Use cases for matching semantic annotations of API operations

Michał Rój

Institute of Telecommunications,
Warsaw University of Technology
mroj@tele.pw.edu.pl

Semantic annotation of application programming interfaces (APIs) can enable various functionalities useful for software engineers. In this paper we discuss selected applications for semantically annotated API operations; we focus on applications which exploit matching of API operations together. We describe several use cases which are aimed for both API users (e.g., finding desired API operations or checking API portability) and API designers (e.g., checking APIs for redundancy or completeness). The paper outlines a possible implementation of the approach and discusses its advantages and limitations.

1 Introduction

Application programming interface (API) is a source-code level interface that allows applications access functionality exposed in an abstract way and provided by entities such as operating systems or libraries.

We can observe an increasing number of APIs nowadays. Examples include new APIs for mobile devices or Web APIs. Number, size and complexity of current APIs result in a relatively high amount of effort needed to get familiar with APIs and start using them. Existing tools such as IDEs address such problems only in a limited way.

To facilitate programming with APIs, multiple research activities have been initiated (e.g., [1-3]). The activities have shown interesting results in supporting programmers (e.g., they ease the process of searching for example code snippets or software artifacts of specific functionality).

Available approaches employ a variety of technologies. Semantic technologies, one of the options here, have proven to be useful in tasks such as API artifact search for reuse or application composition. The technologies can improve the process of API artifact discovery (e.g., in large API libraries) by using logic-based inference, enabling users with precise vocabulary to be used for building queries or translating user queries into meaningful domain-specific expressions. A prominent example (and one of the earliest approaches of this kind) is the LaSSIE system [4].

In contrast to typical applications of semantic technologies, we postulate a very simple communication model between a tool which supports programmers in API-related programming tasks and its user. We assume that the user points only the name of an API artifact (or a set of artifacts) and then runs a specific tool's functionality; no

domain-specific vocabulary is presented to or required from the user. Our motivation is to make the user interface possibly simple and intuitive. In addition, such simplicity can enable easier integration with software development tools.

The API artifact type we focus on is *API operation*[1]. In our approach we assume that API operations are semantically annotated and the annotations express the API operations' core functionality (i.e., they describe what the API operation do).

The key technique used to implement our approach is matching two API operations. The matching result can be one of the following (we assume that there are two semantically annotated API operations, *opA* and *opB*):

(1) both *opA* and *opB* do exactly the same thing,

(2) *opA* offers wider functionality than *opB* (i.e., *opA* covers fully the functionality offered by *opB* and extends it with new elements),

(3) *opA* offers narrower functionality than *opB* (opposite to case 2),

(4) no meaningful relationship is discovered.

We assume that matching of operations from different APIs is possible.

This paper describes several use cases which are available if relationships between API operations (as described above) can be detected. The identified use cases include the ones aimed for API users (finding desired API operations or checking API portability) and the ones aimed for API designers (checking API redundancy and completeness).

2 Related work

A typical use case for semantically annotated APIs is discovery of existing artifacts for reuse. This is often made by letting the user compose a query using domain-specific terminology acquired from an ontology. This approach is described, e.g., in [4-6]. Another use case is composition of applications from elements distributed across the network. Such approach is described, e.g., in [7].

Literature positions known to the author do not describe use cases exploiting matching descriptions of API operations together. The conceptually closest idea to the one postulated in this paper is described in [5]. Semantic annotation of API operations are used to sequence API operations in an appropriate order within an application under development. In [5] not whole API operations' functionality is described but API operations' parameters and return values. Parameters are matched with return values and when they 'fit', a sample code sequence is automatically generated, where API operations that return a specific kind of information precede the ones that use the information as parameters.

[1] In *API operations* we understand those API artifacts which are used by programmers to communicate information and initiate behavior. Depending to a programming environment, API operations are called differently, e.g., 'subroutines', 'functions', 'operations', 'methods', 'commands', etc. Examples of API operations are the `printf()` and `MoveFile()` functions, the `mkdir` command or the `File:canExecute()` method.

3 Use cases

In this section we present a list of generic use cases that exploit matching of API operations. The use cases have been developed by the author as a result of research and experiments with semantically annotated API operations. We do not claim that the selection of use cases is complete, however, they express a range of different (and possibly inspirational) functions.

In the use cases we assume the existence of a tool which is able to collect input data from its user, match API operations and display results (in Section 4 we outline implementation ideas of such a tool). In all use cases the tool's input and output data is a name (or names) of API operations or a name (names) of whole APIs.

The use cases are of two groups. First, use cases UC1-UC5 are aimed for API users (those who use existing APIs). Second, use cases UC6-UC7 are aimed for API designers (those who design/develop new APIs). In should be noted, however, that the described use cases are not strictly tied to a specific group of users and, slightly modified, can be used for different purposes than proposed below.

UC1: Finding functionally similar API operations in another API

Assume that an API user knows API X_{API} better then API Y_{API}. However, API Y_{API} must be used in the current task. The API user is provided with means to make the following query "Find me an API operation in API Y_{API}, which provides the similar functionality as API operation *O* in the API X_{API}." As a result of this query execution, the user is provided an API operation (or a set of API operations) in Y_{API} which have the similar functionality to the *O* API operation.

UC2: Reducing API diversity in an application

This use case is about checking the source code and detecting if the code uses multiple APIs from the same functional domain (for example the API user mixes *Windows* memory allocation functions with *C standard library* memory allocation functions). If so, the API user is proposed to 'unify' the code to use a single API; possible changes in the code are proposed.

UC3: Finding API operations of wider or narrower functionality in the API

This use case is about finding all API operations, which have wider (or narrower) functionality comparing to the one pointed. Finding 'wider' functionality can be useful in situations where the existing functionality of a program needs to be extended (e.g., until now the program uses an API operation what deletes directories, but a future version will need to delete both files and directories). A 'narrower' API operation can be useful if the goal is to make the program code as safe as possible. For instance, if files are the only type of entities to be deleted, the used API operations should not be able to delete other kinds of entities (such as directories).

UC4: Semantic comparison of two APIs

This use case is about semantically comparing two APIs. The comparison is made by matching the functionality of all API operations in the API. A report is generated as the result of the comparison, including, for instance, API operations which are present in one API and not present in the other API.

UC5: Checking if a program can be ported to another environment

This use case is about checking if a program written for one programming platform (represented by a set of APIs) can be ported to another programming platform. The checking is based on the semantics of API operations used in the code – if all used API operations have their semantic counterparts in the target platform (a set of APIs), the program is considered 'portable'. An extension of this use case can be a generation of a list of 'portable platforms' from a set of known programming platforms. In addition, a report can be generated containing the proposed API operation mapping from the source platform to the target platform.

UC6: Detecting 'semantic redundancy' in an API

This use case is about detecting 'redundant' API operations (i.e., API operations with shared functionality with other API operations). Such analysis can be made in order to make API more consistent (e.g., to redesign the API to limit its redundancy).

UC7: Checking 'semantic functional completeness' of an API

This use case is about checking if the API is 'functionally complete', i.e., if it supports all the functions typically available for a given API domain (otherwise the API can be called 'functionally incomplete'). This use case can be used in order to detect and possibly update an incomplete API by adding the missing API operations.

4 Proposed implementation

Our proposed implementation requires the following elements: (1) a domain-specific ontology which contains the terminology that can be used to express the semantics of API operations, (2) a set of descriptions of API operations' functionalities (i.e., semantic annotations of API operations).

A domain-specific ontology contains different categories of basic domain-specific concepts (e.g., *actions*, such as '*opening*', '*closing*', '*deleting*' and *objects*, such as '*file*' or '*directory*'). Semantic annotations of API operations are *classes* described in terms of concepts from a domain-specific ontology; the concepts are combined together in order to precisely express the API operation's semantics. For instance, by combining the action '*deleting*' with object '*directory*' and attribute '*empty*', the

semantics of *an API operation used for deleting empty directories* can be expressed. By combining a relatively limited set of concepts in different ways, a high number of different API operations can be described.

In our experimental platform both ontology and API operation's descriptions are encoded in OWL. We use OWL class constructors, e.g., restrictions and intersection expressions, to build the descriptions (our approach for representing API operations is similar to the one used to express advertisements in [8]).

In Section 1 we described four result types for matching API operations. Matching API operations is implemented by means of matching OWL classes representing API operations semantics. We perform two kinds of checking: if two classes are equivalent or if two classes are in a class-subclass relationship. Those simple matching methods have proven to serve well for service matchmaking [8] and are typically available from ontology processing and reasoning engines (such as Jena [9]). We implement API operation matching by using OWL class matching as follows:

(1) 'both API operations do the exactly same thing': the annotation class for *opA* is equivalent to the annotation class for *opB*,

(2) '*opA* offers wider functionality than *opB*': the annotation class for *opB* is a subclass of the annotation class for *opA*,

(3) '*opB* offers wider functionality than *opA*': the annotation class for *opA* is a subclass of the annotation class for *opB*,

(4) 'no meaningful relationship is discovered': none of the above holds.

Implementation of use cases presented in Section 3 with the above matching methods is relatively easy. We assume that for a specific API operation's identifier (e.g., '`printf()`'), its semantic annotation (i.e., an OWL class) can be retrieved. Similarly, for a given semantic annotation, the API operation's identifier can be retrieved. As said earlier, when interfacing with the user, only the API operations' identifiers are used.

UC1, UC3 and UC6 are implemented in quite a similar way. For instance, UC3 is implemented by simply comparing the source semantic annotation of API operation to all semantic annotations of API operations in the target API (which is the same as the source API). We check if they are superclasses (have 'wider' functionality) or subclasses (have 'narrower' functionality). UC1, in turn, checks the two above relationships and also the 'equivalence' relationship. The target API in UC1 is different from the source API. UC6, similarly, discovers all the relationships checked in UC1 but the source and target APIs are the same.

In UC4, we check for each API operation in the source API if there are API operations of 'exactly the same' or 'wider' functionality in the target API and vice versa. The results are used to prepare a report where the APIs are contrasted.

Implementation of UC2, UC5 and UC7 is more complicated[2] and is not discussed here. However, in all cases the core technique is matching OWL classes (semantic annotations of API operations) together for being in equivalence or super/subclass relationships.

[2] UC2 and UC5 require code analysis procedures while UC7 requires an API to be matched against a 'reference complete API'.

5 Discussion, summary and conclusions

This paper presents a set of use cases for matching semantic (ontology-based) annotations of API operations. In the use cases, the ontology content is not revealed to the programmer and all the semantic representation's complexity is hidden behind the façade of simple relationships between API operations. A distinguishing feature of the approach is that is allows operating on both single API and multiple APIs, which, in our opinion, allows novel and original functionalities.

Our approach uses intuitive notions of 'doing the exactly same thing' or 'offering wider/narrower functionality' when applied to API operations. We showed that these relationships can be easily expressed with simple ontology classes matching techniques (checking for semantic equivalence and subsumption).

Our preliminary results with an experimental domain ontology with a set of semantic annotations of API operations from three different APIs and a prototype implementation of selected use cases (UC1, UC3 and UC6) are encouraging. We observe new means of gathering knowledge from analysis of both APIs and source code, which might be useful at different stages of application/API development.

The approach presented in this paper has the following limitations. First, we focus on detecting if two API operations 'do exactly the same thing' and 'offer wider/narrower functionality'. In some cases, however, detection of other kinds of relationships might be useful (e.g., that two API operations 'offer a shared functionality'). We plan to investigate those cases further, which might result in a revised set of use cases.

Second, the applicability of some use cases is limited by different programming abstractions available for different platforms. A challenge is, for instance, to develop inter-platform domain-specific ontologies. We see that it can be at least problematic for some domains (for instance, the UNIX file abstractions are different from the Windows file abstraction in many points).

Finally, our proposed approach requires both development of quality domain-specific ontologies and ontology-based descriptions of API operations. This is a costly task, which should be justified by profits from using the approach. Automation of the ontology/annotation generation process might be an important factor here.

References

[1] Bajracharya, S., J. Ossher, and C. Lopes, *Searching API usage examples in code repositories with sourcerer API search*. Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation. 2010, Cape Town, South Africa: ACM. 5-8.

[2] Hummel, O., W. Janjic, and C. Atkinson, *Code Conjurer: Pulling Reusable Software out of Thin Air*. IEEE Software, 2008. **25**(5): p. 45-52, IEEE.

[3] Stylos, J. and B.A. Myers, *Mica: A Web-Search Tool for Finding API Components and Examples*. Proceedings of the Visual Languages and Human-Centric Computing. 2006: IEEE Computer Society. 195-202.

[4] Devanbu, P.T., R.J. Brachman, P.G. Selfridge, and B.W. Ballard. *LaSSIE: a knowledge-based software information system*. in *Proceedings of the 12th international conference on software engineering*. 1990. Nice, France: IEEE.
[5] Eberhart, A. and S. Agarwal. *SmartAPI - Associating Ontologies and APIs for Rapid Application Development*. in *Ontologien in der und für die Softwaretechnik Workshop anlässlich der Modellierung 2004*. 2004. Marburg/Lahn.
[6] Oberle, D., S. Lamparter, S. Grimm, D. Vrandecic, S. Staab, and A. Gangemi, *Towards ontologies for formalizing modularization and communication in large software systems.* Applied Ontology, 2006. **1**(2): p. 163-202.
[7] Kazakov, M.L. and H. Abdulrab, *On aspects of software integration based on logical inference.* Applied Mathematics Bulletin of Russian Academy of Science, 2003. **4**: p. 71-78.
[8] Li, L. and I. Horrocks, *A software framework for matchmaking based on semantic web technology.* International Journal of Electronic Commerce, 2004. **8**(4): p. 39-60.
[9] Carroll, J., I. Dickinson, and C. Dollin. *Jena: Implementing the Semantic Web Recommendations*. in *Proc. of World Wide Web Conference*. 2004. New York, USA.