by other researchers and software companies. To overcome this problem we propose a distributed and collaborative software analysis platform to enable a seamless interoperability of software analysis tools across platform, geographical and organizational boundaries. In particular, we devise software analysis tools as services that can be accessed and composed over the Internet. These distributed services shall be widely accessible through a software analysis broker where organizations and research groups can register and share their tools. To enable (semi)-automatic use and composition of these tools, they are classified and mapped into a software analysis taxonomy and adhere to specific meta-models and ontologies for their category of analysis.

1. Introduction

Successful software systems must change or they become progressively less useful, but as they evolve, they become more complex and consequently more resources are needed to preserve and simplify their structure [29]. Studies estimate the costs for the maintenance and evolution of large, complex software systems from 50% to 95% of the total costs in the software life-cycle. To reduce these costs several techniques and tools have been developed: to discover components that need to be modified when a new feature is integrated, to detect architectural shortcomings, to detect error prone modules, or for project managers to estimate the maintenance costs and allow for better planning, etc.

Software analysis tools mainly focus on a particular kind of analysis to produce the results wanted by the engineer. So, if different analyses are required, the engineer needs to run several tools, each one specialized on a particular aspect, ranging from pure source code analysis to duplication analysis, co-change analysis and visualization. In addition to this, all these techniques have their own explicit or implicit meta-model which dictates how to represent the input and the output data. Thus the sharing of information between tools is at most possible by means of a cumbersome export towards files complying to a specified exchange format. Also, if there exist services of the same kind (e.g. duplication analysis) there is no way of comparing the results or integrating them other than manual investigation. And there are even more issues that limit tool interoperability as for example, platform and language dependence.

We claim that this status quo severely hampers software evolution research and a critical assessment of the research fields uncovers the fact that people keep reinventing the same wheels with little advancement of the field as a whole.

So our goal is to devise a distributed and collaborative software analysis platform to allow for interoperability of software analysis tools across platform, geographical and organisational boundaries. Such tools will be mapped into a software analysis taxonomy and will adhere to specific meta-models and ontologies for their category of analysis and offer a common service interface that enables their composite use on the Internet. These distributed analysis services will be accessible through an incrementally augmented software analysis catalog, where organisations and research groups can register and share their tools.

The remainder of this paper is organized as follows:
Section 2 gives an overview about current software analyses to be supported by a service platform. Section 3 explains our proposed approach, going over its main constituents. In Section 4 we go over the first prototype we implemented upon which all the future work will be based on. In Section 5 we outline a first use case scenario we intend to use as a first proof of concept in order to validate our ideas. The works related are presented in Section 6. We conclude in Section 7 with a discussion and future work which will be built upon from the ideas presented in the paper.

2. Software analyses to be supported

Software analysis is one of the key activities in software engineering as it allows to extract the most diverse and extensive information regarding a software system. The classic analyses have been for years the ones targeting models and source code [18]. In the last years many research groups have shifted their attention to software evolution and the whole established community of reverse engineering, reengineering, and program understanding has actually acknowledged that evolution is indeed the umbrella of their research activities.

Software analysis research can be divided in three main categories, with regard to what topic they address:

- Development: the extraction and/or the analysis of information about the development of software artifacts.
- Models: the extraction and/or the analysis of models representing different features and views of software artifacts.
- Code: the analysis of software artifacts’ source code to extract information and assess properties as well-formedness quality, correctness, etc.

There is a plethora of research on these topics, but it is not in our intention to give a complete picture of the state of the art. We just want to give a brief overview of the current analysis techniques to setup a service platform for software analyses.

Approaches focusing on the software development either study its source code change history, bug history, its underlying dynamics or a combination of them. Fischer et al. [7] populated a release history database, combining information from version control and bug tracking systems, namely CVS and Bugzilla to facilitate further analysis. Draheim et al. [6] had a similar approach but only worked with version control data from CVS. Many other works detect and track changes made on the source code during the software project lifetime. Zou et al. [46] used origin analysis to detect merging and splitting while S. Kim et al. [24] used it to track function name changes. M. Kim et al. [23] focused just on code clone evolution and built a clone genealogy tool to extract code clones history from a project CVS repository.

Works by Zimmermann et al. [45] and Ying et al. [44] tried to predict future source code changes given past source revision history of a project stored into CVS repositories to then recommend potentially relevant source code for a particular modification task.

Source revision history is analyzed to extract also other kinds of information. Livshits et al. [30] combine that with dynamic analysis techniques to identify application-specific patterns and find pattern violation. Hipilat [40] forms an implicit group memory combining CVS source repository data, Bugzilla data, messages posted on developer forums and other project documents to recommend artifacts that are relevant to a particular task that a developer is trying to perform.

Gall et al. [11] extracted logical couplings of software modules by analyzing CVS data, in particular check in and check out time and the authors of those actions, and from that they were able to discover design flaws without analyzing a single line of code. Fluri et al. [8] focused on the extraction of several fine-grained source code change types and the assessment of their significance in terms of their impact on other source code entities and whether a they may be functionality-modifying or functionality-preserving. Then, Nagappan et al. [33] predicted defects density for a system using code churn metrics fetched from its change history.

Similarly to the works on source code change, bug analysis addressed extraction of data from a bug repository (as we already saw in [7]), its prediction or its analysis. For that, Hassan et al. [16] developed a dynamic cache of the ten mostly error prone subsystems (directories). Kim et al. [25] proposed a similar approach, but they dynamically cached the most likely fault prone source code locations. Sliwerski et al. [36] related version history and a bug database to detect, as Kim et al. [25], code locations whose changes had been risky in the past and annotated them with color bars to show their risk rate in Eclipse [5]. While much effort has been spent on software cost/effort prediction, very little has been done on bug fixing effort prediction. As for example the work by Weiss et al. [42] in which, for every new bug report in a issue tracking system, similar earlier reports are fetched and their average time is used as a prediction for the new one.

Not only the history of a software development pro-
cess has been addressed, but also its underlying dynamics. In particular, a lot of research has also been performed on the role of the developers in evolutionary processes. For example, Ćubranić et al. [41] and Anvik et al. [2] both developed approaches for bug triaging that recommend a set of developers with the appropriate expertise to solve a particular bug by applying machine learning techniques on bug reports fetched from a bug repository (in these cases Bugzilla). Mockus et al. [32] located people with desired expertise not using bug reports but by analyzing data from change management systems. Girba et al. [12] analyzed CVS logs to reconstruct code ownership to help in answering which authors are knowledgeable in which part of the system and also reveal behavioral patterns: when and how different developers interacted in which way and in which part of the system.

Most of the approaches focusing on models target the reverse engineering of a wide range of abstractions and forms of representations from software systems. For example, some work has been done to address UML models. Kollman et al. [26] and Tonella et al. [39] extracted UML Collaboration Diagrams by statically analyzing the source code, while Rounev et al. [35] extracted UML Sequence Diagrams. Some other works studied the runtime behavior of a software to recreate UML State Machine Diagrams [28, 38] or UML Sequence Diagrams [14, 28]. There is then a score of UML modeling tools that allow reverse engineering of Class Diagrams, from open source ones as Fujaba¹ (which offers also basic Activity Diagram reconstruction) and ArgoUML² up to commercial ones as Together³ and IBM Rational Rose⁴.

Most works target CVS repositories as there is a great deal of big and significant open source projects that use it (as CVS itself is opensource), thus giving researchers a huge amount of information that can be freely studied and analyzed, but as more and more projects are being now moved to SVN, we expect that also researchers will soon start to focus significantly also on it.

3. Our Approach

There is a huge variety of tools and techniques out there offering the most disparate analyses on a software system. Such analyses are currently offered on a purely local basis and have their own distinct input and output format making it impossible to combine and integrate them effectively. What follows is the description of how we tackle the problem.

Figure 1 gives an overview of our approach, which is made up by four main constituents: several software analysis web services, an analysis services catalog, an analysis broker and ontologies.

Software analysis web services are “wrappers” of already existing analysis tools exposing their functionalities and data through a web service.

The analyses catalog, as the name suggests, classifies all the registered analysis services with respect to a specific taxonomy we defined and stores other information about them.

The analyses broker acts as the interface between the catalog and the users.

Specific ontologies are used to define and represent the data consumed and produced by the different types of analysis, while upper ontologies define much more generic concepts common to several specific, ontologies, thus providing semantic links between them (otherwise they would remained decoupled).

In the following sections we explain in greater detail what these constituents are, what they do, and the benefits we can gain by using them.

3.1. Software Analyses as Web Services

Our solution proposes software analyses to be available as web services. We decided to leverage this paradigm as it is a well known standard and it was devised to overcome some of the problems we also face.

¹ http://www.cs.uni-paderborn.de/cs/fujaba/
² http://argouml.tigris.org/
and thus already offer many of the features we need, namely: language, platform and location independence and service composition.

Independence is achieved with the use of XML-based language to describe the services (WSDL [4]) and a simple, lightweight communication protocol (SOAP [13]) intended for exchanging structured information, formatted into XML-based messages, in a decentralized, distributed environment, normally using HTTP/HTTPS, which allows also easier communication through proxies and firewalls. Composition and orchestration is provided by BPEL4WS (Business Process Execution Language for Web Services) [20], an XML-based language designed to enable task-sharing for a distributed computing - even across multiple organizations - using a combination of Web services. Using BPEL, a programmer formally describes a business process that will take place across the Web in such a way that any cooperating entity can perform one or more steps in the process the same way.

To share a software analysis three things would need to be done:

- **Write and publish a web service** offering the methods to perform that particular analysis and to fetch the results, formatted with the ontology specific to that analysis.  
- **Write an adapter** that calls the actual underlying tool, doing the necessary data format translations: from the web service input format (represented, as we explained, by a specific ontology) to the tool specific one and vice versa, from the tool output to the web service output, represented by the ontology defined for the specific analysis the tool is offering.  
- **Register the service** on the analysis catalog to make it available to anyone interested.

As it can be seen, the internal logic, the input and output formats used, the platform and language under which the original tool runs will remain hidden behind the web service not being a burden for interoperability anymore.

More specifically, these analyses are the extraction and storage on our Release History Database [7] of CVS, Bugzilla data and FAMIX model of software projects offered by our **Evolizer**\(^5\) platform and the ones offered by our **ChangeDistiller** built on top of that, namely, the extraction of fine grained source code change [9], and their classification based on their significance level [8].

### 3.2. The Analyses Catalog

The Analyses Catalog, as the name suggests, stores and classifies all the registered analysis services so that they can be automatically discovered, invoked and their results fetched. To do that, a clear and unique classification is essential, so, as a first step, we created a specific software analysis taxonomy to systematically classify the existing and future services.

![Figure 2. A condensed view of the taxonomy](image)

Figure 2 gives a condensed view, due to space limitations, of that taxonomy. This taxonomy divides all the possible analyses in three main categories based on what their main focus is: the development of a software system, the underlying models of it and the actual source code. Each of those categories is in turn made up of many other subcategories.

Software development analyses are further divided into those targeting the history (extraction, prediction and analysis of source code changes and bugs), the process (its dynamics and metrics, as the ones defined by Lorenz et al. [31] and Nagappan et al. [33]) and the teams involved (their dynamics and metrics).

Model analyses are further divided into those targeting the extraction, either dynamic or static, of specific model representations (UML, FAMIX, call graphs, Rigi, etc.) and those computing differences between two models, usually of two versions of the same system. Code analyses, being the oldest and thus most studied topics of this taxonomy, are by far the most numerous and thereby are further divided into many other categories, as for example those checking code well-formedness, its correctness and its quality.

We will not go into the details of this part for space limitations and because it is still a work in progress and the taxonomy is not yet stable and complete. The only part that is already stable is the one about code quality and in particular its subcategory containing analyses focusing on the design quality. We decided to first focus on this area as it is the most used in the field of software evolution analysis as it can show whether and how the quality of the system being studied evolved.

Tools belonging to this category are, for example, the

---

\(^5\)http://seal.dif.uzh.ch/evolizer/
ones extracting and analyzing design metrics, as defined by Lanza et al. [27] and Lorenz et al. [31], and code smells, as defined by Fowler et al. [10], such as code clones detectors and predictors.

This proposed taxonomy is obviously not the only one possible and by no means complete, so it is most likely that some parts will be added on the way and some others will be modified. But the proposed categories are reasonable enough and make sense, in particular from the perspective of a user who wants to find some particular analyses without struggling with many and sometimes obscure categorizations but at the same time wants them to be expressive and meaningful. Since, to our knowledge, the literature lacks any preexisting taxonomies of this kind, we structured it mainly using the currently existing approaches as a blueprint and so that they would “fit” reasonably well into that.

We chose to implement the whole taxonomy as an ontology, more precisely in OWL [37], for three reasons.

1. We can achieve a formal representation of a set of concepts within a domain and the relationships between those concepts.

2. It is possible to reason about the properties of that domain and infer additional information based on the data explicitly provided.

3. Together with OWL we can use languages as SPARQL [34] to effectively query instances of the ontologies and fetch the services we are interested in.

In this way, the catalog is just an instance of an ontology, so it essentially comes down to an .owl file, which could then be published on the Internet to be accessed and queried by anyone who is interested, without the need of a web service to access it. But, since on top of that we wanted to offer other useful and more complex functionalities, we decided to make accessible through what we called the Analyses Broker.

3.3. The Analyses Broker

The Analyses Broker acts as a “layer” between the catalog and the users through which they can query, update, manage the catalog (namely register, update and unregister analysis services) and expand the taxonomy, as the one proposed is not supposed to be complete and new types of analyses that were not yet classified, or some modification to the already existing classification, could come up in the future.

Moreover, we can also offer more complex functionalities such as automatic composition of services. So, for example, if a user wants a series of analyses to be done on a project, the Broker would take care of finding, composing, executing them (for example with BPEL) and then just returning the final results to the user.

3.4. Ontologies

The large majority of the analysis tools is being used within institutional boundaries by single researchers who often are also the tool authors. The results of these tools are stored internally and are not accessible to third parties for the combination or integration of the results. Several researchers have pushed for common interchange formats such as GXL (Graph eXchange Language) [43] or XMI [1], but their efforts have remained largely unheard. The MSR (Mining Software Repositories) community is striving for integration especially in their Mining Challenge track, but it is limited to the application of the analysis tools on the same case studies. The integration and combination of results, especially of different kinds of analyses, remain completely open and is the major challenge we need to tackle as its solution is one of the main motivations behind our work.

A promising alternative to solve those problems is to use ontologies, in particular OWL, to represent both results and input. With an ontology we define and enforce how the results (their structure and internal relation) of analyses belonging to a certain type would be. So any new service would have to support inputs and provide results conforming to the ontology defined for the specific analysis it offers. Furthermore, it gives us a sound and well known data format to use and the ability to share that data between different types of computers using different types of operating system and application languages, as it is written in XML.

But what really makes OWL stand out and worth using are the properties related to its ontological nature: (1) heterogenous domain ontologies can be semantically “linked” to each other by means of one or more upper ontologies, which describe general concepts across a wide range of domains. In this way it is possible to reach some semantic interoperability between a large number of ontologies accessible “under” some upper ontology; (2) with OWL Description Logic foundation it is possible to perform automatic reasoning and derive additional knowledge; (3) we can use a powerful query language such as SPARQL or its extension iSPARQL [21], that uses similarity operators to query for similar entities; and (4) in contrast to XML and XQuery [3] that operate on the structure of the data, OWL treats data based on its semantics. This allows for an extension of the data model with no backwards
compatibility problems with existing tools.

4. The First Prototype

Having described the infrastructure, we will now show the main features of the first prototype of the Analysis Broker that we have developed. We decided to develop the Analysis Broker at first as it constitutes the foundation upon which everything else will be based on. In fact it is where all the future analysis services will be registered and through which a user or a tool would find and fetch services of interest.

More precisely, the Broker can be queried to get the content of the analyses catalog (in other words, the registered analyses) and if one or more specific analyses have been performed on some projects. We decided to offer just these two functionalities because in our opinion those two pieces of information are everything a user might want to know in this context, furthermore any additional information can then be fetched from a combination of them.

Those two queries are offered through a web service interface and the results formatted into a standardized machine readable format, more precisely OWL. In this way tools of any kind can (semi-)automatically fetch the analyses they need to then call them. However, this makes the results hardly readable by humans. So we chose to let the Broker be queried in the same way also through a website, which will format and present the results in a much more human understandable form. Therefore also here we will show the Broker functionalities through its website interface.

Figure 3 shows the initial view that will be presented to the user.

The user can do two things, either navigate through the catalog or query it. With the navigation option he/she can get an idea on the analysis taxonomy structure or see what the analyses being offered are. With queries, more specific information for the successive invocation of the services can be gathered.

Figure 4 shows what the Broker returns when queried for the currently registered analyses which is essentially the current instance of the catalog. So for every service is reported the name, the address through which it can be invoked and the type of analysis offered. Knowing the latter gives the user all the information on the service input and output. In fact, as we explained in Section 3.4, every analysis type is associated with ontologies to which the input and output of every service offering that analysis must conform.

Thus with this query it is possible to know what analyses can be performed and gather all the information needed to then call the ones that are of interest.

So it will be used when a user or a tool, given a project, wants to conduct some analysis.

Figure 5 shows what the Broker returns when queried to get if one or more types of analysis were performed on for some specified projects. Note that for all the projects is displayed whether or not every single requested analysis has been already performed, without explicitly showing what is the actual service that did it.

In fact, as long as it is performed, it does not really matter who performed the analysis since, as we explained before, all the services offering it will comply to a common input and output. Nevertheless the address of the actual service offering the analysis is simply hidden by the html representation behind the “check” symbol. So it can be immediately invoked to get the available data without having to query the Broker for any other information.
Figure 5. Broker list of analyses and projects

All this provided information is useful to see what data about a project is already available to then fetch it or trigger the analysis to produce it. Furthermore, it can be handy for tools and users that need case study data from existing projects to then run their own analysis. For example a tool extracting some newly defined software project metrics might need CVS history data of software projects for case studies and proofs of concept for validation. So, instead of finding suitable projects and extracting their CVS data by itself, it could take advantage of the previous analyses and thus just fetch the data that has already been extracted by the registered services offering CVS data extraction.

5. Validation

After having finished developing our prototype, we need to study and prove the potential of the Analysis Broker, the semantically enhanced catalog of software analysis tools and the usage of ontologies to represent and share analysis data.

Figure 6 sketches the scenario we have built for that first proof of concept, showing our solution at work.

We have created a test tool that plays the role of a development process analyzer which, given CVS versioning history data described by a CVS history ontology, extracts team metrics such as: the number of developers involved in each file, the total number of commits for each developer, etc. After that it returns the results to the user and stores them for future use by other tools.

In order to do that, our tool first queries the Analysis Broker to see if CVS history data of the chosen project has already been extracted by some service. If not, it will query again the Analysis Broker to get a list of the registered CVS history data extraction services and then ask one of them to carry out the extraction.

On the other hand, if the project history has already been extracted, it gets that data and performs standard predefined OWL queries and reasoning to get the information it needs to extract the wanted metrics. In fact, since all the services offering CVS history data have to represent and format the data they provide with a standard ontology it does not matter where the data comes from.

Once the metrics are computed, they are returned to the user and stored so that if the analysis was registered in the catalog and a proper web service to access it exist, other tools or people can retrieve that data.

6. Related Work

The use of web services and ontologies in connection with software analysis and evolution has, to the best of our knowledge, been addressed only recently by only a few researches.

A few works have addressed software analysis data and concepts representation with ontologies. Hyland-Wood et al. [17] presented an OWL ontology of software engineering concepts (SEC), including classes, tests, metrics and requirements. Happel et al. [15] in their KOntoR approach stored and queried meta-data about software artifacts to foster software reuse. What is interesting for us is that they proposed various ontologies to provide background knowledge about software components, such as the programming language and licensing models. Both works could be really valuable for us as upper ontologies, since they could provide us general concepts common to many specific software
analyses ontologies, so that they could be semantically “glued” together.

Highly related to our approach is the work by Kiefer et al. [22], which proposed EvoOnt, a software repository data exchange format including software, release and bug related information based on OWL. To effectively mine software systems represented in that OWL format and find, for example, code smells, they introduced iSPARQL, a query engine supporting similarity joins. From their work we borrow the idea of using ontologies to represent software analysis data to facilitate data exchange and automatic reasoning.

To the best of our knowledge Jin and Cordy [19], with their Ontological Adaptive Service-Sharing Integration System (OASIS), are the first and only researchers that so far studied an ontology based software analysis tool integration system that employs a domain ontology and specially constructed external tool adapters. They also implemented a proof of concept with three diverse reverse engineering tools that allowed them to explore service-sharing as a viable means for facilitating interoperability among tools.

We share with them the overall concept, but the two approaches have many differences as they have partially different goals. In fact, the objective of their integration effort was to allow the functionality/analysis available in one tool to be applied to the fact-base of another one. For this reason, they used an ontology just to describe the set of representational concepts that the different tools to be integrated might require and/or support. On the other hand, as we already showed, we intend to exploit ontologies on a much broader scale: to catalog and describe the services, to represent and standardize their input and output accordingly to the type of analysis offered, to semantically link different results and to perform (semi)-automatic reasoning on them.

Moreover, to overcome language, platform and location dependencies, we expose the functionalities of the different tools through web services, while they use not better specified ad-hoc adapters.

7. Conclusions and future work

The combination and integration of different software analysis tools is a challenging problem an engineer faces when he/she needs to gain a deeper insight on a software system and its history. For every required analysis a specialized tool, with its own explicit or implicit meta-model dictating how to represent the input and the output data, has to be locally installed, configured and executed. Even if different analyses of the same kind exist, the only way to compare them is to do it manually.

In our opinion the combination of ontologies and web services we presented in our approach can be extremely valuable to solve that problem. Using web services to expose the functionalities offered by the analysis tools gives us total independence in terms of platform, language and location and the possibility in the future to explore the use of well known mechanism of composition and orchestration (e.g. BPEL4WS) of several analysis services. OWL ontologies specific to distinct types of analyses allow us to have standard formats to define and represent the data consumed and produced by the analysis services, which can then be integrated with each other thanks to semantic “links” provided by generic, upper ontologies. In addition to that, thanks to OWL’s powerful query language (SPARQL) and its Description Logic foundation, data can be extracted and additional knowledge can be inferred with existing tools.

The purpose of this paper was to provide the foundations upon which subsequent improvements and implementations will be based on. With our use case we previously introduced, we will first validate and prove the potential of all features of our first prototype of the Analysis Broker: the semantically enriched catalog of software analysis tools and the usage of ontologies to represent and share analysis data. This phase will also help us to find weaknesses, refine and stabilize our approach and the prototype.

This is crucial because from there want to add other analyses to our catalog (possibly coming from other research groups), such as clone detectors and change predictors, to assess the feasibility and usefulness of the integration of results coming from different analyses. Furthermore, to make that possible we will also create and add upper generic ontologies that will semantically “glue” together the specific ones defining the tools output. We plan to re-use some of the existing ideas, as for example the software engineering concepts (SEC) ontology [17].

These are the first two main goals we aim at, as we hope that having a sound solution of proven usefulness would push other research groups to share their analysis approaches through our platform, making it really valuable. However, there are also many other ideas we want to explore in the future. Such as automatic recommendation of services given specific user analysis needs and automatic composition and orchestration of services so that a user can choose a sequence of analyses he wants to be performed on a software system, have it automatically executed and be presented with the final results.

Finally, we are convinced that by allowing disparate
analysis tools to collaborate with each other and share their information via a service platform can be highly beneficial. Not only it would enhance and speed up the work of a software engineer by giving him/her access to a big amount of information available without the need to install several tools and to cope with many output formats, but it would also promote the uncovering of new meaningful and interesting metrics and information deriving from the most diverse types of analysis that can finally “talk” to each other.

References


10