Evaluation of aspect-oriented frameworks in Python for extending a project with provenance documentation features

Arne Bachmann
Deutsches Zentrum für Luft- und Raumfahrt
arne.bachmann@dlr.de

Henning Bergmeyer
Attensity Empolis Europe GmbH
henning.bergmeyer@gmx.de

Andreas Schreiber
Deutsches Zentrum für Luft- und Raumfahrt
andreas.schreiber@dlr.de

Abstract

In this paper we describe two sides of a real life use case of introducing an aspect-oriented framework into an industrial-grade project. This paper is divided into two parts: the selection process for an AOP framework in the Python programming language, and its use for modularized non-invasive recording of provenance data in a distributed data management tool. Criteria for the choice of such a framework are discussed and the background of provenance documentation is laid out.

Keywords*: Aspect-oriented programming, Aspyct, frameworks, provenance recording.*

1. Introduction

In most industrial disciplines, software systems play an import role for research, design, and production. Software is used by engineers for a broad variety of tasks, for example, for complex simulations or data management. Software systems, such as the data management tool *DataFinder* (Schlauch and Schreiber, 2007) are complex systems by themselves. The development of such tools usually involves usage of current software technologies, in our case aspect-oriented (AO) programming. Certainly, the main focus of industrial-grade tools is to support engineers and other users in their daily work in a reliable and convenient manner, but often they offer a chance to employ and evaluate recent methodologies and technologies from computer science and software architecture to stay in the vanguard of the market.

Aspect-oriented programming (AOP, Kiczales et al., 1997) has for at least a decade closed the gap between modular decomposition on object level and the lack of decomposition on system level by extracting many of system-wide cross-cutting concerns (CCC) into a single maintainable and modular entity. These are concerns that cannot be factored out without code scattering (repetition of code in many places) and code tangling (intermingling of

different domains' responsibilities in one place), because of the decompositional approach taken. If one was to take another approach to separate concerns in an object-oriented software system, it would lead to a potentially other stratification and interdependency of responsibilities, but inevitably have again CCC for any but the simplest systems to create. Thus AOP not only allows condensing orthogonal concepts into one code unit which simplifies testability, evolution and conciseness, it also allows for dynamic switching of complete features in a unified way and effortless way.

Although the object-based approach is in principle valuable for the decomposition of complex systems into smaller maintainable parts, the adoption of AOP is still hesitating in many industry areas. A reason for this may be that especially in highly standardized business environments optimized for throughput and reliability, there are already a lot of powerful frameworks (FWs) at hand that work around most pressing CCC and relieve the developer from writing scattered and tangled code by providing means like framework-managed dependency and data injection (IoC, inversion of control). This is often done by a modular configurational approach, where often-needed concerns like logging, persistence, error handling or security are already supported by the framework and extracted into settings choices for the developer, cf. e.g., *Java Platform, Enterprise Edition* (http://www.-oracle.com/technetwork/java/javaee/overview/), or where a conventionalist approach to component separation is mandatory, cf., e.g., *Ruby on Rails* (http://rubyonrails.org) for web development or *Griffon* (http://griffon.codehaus.org) for rich client applications. This again shows the need for the tackling of those recurring concerns and gives reason why powerful frameworks like for example the *Spring framework* (http://www.springsource.org) or the *Python Enterprise Application Kit* (http://peak.telecommunity.com) and similar exist and have thriven over many years.

Nevertheless those heavyweight enterprise FWs cannot close all decompositional gaps introduced by tangled code in object-oriented programming (OOP) and aren't always a reasonable or merely applicable solution, thus a "pure" academic AOP FW providing lightweight AO features might prove to be the best and inevitable solution.

In this paper we show an industry-strength use case where the adoption of an AO FW was not only the logical and fastest way to tackle the difficult problem of integrating provenance recording as presented here, but also the only feasible way for the constraints given which are, in this case, the programming language, the software environment and the distributed infrastructure.

This paper is organized as follows: Section 2 shows current research state in Python FWs for AOP and introduces criteria for the FW selection. A qualitative comparison between the FW candidates closes the section. Section 3 introduces the use case where we applied our selection process and presents integration of provenance features into a data management system by using an AOP Python FW. Section 4 draws the conclusion and gives and outlook to future work.

2. Frameworks

In this section we will provide an exemplary comparison of AOP frameworks for the programming language Python (http://www.python.org), which is the target language of the use case exemplification in the following chapter. By *frameworks* we mean those that provide aspect-oriented functionality for the core language, not FWs *using* AO approaches to support their work (e.g., the aforementioned enterprise FWs).

2.1. AOP in Python

Python is an interesting language choice because it seems to simplify and spur the development of a lot of FWs due to its dynamic character, but is by no means the only dynamic language in this regard, cf. JavaScript with AspectJS at http://www.aspectjs.com, jQuery AOP (http://code.google.com/p/jquery-aop/) and AOJS (Washizaki et al., 2009).

When laying out the decision-making fundamentals for finding a suitable AOP solution in Python, we will, for the sake of brevity, compare only those three FWs that seem to have been most active during the last year's time: Python Aspects (http://www.cs.tut.fi/~ask/aspects/), Aspyct (http://www.aspyct.org) and Logilab Aspects (http://www.logilab.org/project/logilab-aspects). There are a lot of other – mostly dormant – FWs available: Pythius (http://pythius.sourceforge.net) – inactive since 2002, AOPython (http://pypi.python.org/pypi/AOPython/1.0.3) – inactive since 2007, Spring Python (http://springpython.webfactional.com) – active, Transwarp/PEAK (http://peak.-telecommunity.com) – inactive since 2007; the PyPy project uses Logilab's AOP implementation (Rigo and Pedroni, 2007; Fayolle et al., 2007).

In this paper, we will concentrate our study on the comparison of the former three. The next paragraph gives an overview over typical application and use cases of AOP to prepare for the comparison of the FWs.

2.2. Typical cross-cutting concerns

The choice for an aspect-oriented solution in software development (SD) is guided by the need to reduce implementation complexity of certain CCCs, i.e., persistence, logging, tracing, monitoring, provenance, security, validation, contract enforcement, functional correctness checking, layered systems modelling (e.g., exception translation, layer-bypassing) and transaction handling. All these are concerns usually not connected with implementation on the component level, but have effect on a system wide scope, which explains the difficulties of implementing them in methodologies emphasizing smaller encapsulation levels like the decomposition in OOP.

2.3. Classification of AOP frameworks in Python

An interesting side of AOP implementations in Python is the dynamic nature of the language. Thus most implementations work by an technique called *monkey-patching*, cf. http://en.wikipedia.org/wiki/Monkey-patching, which is actually just the replacement of a function reference by another. The replacement function is called an *advice*, and it may add newly introduced behaviour or take a potentially deviating action when the original function is called.

One advantage of using a dynamic language like Python for implementing AO capabilities is the fact that no second "aspect" language for the AOP semantics must be learned, since the behaviour of built-in functions and objects can be extended and modified dynamically (even more so in classless/prototype-based languages like Javascript). Note however, that for statically-typed languages like Java approaches exist that involve a new syntax for aspect description, e.g., AspectJ at http://www.eclipse.org/aspectj/, but also approaches that use only plain Java code, e.g., Java Nanning at http://nanning.codehaus.org.

It's interesting to observe that for a long time no compile-time or load-time weaving AO FWs (also known as out-of-band instrumentation or program transformation) had existed for Python the way they have for Java, thus until recently it hadn't been possible to weave aspects into Python bytecode files beforehand (Matusiak 2009a).

All implemented solutions work dynamically from within Python code, the reason for this may be that Python implementations and bytecode may, e.g., vary in word width and other deployment machine dependent aspects.

Also, how the aspect-*weaving* is performed programmatically needs to be compared: Some FWs provide a set of `Aspect` and `Pointcut` classes that build logical units for the developer, while other FWs provide static (or singleton) module-level functions for wrapping program entities.

2.4. Feature completeness

When asking for features in an aspect-oriented FW, there are many dimensions to consider; we chose strategy, expressiveness, dynamics, versatility and constraints related to the system environment. For our comparison, we will focus on those five topics to provide a solid base for the decision over the framework selection.

- *Strategy* determines the technical realization of implementing AOP in Python,

  Environment constraints are hard requirements to consider, e.g., the allowed operating systems, platforms, virtual machines to run code in and also how aspects are applied to code in a build process. This information is useful when making a decision upon instating a certain FW for aspect-oriented programming,

  Expressiveness is mainly considered for the pointcut definition language and the advice predicates. To consider here are, e.g., the ability to supply regular expressions (RE) for matching code entities (method names, signatures, classes or variables accessed), as well as the choice of advices supported (before/after, around/proceed, call hierarchy and exception handling in advices),

  Dynamics comprises several concepts, too: Being able to enable/disable advices at runtime, to create and modify pointcut definitions at runtime, but also to access and modify the runtime context of a join point during execution,

  Versatility contains aspects of the aforementioned two: The range of programmatical situations in which AOP is supported by the FW. This includes the number of entities

that can be detected by a join point, e.g., interception of object creation and destruction, distinction between a method call and a method execution, or the ability to apply aspects on classes or objects and similar issues.

2.5. Framework comparison

As a pragmatic way to compare the three selected AOP FWs for Python, the following approach has been taken: All capabilities of the FW under test were hand-coded into test cases and automatically checked by using the `doctest` module (http://docs.python.org/library/doctest.html)[1], a unit testing FW similar to `unittest`, also available in the standard library since Python version 2.1, cf. http://docs.python.org/library/unittest.html. The test cases are designed in such a way that if the testing procedure does *not* produce any output warnings, all tests have passed and the tested properties with regards to AO features have worked as expected. Note however, that for Aspyct we had to adapt the code slightly for use with Python version 3.0, because with earlier versions of Python the `doctest` module didn't work well with Aspyct. Nevertheless we have used Aspyct with Python 2.5, which, for example, already provides decorators (annotations).

2.5.1. Strategy

According to Matusiak (2009a) the implementation of AO functionality in Python can be classified into *in-source modifications*, *external invocations* and *program transformations*. The former two can be applied statically (modifying meta-classes) and dynamically (changing method references):

Metaclass as hook relies on the static modification of a metaclass to ensure object modification before its initialization; this is used in Pythius. This requires at least one additional line of code for any advised Python class.

Dynamic mutation – informally known as *monkey-patching* – intermingles object code with aspect code to explicitly "weave" and "unweave" aspects into objects. This approach is most widely used, e.g., in Aspyct, Python Aspects, Logilab Aspects and PEAK. Note, however, that the widely known *Proxy*-approach to AOP as in Spring Python could be rather perceived as a "middleware than an aspect oriented transformation" (Matusiak 2009a, p.11).

Program transformation on the other hand doesn't require any change to the instrumented Python code at all (no import, no metaclass assignment). The only known FW to support this approach which is much closer to the original idea of AOP as defined in Kiczales et al. (1997) and implemented in *AspectJ*, is `aopy` Matusiak (2009b).

[1] Proposal for *docstring-driven testing* in 1999: http://groups.google.com/group/comp.lang.python/browse_thread/thread/0dfcc7e4daedd391/ 1c57cfb7b3772763

2.5.2. Environment

The Python port *Jython* (Pedroni and Rappin, 2002; http://www.jython.org) for the Java Virtual Machine (JVM) has been proven to be very useful when combining the rapid prototyping abilities (Bill, 2001) of the dynamic so-called *duck-typed* language Python with the reliable stability and enterprise integration of the JVM (Bagwell, 2009), i.e., http://code.google.com/p/robotframework/. Therefore the wish to use AO Python FWs directly from within Jython code is a desirable feature to consider.

In our tests, *Logilab Aspects* worked very well with Jython version 2.5.0, but *Python Aspects* wouldn't compile because of illicit parameters within the aspects library itself, plus our test suite failed because Jython seems to have non-standard interfaces in its built-in functions. The *Aspyct* test suite was designed for Python 3 and therefore couldn't be tested at all with Jython, because it hasn't been upgraded to version 3 yet, as of the time writing. These tests need to be repeated as soon as more recent versions of Jython become available.

Table 1 shows some basic information about the FW authors, licensing and environment compatibility.

Framework	**Python Aspects**	**Aspyct**	**Logilab Aspects**
Author(s)	Antti Kervinen	Antoine d'Otreppe	Logilab S.A. (France)
Website			
Version	1.3	3.0 beta 4	0.1.4
Lizenz	LGPL 2.1	LGPL	GPL?
Since	2003	2008	2006
Last update	2008-10-11	2009-04-01	2008-09-19
Python version (pass)	2.4/2.5 (2.1)	2.4/3.0 (2.x+3.x)	2.1/2.4
Jython version (pass)	2.5/-	-	2.5/2.5

Table 1: General framework overview. Values in parenthesis are official values. Versions given show the minimal version able to install the FW and to pass the test cases.

In the following subsections we will compare the FWs in detail according to the very specific properties of AO FWs enumerated above.

2.5.3. Expressiveness

The expressive power of join point definitions in the given implementations doesn't vary much: All FWs support method interception and wrapping.

With regards to encompassing several join points at different source code locations (called join point shadows), i.e., to capture not only calls to different objects, but also executions of different methods with only one expression to build up a pointcut, there is only Aspyct that supports pattern matching with the Pointcut class, at the same time being the only FW having a pointcut class at all. Logilab supports a "catch-all" mechanism for all class methods when advising a class or object, which can be useful in some cases, but is really coarse-grained compared to the powerful semantics of, e.g., AspectJ. When wrapping a complete class or object instance, for Python Aspects and Aspyct this works as an initialization wrapper (calls to `__init__` become advised).

When searching for differences between old-style and new-style classes, we couldn't find any harmful differences for applying aspects to both class types.

Exception handling within aspects is supported in all FWs by some degree.

The built-in advices supported by the FWs are always either `before` and `after` and/or `around` (wrapping), which can be used to emulate the former two. In Aspyct aspects are implented as classes, thus it's possible to emulate an around advice by using before and after advices (called `atCall` and `atReturn`/`atRaise`), cf. Figure 1.

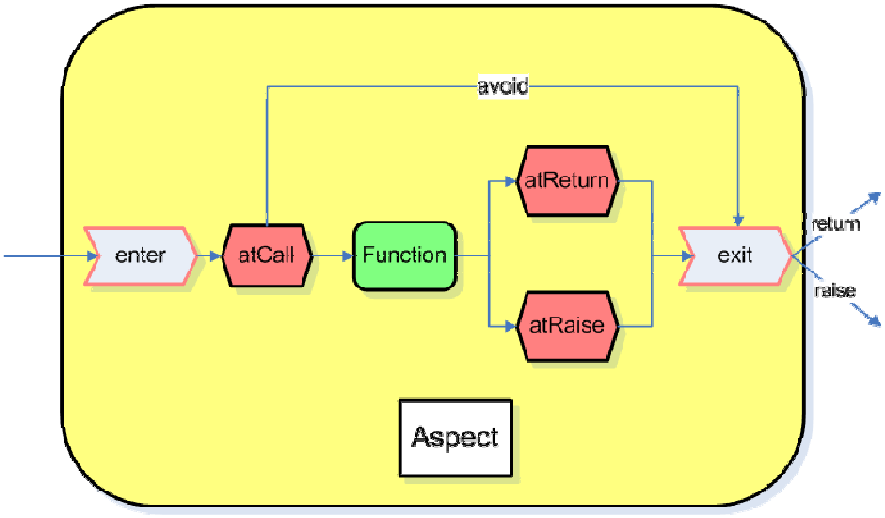

Figure 1: Call interception model of the Aspyct framework.

A `proceed` can be decided upon at runtime, in Aspyct by calling `avoid` on the call-data object to *not* proceed and in Python Aspects by yielding the static library functions `proceed`, `return_stop` or `return_cont`.

For consideration of the call hierarchy there is no expressive syntax in any of the FWs available, only Python Aspects provides some rudimentary support for constraining an advice to calls from specific objects, which could be considered a simple version of, e.g., AspectJ's `cflow` predicate. Nevertheless it is still possible to use introspection (reflection) mechanisms to access calling objects on the program stack for further investigation within the advice, as will be layed out in Section 3.

2.5.4. Dynamics

Dynamic disabling and enabling of woven aspects during runtime is only supported by Python Aspects, but can be straightforwardly implemented in the aspect classes supported by Aspyct and Logilab Aspects.

Removing the weaving itself is in most cases very easily possible, because method call interception is usually performed by simply bending the method or function reference to a dynamic proxy object that contains the advice logic and a reference to the original object; this arrangement is called monkey-patch. In Python Aspects and Logilab Aspects all wrapping calls are symmetric with unwrapping calls, in Aspyct one can simply save the reference to the unwrapped object and restore the reference when needed, a more global solution for removing aspects is scheduled for the next release (3.0 beta 5), although there hasn't been much activity on the website recently.

To create pointcuts on the fly during runtime seems to be uncommon among AO approaches, not only in Python. Due to the dynamic nature of Python itself, it can be done via meta programming techniques, which would be a research area yet fully to be explored ("Meta-Aspects" or runtime-predicated aspects).

For the considerations in Section 3, the ability to access information of the runtime context are of vital importance. Only Aspyct and Logilab Aspects seem to provide access to some limited information about the wrapped object and return/exception values of it from within the join point/advice.

2.5.5. Versatility

In Python Aspects and Aspyct only classes can be wrapped, unlike instantiated objects. Logilab Aspects, on the other hand, supports wrapping on the module level, applying interception of access to all objects within the module. Advising of classless Python functions is supported by all FWs, while wrapping a built-in function not written in pure Python but in pre-compiled C code doesn't work in all FWs; Logilab has the worst support for built-in function wrapping, but this may also be related to problems of interdependencies with the doctest suite. For recent Python versions, especially of the version 3 family, many modules have been rewritten to transparently use either a pure Python implementation or an accelerated C-version of the same module, thus offering better aspect interoperability, i.e., `datetime`, `functools`, `json`.

Wrapping of class methods in Python objects works only in Aspyct.

Unlike full-fledged AO semantics as found in AspectJ, there is no way to express the difference of an entity being *called* and being *executed* in any of the Python FWs.

Currently only Logilab Aspects allows removing all advices at once with the `reset` command.

2.5.6. Miscellaneous features

Aspyct and Logilab Aspects allow encapsulation of aspects in classes, while Python Aspects provides only static wrapping function. Python Aspects works with continuations in advices via `yield`. Aspyct is the only FW allowing for decorators to apply aspects, which consitutes, of course, a static dependency on the AO FW used.

2.6. Framework selection

For the use case described in the following section, we chose the AOP FW Aspyct. Its advantages are as follows: Aspyct is a very lightweight FW which can be run both on Python version 2.5 or higher and Python 3.0 or higher. It employs a very clear call interception model, the author provides helpful documentation and answers technical questions regarding the FW very fast, so Aspyct seems to be in healthy development as of the time writing.

The implementation of AO entities like Aspects, Pointcuts and context information is completely object-centered and very easy to use by inheritance of aspects. Join points can be build up by using regular expressions, a feature missing in the other FWs compared. Accessing the call context from within the advice is provided by the so-called `CallData` objects that provide references to the function or method advised plus the `self` instance of the object invoked. Also accessible are provided call arguments, both positional and keyword-arguments. Return or exceptional values from the advised object can be inspected too.

Aspyct allows for annotation of methods with aspects by using decorators, a feature not found in the other AOP FWs compared in this paper.

After all, Aspyct is a solid, well-designed and light-weight implementation providing AO functionality for Python developers.

3. Provenance Recording using Aspyct

As an example for utilization of the aspect-oriented principle we describe our implementation of provenance-awareness (PA) in a scientific data management application called *DataFinder* (Schlauch and Schreiber, 2007) in the *AeroGrid* project (http://www.aero-grid.de). We demonstrate how the lightweight library *Aspyct* eased the development of method observers for data-related object interactions and where additional effort was required. Using Aspyct we have developed the application binding layer of a FW for recording process documentation which is currently used to realize command line script wrappers and metadata change observers in DataFinder.

Before we describe our reasons for aspect-orienting our solution we introduce the field of application, what purpose provenance documentation serves and what technology our library implementation is built on. We then present the interface that the Python Provenance Client-side Library (PyProvenance) exposes to application developers, and how our data management application was made PA without changing its already existing code and without creating strict dependencies on the provenance library.

3.1. Provenance for scientific data management

A scientific application may produce a large volume of data as a result of complex workflows which are carried out by various specialists using scientific software and hardware of various kinds. In the domains of aerospace engineering and climate research there are process steps involving parallel simulation codes running on high performance computers, graphical applications for setup of parameters or analysis of results, tools for data

files transport through networks, metadata extraction utilities, data management services and so forth. Scientists and engineers of different experience levels, domains, companies and institute affiliations make use of these, creating, changing and using data products. Sometimes between process steps periods of time pass during which experts leave the participating institutions, taking their knowledge with them. Thus, keeping track of the creation history of data, the so-called Data Provenance, is an important requirement to guarantee quality of data and evaluate the correctness and reliability of production workflows.

One implementation of a service and client-side library to incorporate this kind of functionality into software has been developed in the project *EU Grid Provenance* (http://www.gridprovenance.org). This specification for provenance documentation provides a description format for the messages and their causal relationships that data processing software components exchange among each other. These components are referred to as *Actors* which perform *Interactions* which again depend on and produce *Information Items*}. In a software system these messages may be SOAP calls, method calls, subprocess executions, file changes, but it is not far-fetched to consider emails, telephone calls and sticky notes, as long as they have measurable influence on data products. One simple example of a causal relationship between two messages is the response of a service to a specific request. Since in turn requests are not sent without a reason, modeling these relationships allows for describing complete workflows as interaction chains between the involved actors. From process documentation of this kind not only the original processes can be inferred that led to a piece of data, but even why they have been carried out.

One necessity for such abilities for inference is the complete documentation coverage of a process chain. Since every participating component in the chain must be made PA without changing its business logic, tools for an orthogonal implementation of automatic provenance recording functionality are a requirement. A software system must become enabled to document important interactions and collect the used information items. The provenance architecture specification recommends the implementation of *provenance wrappers* around components whose behavior shall be documented. These wrappers expose the same messaging interface to their interaction partners as the contained service, forward to it every message they receive and send every response or request of the contained service to its targets. Descriptions of all those messages are recorded to dedicated *provenance stores*, including the corresponding interaction contexts, such as service state or request-response-relationships. Since the application cases can be very different, the provenance client-side library from the EU project does not provide an implementation of such a wrapper nor specific tools to accomplish the wrapping. What information has to be recorded and how it is collected must be modeled specifically for each respective workflow. The purpose of the Java-based library *PyProvenance* is focused on providing means to create documentation records formatted to conform to the provenance service protocol and to send it to a provenance store.

What remains to be accomplished by the software architects is modeling the interactions, recognizing when they take place and describe the information items to which to refer to in the documentation.

3.2. The DataFinder application

DataFinder is a light-weight client application for scientific/technical data management. It allows for distributed storage of documents on several backends, e.g., WebDAV, FTP, GridFTP, Subversion or Amazon's S3, while keeping all metadata in a structured central repository, allowing for structured search queries. There are three GUI applications: a web frontent used by the larger part of our users, a desktop client application and a desktop administrative application, that allows to model and customize the underlying data model. DataFinder is open source software developed at DLR since 2005 and is used mainly by engineers in DLR institutes with a user base of several hundred scientists in diverse projects and departments. There are usually one or two developers working full-time on DataFinder, while often students help in implementation of certain features when writing their theses. DataFinder has a code base of nearly 100.000 lines of code in the current version 2.2, as reported by http://www.ohloh.net/p/datafinder/analyses/latest.

3.3. Indicators for an aspect-oriented solution

The PrIMe Methodology (Munroe et al., 2006) describes an analysis method for systems to be made PA. Its aim is to identify the relevant actors, interactions and information items in a system to determine which components have to write process documentation and what they are required to contain. Another result of the PrIMe analysis are the types of relationships between the identified entities. From an AO point of view the join points in such a system correspond to the interaction routes.

A natural solution is applying some kind of observer to each actor and let the observers document the actor's behaviour in terms of incoming requests and returned responses. The most important motivation for this is, that this way the code of the actors does not have to be altered and only an easily adjustable one-way dependency from the documentation features to the business logic is established (and not the other way around), cf. *inversion of control* (Fowler, 2004). Another one is that these independent observers can be arranged to exchange information about states and behaviours of their respective observed actors to establish a more complete picture of the application context than would be possible from just a single call. This latter point is of high importance to overcome availability restrictions of certain information items within object scopes in an application, such as, for instance, user session information, which is outside the accessible scope of a data property management component.

3.4. Issues with automatic interaction documentation

While at first glance recording provenance information may seem to be a usual logging task, the exercise is more complex. Ordinary logging methods, as offered for example through the Python module `logging` are just suitable to write a human readable representation of a few properties that represent an execution state to a log file. A provenance system is used to record a more context involving, machine readable representation of *what* is actually *happening*, *how*, and *why*. A provenance model can be designed to reflect the inner mechanics of wrapped actors through the relationships of the messages it receives and sends.

Yet provenance recording is similar to standard logging methods in that it is not allowed to interfere with the ordinary flow of the process it documents and thus realizes a feature set which is orthogonal to that of the business logic.

The problems described in the following sections all refer to the difficulty of handling interaction contexts, for example when the documenting entities have a static relationship with the actors whose behaviours they document. A fully aspect-oriented documentation system can overcome these, since aspects can be designed to remember context, for example by retaining context information in map-like structures (a *dictionary* in Python lingo) until the corresponding process is confirmed to have ended.

3.4.1. Ambiguities in interactions of executables and services

When we see executables and services as closed applications that we cannot or should not change internally to enable documentation of their interactions, a feasible solution is to use some kind of proxies. For a stand-alone executable this can be a wrapper script, which calls the original executable and records command line parameters, exit codes, input/output streams, file system changes and so on. For a web service it would be a proxy service, that receives requests to the wrapped service, delegates them after recording and proceeds with recording the response of the original service afterwards.

Nonetheless there is a generalization problem with service proxies. In all but the most simple communication cases, namely serial one-to-one interactions, the proxy must take the inner mechanics of the wrapped service into account to determine, which interactions of a service are related to each other. A service `B` might call another service `C` on behalf of `B`'s caller `A`. The two interactions `A` → `B` and `B` → `C` are causally related, but are not a request-response-pair. In another case `B` might contact `C` frequently, independently from any request from `A` and so `B`'s proxy should *not* document any such relationship, even though `B` → `C` might happen directly after a request `A` → `B`.

Overcoming these ambiguities in wrappers and proxies can be a tedious and error-prone task, since there is usually no shared superior execution context, that wrappers could access for documentation reference, unless they create one explicitly. This is accomplished by PA components explicitly exchanging provenance-relevant information. One form are invocation tracers that are simply identifiers for a chain of invocations. Detailed information becomes accessible per *GlobalPAssertionKey*s (Groth et al., 2006, p. 68-69), which are references to other recorded provenance documentation, in this case of a superordinate process.

3.4.2. Interactions between objects

Making an integrated application PA benefits from the clear execution context of every method within it, that can be looked up from the current call stack by using introspection. But the proxy or wrapper approaches are not as useful here as the AO approach we are going to show in the following.

In any case the application has to be decomposed into the components that communicate the important information items. These so called *Knowledgable Actors* can be objects, and their

interactions are carried out by invocations of methods of one object from methods of the other.

In our use case the data management system *DataFinder*, mentioned in Section 1, the information items of interest are, e.g., meta data properties, the user session, storage resources and external executables. Two components that work with these items are the GUI layer and the façade object, which provide the most important features of the data management components to GUI and extension scripts as Python methods. Some of these methods cover a similar set of features as usual file system operations, such as *create*, *copy*, *move*, *delete*. Another important one is a `login` method that establishes a network connection to a meta data server using the user's credentials.

Implementation of the documentation features as proxies or wrappers would manifest itself in the extension of each class identified as an actor by overriding the interactive methods with provenance documenting wrapper calls. The AO approach allows for a much more generic implementation instead. A *pointcut* is defined to cover the interactive methods and a provenance recording advice is "woven" into the program code, which again uses the original method. The created aspect consisting of pointcut and advice creates documentation according to the signature of the called method and the parameter values. When the method returns, its result is recorded accordingly. Before the advice call returns, it can as well document the relationship of the recorded messages that represents the invocation of the method and its return value. Such a generic documentation aspect for method calls can easily be extended to record and relate additional pieces of information according to the recognized method which is what has been done to cover, e.g., the AeroGrid project provenance model in DataFinder.

One problem that the AO model cannot inherently solve stems from application contexts that do not represent running processes, but just program states, such as an active user session. Since in a provenance model responsibility for process execution or data changes usually is a property of high importance, any performed interaction of the GUI must be related to the running user session and thus requires a specific adaption of the documenter aspect by keeping a key referring to the session documentation.

Another requirement of process documentation arises from the communication facet of the provenance model. Two interacting actors each have their own view onto sent and received messages, or at least have access to different context details about any one interaction they perform. This means that both partners would probably record different information about the same interaction, for example with relationships to different context documentation. Our presented provenance implementation provides means to enable partner services to refer to the exact same interaction when they record, by explicitly exchanging reference keys with their messages, but this is a feature that cannot be used in an aspect-oriented provenance extension for an – any other way unmodified – stand-alone application. Since the pointcut places the execution of the documentation aspect between the sending and the receiving of a method invocation within the same application, the aspect must be able to generate the documentation for both sender and receiver of a message, if necessary. While it is usual for an aspect to have access to the context in that the called method is going to be executed, the reviewed Python aspect FWs of Section 2 do not provide sufficient in-depth access to the

execution context of the callee method. This is a serious issue which in this example could only be solved by a kind of "hack" using the reflection mechanism offered by the `inspect` module of the Python standard library as described in the following section.

3.5. Aspect-orientation in the python provenance client-side library

The previous sections indicated some issues with process documentation. Aspect-orientation can help overcome some of them, others need working around.

For the implementation of the generic method observer as an aspect we chose the Aspyct framework because

- it is very lightweight,
- it provides an aspect class which enables creation of the users' own aspects by simply overriding the `atCall`, `atReturn` and `atRaise` methods,
- it enables definition of pointcuts using regular expressions over the names of object properties,
- it is under active development and the module's developer reacts quickly to support requests.

3.5.1. Implementation of the PyProvenance library

The Python Provenance Client-side Library *PyProvenance* (http://sourceforge.net/projects/provenance-csl/) consists of a *JPype*-based Python wrapper (http://jpype.sourceforge.net) for the Java client-side library *Java CSL* from the project *Provenance-aware Service-oriented Architecture (PASOA)* (http://www.pasoa.org) which was extended to a *Globus Toolkit 4* grid service in the EU-funded project *Grid Provenance* (Foster, 2006; Jackson, 2002; Bochner et al., 2009, p. 229-240). The Java CSL provides means to build provenance records and send them to the recording endpoint of a provenance store service (Gude and Oster, 2007). Additionally it allows performing provenance queries (Groth et al., 2006, p. 89-99).

PyProvenance increases the usability of provenance recording for Python users by providing more "Pythonic" methods to build and express provenance records. There is, e.g., the abstract utility class `Documenter` and several implementations which provide serialized XML representations for safe pickling of standard Python constructs like lists and dictionaries and also a fall-back implementation for other cases.

3.5.2. Aspect model of the library

Provenance recorder. To enable provenance store connection configuration in a PA client application independently from observer implementations and to be able to handle several connections to provenance stores, the provenance recorder class encapsulates a provenance store URL and a reference to a Python wrapper of the Java CSL. A provenance recorder is immutable after initialization and is used by an observer to create recording handles that take

the documentation entries for a single event. A recording handle can then be sent once to the provenance store by the provenance recorder.

Observing method calls. The functionality to observe generic method invocations is realized as an aspect named `MethodCallObserver`, shown in Listing 1. Its constructor stores the provenance recorder instance reference in a local Python property (attribute). This of course limits the use to instance variables of new-style classes; old-style classes and class variables cannot be wrapped in properties (Matusiak 2009a, p. 3).

```
class MethodCallObserver(Aspect):
    def __init__(self, provRec)
    def weaveToMethod(self, classRef, methodRE)
    def weaveToObject(self, classRef)
    def getCallingObject(self)
    def getCalledObject(self, cd)
    def isRecordingNecessary(self, cd)
    def documentMethodDependencies(self, cd, \
            recordhandle, gpakOfMethodDoc, \
            methodArgs)
    def atCall(self, cd)
    def atReturn(self, cd)
    def atRaise(self, cd)
```

Listing 1: Method signatures of the Aspect MethodCallObserver

The `atReturn` method is the only overridden `Aspect` hook that is implemented with logic at this point, because the method documentation should by default only be recorded after its successful execution. It takes the current call data object (CDO) and uses the implementation of `isRecordingNecessary` to determine whether it is going to proceed to creating actual process documentation. This Boolean function returns 'False' by default but can be overridden by specific observers to respect the application state to record the call. The methods `weaveToMethod` and `weaveToObject` are helpers for weaving an observer either to all public methods of an object or to those methods defined by the regular expression in parameter `methodRE`. This way an application using PyProvenance is not required to import Aspyct just to specify pointcuts. When the application process requires more complex documentation than can be derived from the signature and parameters of the invoked method, `MethodCallObserver` can be inherited overriding `documentMethod-Dependencies`. The `recordhandle` stores the additional documentation items. The parameter `methodArgs` contains a dictionary with all parameters of the method invocation, while `gpakOfMethodDoc` provides a global reference key to the already automatically created documentation for the current invocation, so that it can be reused in relationships descriptions.

Using the other helper methods, the current `self`-contexts of the callee and the caller object can be retrieved and their properties can be accessed. Since the Aspyct framework at the time of writing did not provide the calling object context with the CDO, the `getCallingObject` function retrieves it by peeling five frames from the interpreter stack using the Python module `inspect`, cf. http://docs.python.org/library/inspect.html. Even though this could be considered a "hack", this solution ran stable for all tests. It might stop working in the future due to potential implementation changes of the Aspyct framework or

the Python interpreter. Apart from that it is very important to explicitly clear any reference to the self context of the calling object after use in a custom observer implementation to prevent memory leaks by circular references.

3.5.3. Weaving aspects into an application

When basic documentation about interaction recording is sufficient, the core "business" application can begin provenance recording by accomplishing the following steps:

- Configuration of a provenance recorder as described above
- Specification of all methods to be observed using class names and/or regular expressions
- Initialization of a method call observer instance with the provenance recorder
- Weaving the observer to the specified methods with the `weave*` methods.

As soon as more detailed documentation is required, particularly with causal relationships, specific observer implementations have to be created by inheriting `MethodCallObserver` in a module of the business application. Since specialized implementations will most probably only fit to certain methods, it is good practice to define and apply the regular expressions in the overridden constructor of the inheritor. Listing 2 shows this for the user session initialization routine in DataFinder. The observer is woven onto the selected method of the class `ExternalFacade` as soon as it is initialized, which is why it is useful to create a well-arranged central module, in which all observers are woven into.

```
def __init__(self, prov_rec):
    MethodCallObserver.__init__(self, prov_rec)
    self.weaveToMethod(ExternalFacade, \
        "^performBasicDataFinderSetup")
```

Listing 2: Constructor of the user session observer for DataFinder

The session observer records a special relation about the user, who logged in using the specific instance of a GUI called *DataFinder*. The `isRecordingNecessary` function is overridden to return 'True', when provenance recording is activated in DataFinder.

Dealing with context. Apart from recording documentation, the user session observer sets a singleton with the value of the provenance key referring to the login interaction documentation, which can be used by other method observers to relate other recorded activities to the responsible person. This is a simple example for a way of keeping context information.

A way of keeping track of nested method execution contexts is implementing a context dictionary in an observer inheritor. By overriding the `atCall` method, any specific information can be put into the dictionary with the CDO as key. This information can be made accessible by other observers who might require it for their documentation. During the

`documentMethodDependencies` the data associated with the current CDO has to be removed from the dictionary again.

3.6. Analysis of experiences with the implementation of AOP-based provenance-awareness for DataFinder

For PyProvenance we have implemented a generic method call observer as an Aspyct aspect, which has no dependencies on the business code and creates basic class interaction documentation by itself based on the method signature, parameter settings and result of the returned method. Through this the effort of covering all relevant DataFinder methods with specific provenance documenters has been almost completely reduced to modeling the provenance expressions. This not only reduced development time, it improved the legibility of the DataFinder-specific code considerably and reduced its sensitivity to errors. Still the most important benefit of the aspect-oriented approach is the complete independence of DataFinder from the provenance specific logging code. No part of the DataFinder core code required any adaptation to enable provenance recording. At the moment the Python Provenance Client-side Library wraps the original Java library, which requires an available Java Virtual Machine (JVM) for the provenance recording feature to be operational. At startup DataFinder looks for the availability of a sufficiently recent JVM and – in case none is available – simply skips weaving of the method observers. At this point the only required changes to include the provenance feature was an import statement for the provenance configuration module in the DataFinder client start script and the insertion of a provenance recording check box in its preferences dialog window.

4. Conclusion and Outlook

We showed selection criteria and feature requirements for AOP frameworks and evaluated three of them for the incorporation of provenance recording features in a data management tool for distributed applications. To compare existing FWs we defined a high-level classification to evaluate the different software solutions against and showed the current state of development. After laying out the criteria for framework selection we explained our goals in more detail and showed that we found the lightweight framework *Aspyct* most suitable for our use case. Overall, aspect-oriented techniques turned out to be the best choice to accomplish non-invasive extension of an integrated application with process documentation features.

In the future a more formal comparison of existing Python AOP implementations needs to be undertaken to elaborate on the internal details regarding the dynamicity of the Python language and the solution strategies chosen for AOP in this language.

Further enterprise-grade applications need to be evaluated to determine if Aspyct can be applied to fulfill their needs regarding AOP, too.

References

Bagwell, D. (2009): *WebSphere z/OS V6.1 - WSADMIN Primer (with Jython Scripting Illustrated)*. White Paper WP101014, International Business Machines Corporation,

Washington Systems Center, Jan, 19, 2009. http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101014

Bill, R. W. (2001): *Jython for Java Programmers*. Sams Publishing

Bochner, Carsten and Gude, Roland and Schreiber, Andreas (2009): *A Python Library for Provenance Recording and Querying*, volume 5272 of Lecture Notes in Computer Science, pages 229-240, Springer

Fayolle, A. and Mascio, A. D. and Thénault, S. (2007): *Aspect-oriented, design-by-contract programming and rpython static checking*. Technical Report D10.1

Foster, I. (2006): *Globus toolkit version 4: Software for service-oriented systems*. Lecture Notes in Computer Science, 3779:2-13

Fowler, M. (2004): *Inversion of control containers and the dependency injection pattern*. http://www.itu.dk/courses/VOP/E2006/8_injection.pdf

Groth, P. and Jiang, S. and Miles, S. and Munroe, S. and Tan, V. and Tsasakou, S. and Moreau, L. (2006): *An architecture for provenance systems*

Gude, Roland and Oster, M. (2007): *Provenance-CSL - A Provenance Client-Side Library*. Fachhochschule Bonn-Rhein-Sieg

Jackson, K. R. (2002): *pyGlobus: a python interface to the globus toolkit*. Concurrency and Computation: Practice and Experience, 14(13-15): p. 1075-1083

Kiczales, G. and Lamping, J. and Mendhekar, A. and Maeda, C. and Lopes, C. and Loingtier, J. and Irwin, J. (1997): *Aspect-oriented programming*. In: ECOOP. Springer

Matusiak, M. (2009a): *Strategies for aspect oriented programming in python*. http://www.matusiak.eu/numerodix/blog/wp-content/uploads/aop_strategies.pdf

Matusiak, M. (2009b): *aopy: A program transformation-based aspect oriented framework for Python*. http://www.matusiak.eu/numerodix/blog/wp-content/uploads/aopy.pdf

Munroe, S. and Miles, S. and Moreau, L. and Vázquez-Salceda, J. (2006): *Prime: a software engineering methodology for developing provenance-aware applications*. In: Proceedings of the 6th international workshop on Software engineering and middleware, Foundations of Software Engineering, pages 39-46. ACM

Pedroni S. and Rappin, N. (2002): *Jython Essentials*. O'Reilly Media, Inc.

Rigo, A. and Pedroni, S. (2007): *JIT compiler architecture*. Technical Report D08.2

Schlauch, Tobias and Schreiber, Andreas (2007): *DataFinder - a scientific data management solution*. In: Ensuring the Long-Term Preservation and Value Adding to Scientific and Technical Data

Washizaki, H. and Kubo, A. and Mizumachi, T. and Eguchi, K. and Fukazawa, Y. and Yoshioka, N. and Kanuka, H. and Kodaka, T. and Sugimoto, N. and Nagai, Y. and Yamamoto, R. (2009): AOJS: *Aspect-oriented javascript programming framework for web development*. In: ACP4IS'09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software, p. 31-36, ACM