A Python Library for Provenance Recording and Querying

Carsten Bochner, Roland Gude, and Andreas Schreiber

Simulation and Software Technology
German Aerospace Center
51147 Cologne, Germany
{Carsten.Bochner,Roland.Gude,Andreas.Schreiber}@dlr.de
http://www.dlr.de/sc

Abstract. In many application domains the provenance of data plays an important role. It is often required to get store detailed information of the underlying processes that led to the data (e.g., results of numerical simulations) for the purpose of documentation or checking the process for compliance to applicable regulations. Especially in science and engineering more and more applications are being developed in Python, which is used either for development of the whole application or as a glue language for coordinating codes written in other programming languages. To easily integrate provenance recording into applications developed in Python, a provenance client library with a suitable Python API is useful. In this paper we present such a Python client library for recording and querying provenance information. We show an exemplary application, explain the overall architecture of the library, and give some details on the technologies used for the implementation.

1 Introduction

The recording and analysis of the provenance for data (i.e., a suitable documentation of the process that led to the data [1,2]) resulting in IT based processes in science and engineering gets more and more important. Such kind of documentation is required or wanted in a variety of application domains, for example in aerospace engineering, medical applications, climate research, or other e-science applications. In some cases, the detailed documentation of processes that led to certain data is required by official regulations.

Since the recording of provenance gets more important in many application domains, it is essential that the overhead for adding the ability to record and query provenance from within application must be as low as possible. This is important to reduce the effort needing to enable existing application for provenance recording which might convince more developers and even scientists to do so. Therefore the availability of suitable libraries for commonly used languages with a high-level API is useful.

As an example, in science and engineering the high-level programming language Python [3] is being used in more and more complex applications. These

J. Freire, D. Koop, and L. Moreau (Eds.): IPAW 2008, LNCS 5272, pp. 229–240, 2008.

applications from industry, research labs, and universities are either implemented in Python completely or are provided with an Python programming API for controlling, extending, or embedding them. Especially, scientists in the fields of mathematics, physics, or engineering are often not interested or even willing to learn and use modern object-oriented languages such as Java, C++, or C# just to configure or extend existing applications.

Having a Python API and a correlative client library implementation for provenance recording and querying has two major fields of application. The first is, to enable existing applications written in Python with provenance recording and, as a special case, the customization of provenance recording on end-user level (e.g., by using an embedded Python interpreter). The second use case is the rapid development of (simple) tools for analyzing already recorded provenance information.

The rest of this paper is organized as follows. In Section 2 we will explain the application context of our work and discuss the advantages of the Python language. In Section 3, we present an overview and the architecture of the Python Provenance client-side library and in Section 4 we give some details on the implementation. Finally, in Sections 5 and 6 we present future work and conclusions.

2 Motivation

Currently, more and more scientific applications are being developed in Python or provided with a Python API. Examples are computational codes written in languages such as C or Fortran with Python APIs for convenient integration in working environments and steering the computation, simulation environments which integrates numerical codes, data management systems which manages input and output data of computation, or Grid environments which are able to distribute computation on a wide range of computing resources. In many cases, computational intensive parts of application codes are still written in C, C++, or Fortran whereas Python is used as a very high level language for configuration of these code, the setup of the overall computing workflow, or for managing the involved data. Very often these applications are multidisciplinary coupled computations, where simple Python scripts are used to implement the coupling scheme for steering the computation.

An example, where provenance-enabling Python applications is important is the German national D-Grid community project AeroGrid [4]. In AeroGrid, a Grid infra-structure for the aerospace research community is being created. The goal of the AeroGrid project is to provide a productive Service Grid for researchers from industry, national research labs, and universities who are collaborating in design and simulation phase of future products. In AeroGrid, the basic grid middleware services of the D-Grid infrastructure (The Globus Toolkit and UNICORE 6) are being used.

One of the project objectives is to enable the user interfaces and the infrastructure to record provenance information about the involved data. The integration of a provenance service means calculations performed in AeroGrid are automatically documented and traceable. Recording provenance information, i.e. complete information on the individual processing steps applied to data, improves the reliability of results for AeroGrid users. For example, during the design of turbine engines many variants of design are being simulated by a number of design engineers using a variety of internal and external computing resources. To get reliable and traceable results for each data file of the different variants, user information of involved engineers, detailed information of used computing resources, changes in parameter settings, as well as information about the used simulation codes are being recorded in a provenance data base.

The main user interface in AeroGrid is the data management client DataFinder [5], a lightweight application software for managing technical and scientific data. It was developed to manage large amounts of data, and allows data to be stored using a number of different storage interfaces (e.g., WebDAV, FTP, GridFTP, Amazon S3, SRB, OpenAFS, or TSM). The structure of the data and descriptive metadata are stored in XML format on the central server and can be edited using the standardised WebDAV protocol. The DataFinder user interface consists of a platform-independent user client that allows users to navigate through the existing data, search for data, create and manage metainformation for all data, and execute scripts stored locally or on the server. The DataFinder client was developed in Python and the Qt GUI library and can be extended with Python easily.

Since the DataFinder is implemented in Python, integrating provenance recording can be done easily using a suitable provenance client library with a Python API.

The advantages of Python for applications in science and engineering are manifold [6,7], but it is an important prerequisite that Python is a general-purpose programming language without any restrictions and available on any platform with an ANSI-C compiler. It supports multiple programming paradigms (functional, object oriented, and imperative programming) and has many libraries and modules for a variety of tasks. But most important is the clear and highly-readable syntax which allows one to learn Python in very short time and which makes Python code very maintainable.

The rest of this paper presents a Python client library for recording and querying provenance information based on specifications developed in the EU Grid Provenance project [8].

3 Overview of the Python Library for Provenance

3.1 Fundamentals of the EU Grid Provenance Concept

The recording and querying of data and the interaction between Provenance-aware applications and a provenance store rests upon the following definitions [2]. The Provenance architecture uses an service oriented architecture (SOA) style, where a *provenance store* acts as Web Service for storing and querying the process information.

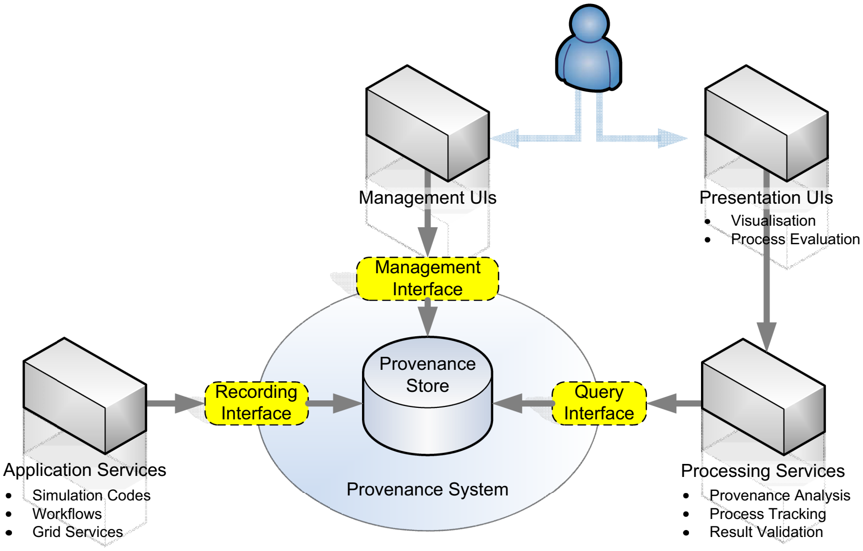

Fig. 1. Main concept of the EU Grid Provenance project

Every service involved in a data process is defined as an *actor*. The interaction between these actors is based on messages which take the form of SOAP messages for Web services. The interaction between actors characterizes the composition of the application workflow and is defined as a *process*.

In order to store and represent the provenance and process, different data models have been defined in the EU Grid Provenance Project. The elementary data model is the *p-assertion* which represent some step of process, while the *Process Documentation* data model represents the whole documentation of the process.

After the provenance of a process is recorded and stored in a provenance store a querying actor can ask for this provenance by sending a provenance query request to a provenance query engine, which is typically implemented by the provenance store [9]. Processing the query request the query engine must decide which p-assertions correspond to the request. Therefore the query must specify this factors and the provenance concept allows different search languages to do so. For executing queries a provenance store categorizes recorded p-assertions in a larger data structure, called the *p-structure*. This structure can be understood as a navigable schema or hierarchy of the provenance store. The p-structure is exposed to the querying actor through query interfaces of the provenance store. Figure 1 gives an overview of the main system concepts of the architecture defined in the EU Grid Provenance project.

A client-side library (CSL) is a collection of functions that allows applications to communicate with the services of the provenance store [10]. Furthermore a CSL should allow application developers to apply the rules of the provenance architecture easily. This should be enabled by a CSL that provides a clear and easy to use API with simple interfaces and data structures.

By developing a reusable library with an easy to use interface many details of this communication can be hidden from an application developer, thus reducing development cost of provenance-aware applications. Such a client-side library needs to implement the defined record and query protocols of the target Provenance store. In this case the client-side library mainly implements:

- **Process documentation recording protocol** - Definies the SOAP message communication between a recording actor and the provenance store [11]).
- **Query protocol** - Specifies the SOAP message communication between a querying actor and the the provenance store [9].
- **XPath protocol** - Definies the XPath-based profile of the provenance queries [12].

Furthermore, the client-side library needs to communicate with the provenance store using a defined technology binding (like the SOAP binding defined in [13]).

Currently two implementations of the provenance store are available. One store was developed by IBM for the EU Grid Provenance project and requires an installed Globus Toolkit. As the requirements are very extensive this store is not used in this project. The IBM store is a prototype implementation.

PReServ is the second provenance store and is developed by the University of Southampton. Currently prototype version 0.31 with WSDL 25 is available and used as reference store in this project [14].

3.2 General Architecture Overview

As the PReServ comes with a Java client-side Library [15], this Java CSL was used as a first reference for the design and implementation of the Python CSL. Therefore both architectures are mainly based on a similar layered model [15]. This model consists of three layers (see figure 2), with two API-layers offering interfaces for the server / the client, and one utilities or adaptions layer mapping this APIs to each other. The reason for this model is the complexity of the server API, which can not be used reasonable by application developers. Therefore the server API is mapped to a simpler client API, allowing application developers an easy access to the features of Provenance.

Client API. The top layer is the client API. This layer is exposed to the users of the library.

The client API offers interfaces to interact with the Provenance store, hold content of recordings (i.e. PAssertions) and queries. For querying the Provenance store currently the XQuery XML query language is supported, as defined in the PreServ XQuery interface [14]. Future versions of this Python CSL and the PReServ Provenance store might also support security, documentation style helper and policy helper.

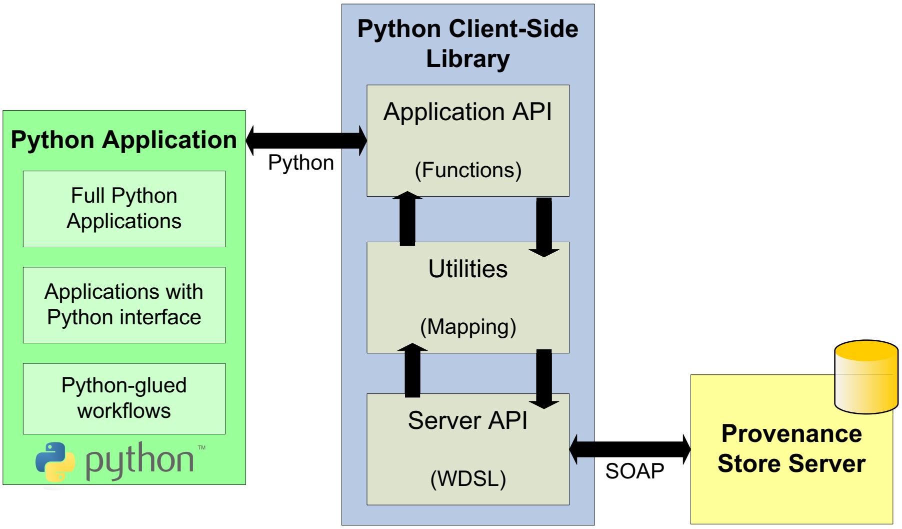

Fig. 2. Layer model of the Python CSL

The client API is used directly by application developers, which want to make their Python application provenance-aware. Hence the client API should contain simple and clear interfaces and data structures. Furthermore these interfaces need to be very robust and should not be subject to changes. The client API is defined using pyProtocols (see 4.1). The client API uses the utilities layer for mapping its services to the Service API.

Utilities. The middle layer is an adaption layer which maps the server API to the client API. This utilities layer or adaption layer implements the interfaces (or adapters) of the client API and contains several utility and helper classes for mapping the relatively simple client API to the complex server API. Furthermore several test modules have been integrated to ensure the correctness of the Python CSL implementation. The adaption layer consists out of pyProtocol adapters (see 4.1) which map a specific server APi to the client API.

Server API. The lowest layer is the server API. This layer directly communicates with a certain provenance store implementation. The interfaces of the Provenance store web services are defined in several WebServiceDefinition Language (WSDL) files. A WSDL uses the XML schema and can be understand as a contract between the service and the communicating component and mainly defines services, ports, operations and messages. The Python CSL contains Python stubs automatically generated from the WSDL files with the wsdl2python tool of the Zolera SOAP Infrastructure (ZSI) Python library. Currently the Server API module of the Python API consists mainly of four files:

- ProvenanceService_client
- ProvenanceService_server
- ProvenanceService_services
- ProvenanceService_types

The server API is specific for each supported provenance store implementation. Current files enable the communication with the PReServ Provenance store [14] according to the defined WSDL files of version 0.25.

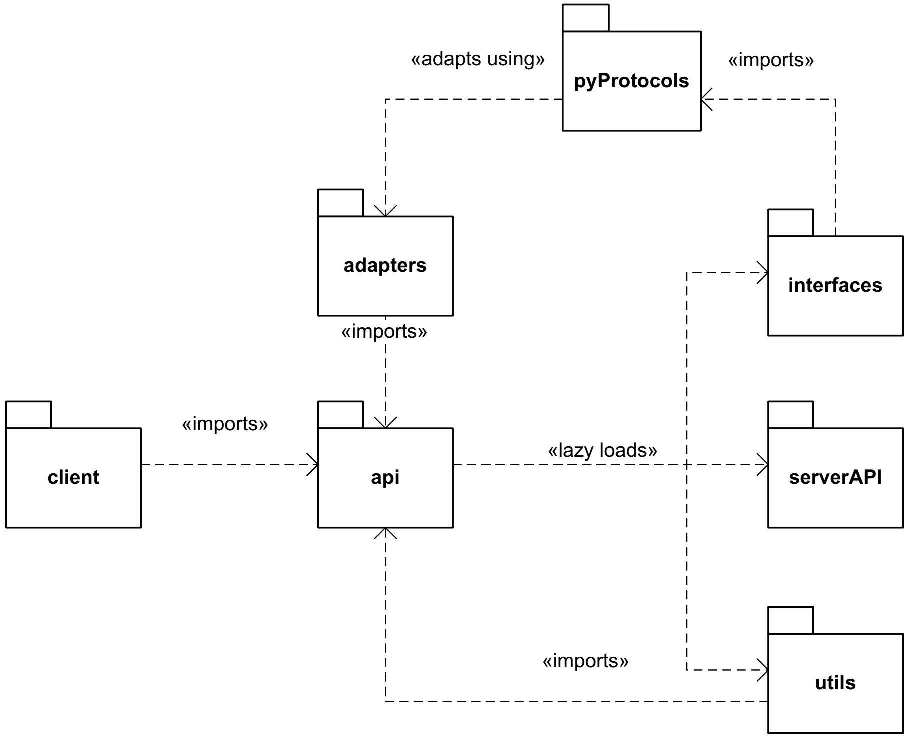

Fig. 3. Overview of the relation between the Python Provenance-CSL packages

3.3 API Description Overview

This section gives an overview of the API of the Provenance-CSL. See also figure 3 for the relations between packages.

- **Provenance -** Base package. It is only used as a collection of all the other parts.
- **Provenance.api -** This package contains the package users should utilize for their application. It is a collection of all parts of the Provenance-CSL which might be useful for a typical user.

- **Provenance.interfaces** - This package contains the interface-definitions for all types used in Provenance-CSL. Users will need this solely if they wish to define new adapters.
- **Provenance.adapters** - This package contains predefined adapters for several data types and interfaces.
- **Provenance.serverAPI** - This package contains the data types and stubs which are generated by wsdl2py. Users will usually not work with this package directly.
- **Provenance.utils** - Collection of utility functions which help with the generation of data which can be recorded on a Provenance store. If users do not create their own adapters for their data types, they should use these functions to create the correct data types for recording.
- **Provenance.client** - Contains the implementation of the Provenance store service client, which is the interface to the Provenance store and allows to store data on it or query it from there.

4 Implementation Details

This section briefly describes the used technologies, methods and materials which had a significant impact on the project. Furthermore it shows the main differences between the Java and the Python client-side library implementation.

4.1 Used Technologies and Methods

Zolera SOAP Infrastructure. The SOAP-bindings of the Provenance protocols have been used to communicate with Provenance stores. SOAP is a protocol for data exchange based on the eXtended Markup Language (XML).

The Zolera SOAP Infrastructure (ZSI) [16] is an implementation of SOAP version 1.1. A special feature of ZSI is that it comes with a Web Service Definition Language (WSDL) compiler wsdl2py, which generates Python stubs for the client-side of a web service. Since the Provenance protocols are defined using WSDL, this was an important feature. ZSI has been used to generate Python code from the WSDL definition of the Provenance protocols. It has also been used for all SOAP communication.

Python Enterprise Application Toolkit. The Python Enterprise Application Toolkit (PEAK) [17] is a collection of Python modules which adds useful features for component based design to Python. Its subpackages *importutils* and *pyprotocols* have been used to enable lazy loading and automated protocol adaption.

Lazy loading is a technique that allows the importing or loading of a library on demand. The *importutils* package of PEAK allows to define modules as *lazy modules*. It is completely compatible with the normal Python importing mechanism.

Unlike other object oriented programming languages, Python makes no use of anything like interfaces. In some object oriented programming languages (i.e. Java), interfaces are used to describe the methods a class has to provide in order to implement an interface. As an alternative concept, PyProtocols introduces protocols and protocol adaption to Python. Protocols are used to describe the behaviour of objects by defining which methods have to be supported and which members (i.e. variables, types) have to be provided in order to support the object. A really valuable feature of PyProtocols is the automatic adaption mechanism, which allows automated adaption from one data type *d1* which supports a protocol *p1* to support another protocol *p2* if an adapter from *p1* to *p2* has been defined.

An example of this adaption mechanism in the Provenance context is the following: Provenance records usually have a sink and a source. Both are complex types which usually contain a URL. By defining an adapter from strings which match the URL pattern to the complex type behind sink and source, it is now possible to use strings whenever the complex type is expected. In that case PyProtocols will automatically convert the strings to the expected complex type. This technique eases the usage of the generated code and the library.

For instance if a developer wants to use the Python CSL to store messages defined by his internal data types on a Provenance store, all he has to do is to define an adapter from his data type to the corresponding P-Assertion interface of the CSL. If he wants to record something on the store now, he can send the designated information to the store by simply supplying the Python CSLs recording API with instances of his data type. By defining adapters for the return types to his own data types, he would also be able to receive his own internal data types from a Provenance store using Provenance-CSL. Section 4.2 shows example code of how to define custom adapters.

4.2 Examples

The following source code shows examples for illustrating some basic provenance concepts.

Lazy-loading of all necessary modules and initializing a client-side access to a provenance store:

```
from provenance.api import *
cl = client.Client(http://localhost:8080, tracefile=sys.stdout)
```

Example of recording some complex provenance information (error handling code omitted)

```
viewKind = "isSender"
subj = utils.createSubjectId(1, dataAccessor, "parametername")
objlist = [utils.createObjectId(
    utils.createInteractionKey("http://sink", http://source"),
    pAssID, ’anything’, ’dataAccessor’, ’parameter’, ’isSender’)]
keys,response = self.cl.record([
        [utils.createActorState(a_content_0, doc_style),
         utils.createRelationship(subj, rel_type, objlist),
         utils.createInteraction(m_content_0, doc_style),
         utils.createInteraction(xml_content_0) ],
[utils.createActorState(a_content_1),
 utils.createActorState(a_content_2),
 utils.createInteraction(m_content_1, doc_style) ],
[utils.createRelationship(subj, rel_type, objlist)]
       ], viewKind, sink, source)
res = interfaces.IRecordAck(response)
```

Example of data querying using an XQuery expression

```
queryString = "for $n in $ps:pstruct return $n"
response = self.cl.query(queryString)
result = interfaces.IQueryAck(response)
```

Afterwards variable "'result"' contains an XML structure containing all pstructs available in the store.

The following code represents an example of extending data types using adapters. Here the adapter allows using type string when type address is expected.

Example of extending data types

```
class IAddress(IZSITypeCode):
    """ interface for string typecodes """
    def getAsString(self):
    """ returns a String with the Value of the Stringlike. """
    IString = protocols.protocolForType(basestring,[])

class AddressAdapter(object):
    protocols.advise(instancesProvide=[IAddress],
                     asAdapterForProtocols=[IString])
    def __init__(self, string):
        self._delegate = serverAPI.Address(string.__str__())
    def getAsString(self):
        return self._delegate.__str__()
    def toTypeCode(self):
        return self._delegate
```

5 Current and Future Work

5.1 Current State

The Python client-side library currently supports recording of P-Assertions on a PReServ Provenance store using the Provenance protocols of version 0.25. Querying the store using the XML XQuery language is possible. The concept of P-Headers has not been implemented yet. The current implementation features a complete set of utility functions for easy creation of P-Assertions and records and everything that is necessary for that. All interfaces which are necessary for recording have been defined as well as interfaces for the result types. Several interfaces have been defined to be context sensitive (i.e. is an Endpoint used as a sink or as source).

Adapters for all wsdl2py-generated types to the appropriate recording interfaces have been defined as well as adapters for a wide range of simple Python data types (like strings, lists and dictionaries) to support several recording interfaces. Adapters for the results of recording operations to the appropriate interfaces have been defined as well.

5.2 Future Work

Future works will focus on two main goals. First, current Python CSL will be adjusted to support upcoming new releases of the PReServ Provenance store and its changed WSDL definitions and functionalities. Therefore especially the package Server.API needs to be overworked, as this is always designed for a certain provenance store implementation. Further changes might become necessary, if the intent to change the current concept of an integrated XML database will be realised in the new PReServ release. In this case, the use of XQuery might become obsolete and a different query protocol must be integrated. Furthermore this process of adapting to a new PReServ version will be used for a redesign of the Python CSL and further quality assurance.

As the current version of the Python CSL is a proof of concept, not all protocols and functions of the provenance architecture [2] are supported. Therefore the second goal of our future work is the support of further functionality, including aspects as p-headers, security and different query protocols. This work will be subsequent to the fulfilment of our first goal and thus already support the new PReServ store version.

6 Conclusions

This paper presented a Python implementation of a provenance client-side library, which is currently able to record provenance information and to query provenance stores. The Python API as well as details on the implementation have been described.

Having this library eases the task to add provenance-awareness to existing or new Python applications. Especially, this includes applications written in other programming languages which have a Python API for extending or embedding. In particular, if the application has an embedded Python scripting functionality, end users could add provenance recording on their own or, at least, customize or extended existing provenance recording capabilities. Using the querying API, users can also use the Python CSL to rapidly develop specific analysis tools in Python.

Acknowledgment

This work has been supported by the German Federal Ministry for Research and Technology (BMBF) under Grant 01IG07006A.

References

1. Moreau, L., Groth, P., Miles, S., Vazquez-Salceda, J., Ibbotson, J., Jiang, S., Munroe, S., Rana, O., Schreiber, A., Tan, V., Varga, L.: The provenance of electronic data. Commun. ACM 51(4), 52–58 (2008)
2. Groth, P., Jiang, S., Miles, S., Munroe, S., Tan, V., Tsasakou, S., Moreau, L.: An Architecture for Provenance Systems. Technical report, University of Southampton (2006)
3. The Python Website, `http://www.python.org`
4. The AeroGrid Project Website, `http://www.aero-grid.de`
5. Schlauch, T., Schreiber, A.: Datafinder - a scientific data management solution. In: Ensuring the Long-Term Preservation and Value Adding to Scientific and Technical Data, PV 2007, Oberpfaffenhofen, Germany (2007)
6. Dubois, P.F.: Ten good practices in scientific programming. Computing in Science and Engg. 1(1), 7–11 (1999)
7. Jackson, K.R.: PyGlobus: a Python interface to the Globus Toolkit. Concurrency and Computation: Practice and Experience 14(13-15), 1075–1083 (2002)
8. The EU Grid Provenance Website, `http://www.gridprovenance.org`
9. Miles, S., Moreau, L., Groth, P., Tan, V., Munroe, S., Jiang, S.: Provenance Query Protocol. Technical report, University of Southampton (2006)
10. Jiang, S.: Client side library. Architecture tutorial. Technical report, University of Southampton (2005)
11. Groth, P., Tan, V., Munroe, S., Jiang, S., Miles, S., Moreau, L.: Process Documentation Recording Protocol. Technical report, University of Southampton (2006)
12. Miles, S., Moreau, L., Groth, P., Tan, V., Munroe, S., Jiang, S.: XPath Profile for the Provenance Query Protocol. Technical report, University of Southampton (2006)
13. Munroe, S., Tan, V., Groth, P., Jiang, S., Miles, S., Moreau, L.: A SOAP Binding For Process Documentation. Technical report, University of Southampton (2006)
14. The PReServ Website, `http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/SoftWare`
15. Jiang, S., Moreau, L., Groth, P., Miles, S., Munroe, S., Tan, V.: Client Side Library Design and Implementation. Technical report, University of Southampton (2006)
16. The Python Webservices Project Website (including ZSI), `http://pywebsvcs.sourceforge.net`
17. The Python Enterprise Application Kit (PEAK) Website, `http://peak.telecommunity.com`