WPS orchestration using the Taverna workbench: The eScience approach

J. de Jesus *, P. Walker, M. Grant, S. Groom

Plymouth Marine Laboratory, Prospect Place, Plymouth, Devon PL1 3DH, UK

Abstract
eScience is an umbrella concept which covers internet technologies, such as web service orchestration that involves manipulation and processing of high volumes of data, using simple and efficient methodologies. This concept is normally associated with bioinformatics, but nothing prevents the use of an identical approach for geoinformatics and OGC (Open Geospatial Consortium) web services like WPS (Web Processing Service).

In this paper we present an extended WPS implementation based on the PyWPS framework using an automatically generated WSDL (Web Service Description Language) XML document that replicates the WPS input/output document structure used during an Execute request to a server. Services are accessed using a modified SOAP (Simple Object Access Protocol) interface provided by PyWPS, that uses service and input/outputs identifiers as element names. The WSDL XML document is dynamically generated by applying XSLT (Extensible Stylesheet Language Transformation) to the getCapabilities XML document that is generated by PyWPS.

The availability of the SOAP interface and WSDL description allows WPS instances to be accessible to workflow development software like Taverna, enabling users to build complex workflows using web services represented by interconnecting graphics. Taverna will transform the visual representation of the workflow into a SCUFL (Simple Conceptual Unified Flow Language) based XML document that can be run internally or sent to a Taverna orchestration server. SCUFL uses a dataflow-centric orchestration model as opposed to the more commonly used orchestration language BPEL (Business Process Execution Language) which is process-centric.

1. Introduction

Over the last decade, significant increase in data availability associated with the use of the World Wide Web by researchers has popularised the terms eScience and eScientist (Cavalcanti, 2005); these terms normally define methodologies that are computer intensive (Giuliani et al., 2011; Huang and Yang, 2010), require large dataset manipulation/discovery (Chen et al., 2010a; Nativi et al., 2009) and use/reuse of atomic web services (Granell et al., 2010; Krishnan and Bhatia, 2009) that can be orchestrated using workflow software (Ludäscher et al., 2006; Oinn et al., 2004) or languages like BPEL (Business Process Execution Language) (Brauner et al., 2009; Bosin et al., 2011).

eScience concepts have been commonly used in geosciences, with development of web services and standards by the OGC (Open Geospatial Consortium),1 that specify how data should be structured and transferred in SOA (Service Oriented Architecture) environments. The number and level of complexity of OGC services has increased over time but four services have become well established in the area of data delivery and processing: Web Map Service (WMS), Web Feature Service (WFS), Web Coverage Service WCS and Web Processing Service (WPS). The first three are used for image, vector data and raster coverage data transfer, respectively; the last one (WPS) allows for geoprocesses (algorithms) to be offered as service instances (Meng et al., 2009a), which can be applied to the data services. WPS is of particular interest for geoscientists, since simple algorithms working together can be used to build complex models (Granell et al., 2010; Chen et al., 2010b) using an orchestration structure.

Building complex processing chains using WPS has been studied and/or applied by several authors (Foerster et al., 2009; Brauner et al., 2009; Weiser and Zipf, 2007) and WPS orchestration proposals follow two strategies

- Simple WPS processes that accept a workflow description and coordinate the calling/retrieval of data from different web services, using an internal BPEL engine (Schäffer, 2008; Brauner and Schaeffer, 2008).

1 Corresponding author. Tel.: +44 1752 633 100; fax: +44 1752 633 101.
E-mail addresses: jmdj@pml.ac.uk (J. de Jesus), petwa@pml.ac.uk (P. Walker), mggr@pml.ac.uk (M. Grant), sbg@pml.ac.uk (S. Groom).
• Generic uses of BPEL, that describe the orchestration of web services, which will be run in its own BPEL engine (Weiser and Zipf, 2007; Chen et al., 2010b).

Both approaches have strengths and weaknesses, but neither is suitable for a non-technical user without the addition of a GUI client to guide them through workflow creation and deployment. The 52 North WPS Workflow Modeller2 is a good example of a webGUI helping users to build workflows that use OGC services. The second strategy is more generalist and is the default approach to web service orchestration in fields like computer sciences and business.

This paper, presents an extension to the SOAP and WSDL structures described in the WPS 1.0.0 specification, allowing for a better integration into generic web service structures and workflow managers like Taverna. The final objective is an intuitive graphical orchestration platform that does not require high levels of computer literacy, and allows for geoscientific users to focus primarily on model building and data manipulation.

2. Technical background

This section reviews the relevant technologies underpinning this work; WPS (specifically PyWPS), WSDL and the workflow orchestration approach used in the Taverna system (including SCUFL).

2.1. WPS and WSDL

The WPS 1.0.0 (Schut, 2007) standard specifies how atomic geospatial processes can be run in a SOA environment using HTTP-GET, HTTP-POST and SOAP interfaces. The geospatial processes being served in a WPS instance should provide metadata and I/O (e.g. base64 encoded image, XML, bounding box) description through two operations, getCapabilities and describeProcess, while service execution is controlled by an Execute request containing input data and, if necessary, how the output should be returned.

WSDL (Web service definition language) (Christensen et al., 2001) is an XML based language that can be used to describe a WPS instance, operations supported and type of I/O. WPS 1.0.0 supports the use of WSDL by offering a WSDL description, which can be requested as follows:

• One WSDL document describing all WPS processes (http://hostname/WPSname?WSDL).
• One WSDL document for each individual process (RESTful approach) (http://hostname/WPSname/identifier/service.soap?WSDL).

WPS 1.0.0 defines that getCapabilities should contain a link to a WSDL document that describes all the WPS processes, while describeProcess may (optionally) contain a link to a WSDL document that only describes the process.

WPS 1.0.0 Annex D (Schut, 2007), defines the SOAP interface and how WPS shall accept requests and responses encoded in SOAP, but it also states that the Execute operation can use a different structure in which the process name is converted into an element to be used inside the SOAP content (Listing 1).

Listing 1. WPS SOAP Execute request with process identifier GMLbuffer turned into element ExecuteProcess_GMLBuffer. Source: WPS 1.0.0 document (Schut, 2007).

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <ExecuteProcess_GMLBuffer
      xmlns="http://wpsint.tigris.org/soap/SpatialAnalysis">
      <GmlUrlResource>http://onotta499199/gml/polygon_gml.xml</GmlUrlResource>
      <Distance>10</Distance>
    </ExecuteProcess_GMLBuffer>
  </soap:Body>
</soap:Envelope>
```

Listing 2. Identifier element with process identification as text content.

```xml
<ws:Execute service="WPS" version="1.0.0" ......>
  <ows:Identifier>GMLBuffer</ows:Identifier>
  [...]
</ws:Execute>
```

The “ExecuteProcess_<process identifier>” is a simplified version of an Execute request for a specific process: WPS uses process identification based on an identifier element with a textual content (Listing 2); this content will become part of the identifier element name.

Inputs and outputs follow the same approach where the textual content of an identifier <ows:Identifier> is converted to an element whose name incorporates the content. Annex D does not contain any definition for outputs, how attributes (for example mimeType in complexData) should be processed/transformed nor how to make an asynchronous request.

2 http://52north.org/maven/project-sites/wps/52n-wps-orchestration-site/.
2.2. PyWPS

PyWPS (Cepicky, 2006) is a WPS implementation written in the Python language. The current stable version 3.1.0\(^3\) offers WPS 1.0.0 support. PyWPS provides a framework where programmers can deploy their geospatial algorithms. The approach of PyWPS is not to offer processes but the means to create them by facilitating access to the GRASS GIS and allowing for any Python code to be run and served as WPS. The framework structure is organized into packages and classes that work using a “factory strategy”, starting by parsing the inputs submitted to the WPS, offering them to the process source code, retrieving the outputs and finally generating the WPS XML (or raw) response. The factory structure approach and small size (5000 lines of code) of the framework make it a suitable candidate for testing and developing new WPS features like encryption, XML dynamic transformation, HTTP server support, etc.

2.3. Workflow management systems

Workflow orchestration is the activity of defining a sequence of tasks needed to manage a business, computational or engineering process: the workflow is the template to be followed during orchestration (Ludäscher et al., 2006; Deelman et al., 2009). A workflow could be expressed in any modern programming language (Deelman et al., 2009); however, building a workflow normally does not require programming skills and supporting software tends to have advanced visual design features that help users building complex workflows representing models.

Scientific workflows are supported in many languages, tools and systems (Tan et al., 2009; Bosin et al., 2011; Deelman et al., 2009); wikipedia lists 20 workflow management systems\(^4\) and each one has different aspects that can be categorized by

- Textual workflow editing (workflow language).
- Graphical workflow editing (graphical tools).
- Workflow components (API and core system access).
- Semantic composition (semantic enabled services/data).

BPEL (Business Process Execution Language) is the most common language used to orchestrate web services in workflows and it has implementations from IBM, Apache foundation, Oracle and Microsoft. BPEL has been used in business workflows to deal with repetitive tasks in a fixed pattern, but scientific workflows need to process higher volumes of data with continual changes of execution pattern (change in services and data used) and with the need of a proper failure procedure (Bosin et al., 2011). Despite the successful use of BPEL (Schäffer, 2008; Schäffer et al., 2009; Chen et al., 2010b; Meng et al., 2009b) some literature points to the use of other orchestration languages like SCUFL (Simple Conceptual Unified Flow Language) as a more “scientific and data oriented” orchestration language\(^5\) (Tan et al., 2009; Missier et al., 2010).

2.3.1. Taverna workbench

Taverna\(^6\) workbench is an open-source tool designed for composition and enactment of bioinformatics workflows (Hull et al., 2006), nevertheless it is generic enough for web services from other scientific fields (like geoinformatics) to be run. A Taverna workflow is a linked graph of processors that are processes or executable components that will accept input data, process them and create an output (Tan et al., 2009). Processes are described by WSDL while executable components are created using the beanshell\(^7\) script language, R language\(^8\) using Rserve\(^9\) and Xpath\(^10\) (Clark and DeRose, 1999) expressions. Each process node consumes data that arrives at its input ports and produces data on its output ports; data dependencies are created by linking between output ports (sources) and input ports (sinks) of different processes (Missier et al., 2010).

The Taverna project provides a remote execution environment referred to as Taverna server, which can be accessed using RESTful or SOAP interfaces. The Taverna server can be used as an orchestration engine since it accepts a workflow description in an XML format, accepts input content, executes the workflow and serves the final results to the client.

2.3.2. SCUFL model

SCUFL can be considered an XML document describing the workflow plus a model on how a workflow should be described and run (Oinn et al., 2006). This model assumes

- Services are represented by processors, containing inputs and outputs ports.
- Processors will be run as soon as possible based on data availability.
- Processors will run in parallel as default.
- Data are held in an XML wrapper although the data may be of any type.
- Data sources are connected to destinations by “Data links”.
- Running order dependencies are specified using “Control links”.
- Services are small and atomic.

\(^3\) http://pywps.wald.intevation.org/.
\(^6\) http://www.taverna.org.uk.
\(^7\) http://www.beanshell.org.
\(^8\) http://www.r-project.org.
\(^9\) http://www.rforge.net/Rserve/.
\(^10\) http://www.w3.org/TR/xpath/.
The free flow of data and lack of specific control structures (If, Pick, ForEach, RepeatUntil and While (Curcin and Ghanem, 2008)) are the major difference from BPEL which is more process oriented (Tan et al., 2009) and requires an explicit definition of the control flow precisely determining the order of execution of the process (Missier et al., 2010).

Taverna 1.7 uses a SCUFL XML document as default description format whereas, new versions of Taverna (2.x) define a new XML format (t2flow), also based in the SCFUL model and that will be used until the official release of the new SCUFL2.0 XML format.

3. Implementation

The following subsection describes the implementation and developments made to PyWPS to generate a simple WSDL document that in turn is used by Taverna to orchestrate WPS services. All developments were made in the pywps-3.2-soap branch,11 that will be part of the official PyWPS 3.2 release.

3.1. SOAP development

Following WPS 1.0.0 document Annex D (Schut, 2007), PyWPS 3.2 (new release)11 implemented the default SOAP structure, where a WPS XML request (getCapabilities, describeProcess, Execute) inside a SOAP envelope is processed as a normal WPS request. Both request and response are wrapped in a SOAP envelope.

3.1.1. SOAP execute request

The special SOAP execute request (referred to from this point as compressed SOAP request) uses the same input described in the WPS 1.0.0 Annex D (Schut, 2007).

Listing 3. Example of SOAP request using Literal and ComplexData. Note: SOAP envelope has been removed.

```
<ExecuteProcess_ComplexProcess>
<input1>10 </input1> <!--LiteralData-->
<input2> <!-- <a> <b/> </a> --> <input2> <!--ComplexData-->
</ExecuteProcess_ComplexProcess>
```

There is no difference if ComplexData is a URL or actual data (XML or base64). BBOX input is not currently implemented.

The output structure follows the same principle, but adds the suffix “Response” to the output.

Listing 4. Example of SOAP response from request in Listing 3. Note: SOAP envelope has been removed.

```
<ExecuteProcess_ComplexProcessResponse>
<!--ows:Identifier-->output1 <!--ows:Identifier-->!
<output1Response>11</output1Response>
<!--ows:Identifier-->output2 <!--ows:Identifier-->!
<output2Response> <!-- <c> <d/> </c> --> <!--ComplexData-->
</ExecuteProcess_ComplexProcess>
```

The inclusion of suffix “Response” to outputs is a common feature of WSDL Documents12 and it is also present in the WPS example document for WPS.13 When using a SOAP compressed request there is no way to request the WPS service to output contents by reference (using only XML elements). If a reference output is required, the user could program the PyWPS process to generate the output as a MapServer WxS reference,14 where a raster content will be served as WCS and vector/XML as WFS.

Conversion of text content into an element must follow the W3C XML standard (Bray et al., 2008) concerning allowed characters and name structure.15 It is common for WPS processes to be used as wrappers for existing shell commands, such as `gdalinfo`, which take their options in the form “-option_name option_value”, e.g. “gdalinfo -stats”. These option names, starting with “-” are often used unchanged as input attribute names in the WPS process. This is compliant with the WPS standard, however, when the methods are mapped into a SOAP/WSDL structure the attributes are mapped to XML elements, which must not start with a “-”. In order to allow the use of these legacy WPSs without requiring recoding, an automated process has been developed to transparently map these attributes to valid element names. This subject is further explored in Sections 3.2.3 and 3.2.4.

---

12 http://www.w3schools.com/wsdl/wsdl_documents.asp.
13 http://schemas.opengis.net/wps/1.0.0/examples/example_service.wsdl.
14 http://wiki.rsg.pml.ac.uk/pywps/Mapserver.
15 http://www.w3.org/TR/REC-xml/.
3.1.2. SOAP fault

The SOAP protocol has a fault structure that is used to carry errors and/or status information inside the SOAP message (Box et al., 2000). The fault structure can use four new elements (Listing 5), <faultcode>, <faultstring>, <faultactor> and <detail>. The <faultcode> and <faultstring> are mandatory elements in the response. It is also possible to include WPS exception reports (XML document) inside the detail element (Listing 5). A SOAP fault with WPS exceptions schema description will then be used in the WSDL description. See Section 3.2.1 for WSDL integration.

This SOAP fault used by PyWPS, follows the WCS 2.0 XML/SOAP binding specification (Baumann, 2010).

Listing 5. SOAP fault structure with WPS exception.

```xml
<SOAP-ENV:Envelope...>
  <SOAP-ENV:Body...>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>Server exception was encountered</faultstring>
      <detail>
        <wps:ExceptionReport...>
          <ows:Exception exceptionCode="MissingParameterValue"
            locator="input2"/>
        </wps:ExceptionReport>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

3.2. WSDL development

WSDL development followed the strategy of mimicking the structure of WPS as closely as possible

- Use of DataInputs and ProcessOutputs as name of message part.
- Operation contains I/O message and fault section indicating a WPS Exception Response message.
- Individual I/O element names.
- Compressed SOAP naming structure and generic WPS XML messages.

![Fig. 1. Taverna service representation derived from Listing 6.](image1)

![Fig. 2. Taverna’s service I/O representation, obtained from Listing 8.](image2)
Assigning DataInputs and ProcessOutput name to message parts responsible for I/O messages (Listing 6) allowed for a more comprehensive WSDL resulting in an easier to understand graphical service box in Taverna 2.2.0 (Fig. 1). Taverna represents I/O in two separate graphic boxes related to the process (Fig. 2), since it is considered that I/O XML content is generated by another processor and then passed to the DataInputs/DataOutputs section of the WPS process.

Listing 6. WPS’s DataInput and ProcessOutputs to define I/O WSDL messages.

```xml
<message name="ExecuteProcess_gdalinfoRequest">
  <part name="DataInputs" element="wps:ExecuteProcess_gdalinfo"/>
</message>

<message name="ExecuteProcess_gdalinfoResponse">
  <part name="ProcessOutputs" element="wps:ExecuteProcess_gdalinfoResponse"/>
</message>
```

Listing 7, shows how the WPS process Input/Outputs are mapped to their WSDL equivalents.

Listing 7. Using WPS’s DataInput and ProcessOutputs to define I/O WSDL messages.

```xml
<operation name="ExecuteProcess_gdalinfo">
  <input message="tns:ExecuteProcess_gdalinfoRequest"/>
  <output message="tns:ExecuteProcess_gdalinfoResponse"/>
  <fault name="ExceptionResponse" message="tns:ExceptionResponse"/>
</operation>
```

Taverna will use WSDL message parts as WPS DataInputs and ProcessOutputs XML sections that in turn contain child elements describing I/O (Listing 8).

Listing 8. Using schema definition inside WSDL <type> describing WPS DataInputs/ProcessOutputs content, Taverna will use this information to determine the necessary I/O to run the process.

```xml
<!- Input description - WPS DataInputs ->
<schema targetNamespace="http://www.opengis.net/wps/1.0.0">
  <element name="ExecuteProcess_gdalinfo">
    <complexType>
      <sequence>
        <element minOccurs="0" maxOccurs="1" name="nogcp" type="xsd:boolean"/>
        <element minOccurs="0" maxOccurs="1" name="stats" type="xsd:boolean"/>
        <element minOccurs="0" maxOccurs="1" name="hist" type="xsd:boolean"/>
        <element minOccurs="0" maxOccurs="1" name="mm" type="xsd:boolean"/>
        <element minOccurs="0" maxOccurs="1" name="nomd" type="xsd:boolean"/>
        <element minOccurs="0" maxOccurs="1" name="checksum" type="xsd:boolean"/>
      </sequence>
    </complexType>
  </element>
</schema>

<!- Output description - WPS DataOutputs ->
<schema targetNamespace="http://www.opengis.net/wps/1.0.0">
  <element name="ExecuteProcess_gdalinfoResponse">
    <complexType>
      <sequence>
        <element name="stdoutResult" minOccurs="1" maxOccurs="1"/>
      </sequence>
    </complexType>
  </element>
</schema>
```
The WPS 1.0.0 schema defines ComplexData elements as ComplexDataType that in turn extends the abstract xsd:anyType (Fallside and Walmsley, 2004). The anyType allows for an XML element to contain any sort of data; string, float, XML, etc. This allows any type of XML and raster content to be embedded in the WPS XML request as I/O, and to be consumed/processed. However, use of anyType is considered a bad practice\(^\text{16}\) with an erratic behaviour according to the APIs being used, it should only be used as last resort.

The WSDL development took into consideration the use of the WPS native ComplexDataType.

**Listing 9.** Using ComplexDataType in WSDL description.

```xml
<schema targetNamespace="http://www.opengis.net/wps/1.0.0">
  <element name="ExecuteProcess_RasterBufferProcessResponse">
    <complexType>
      <sequence>
        <!-- tested implementation -->
        <element name="rasterResult" minOccurs="1" maxOccurs="1" type="wps:ComplexDataType"/>
        <!-- actual implementation explicit definition of anyType-->
        <element name="rasterResult" minOccurs="1" maxOccurs="1"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

For example, the WSDL description in **Listing 9** was not implemented since it would return the content as child of the parent element in case of raster or XML contents (Listing 10).

**Listing 10.** Returned example when using wps:ComplexDataType defining element rasterResult, containing a base64 raster encoding.

```xml
<rasterResult>
  SUkqAAgAAAAQAAABAwABAA....
</rasterResult>
```

This behaviour was different if the type attribute was missing, returning raster content without its parent element, allowing for raster images to flow freely between services without the extra overhead of removing the parent element.

3.2.1. WSDL exceptions

WPS exceptions were also integrated in WSDL within the operation section by a fault message (tns:ExceptionResponse) pointing to WPS’s exception structure (ows:Exception) ([Listing 11]). This means that the WSDL parsing system will understand that a WPS Exception is a system error.

**Listing 11.** Generic WSDL example reporting WPS exceptions.

```xml
<message name="ExceptionResponse">
  <part name="msg" element="ows:Exception"/>
</message>

<operation name="ExecuteProcess_fastBuffer">
  <operation xmlns="http://schemas.xmlsoap.org/wsdl/soap/"
    soapAction="http://localhost/wps.cgi/ExecuteProcess_fastBuffer" style="document">
    <input name="ExecuteProcess_fastBufferRequest">
      <body xmlns="http://schemas.xmlsoap.org/wsdl/soap/" use="literal"/>
    </input>
    <output name="ExecuteProcess_fastBufferResponse">
      <body xmlns="http://schemas.xmlsoap.org/wsdl/soap/" use="literal"/>
    </output>
    <fault name="ExceptionResponse">
      <fault xmlns="http://schemas.xmlsoap.org/wsdl/soap/"
        name="ExceptionResponse" use="literal"/>
    </fault>
  </operation>
</operation>
```

3.2.2. WPS asynchronous call support

WSDL 1.1 unlike WPS does not support asynchronous calls. In async mode, WPS returns a statusURL as response, pointing to an XML document which contains the final outputs. This follows a pull model, where the client has to query the WPS to obtain process status and final response.

During WSDL development two possibilities were explored

- Overloading operation.
- Splitting single WPS process into two operations (sync and async).

The WSDL standard (Christensen et al., 2001) states that operation names are not required to be unique, therefore it is possible to have two operations with identical names but different I/O messages. According to the I/O submitted/received, a client would pick the proper operation description (this is identical to function overloading in C/C++, JAVA, C#). A WPS process that is represented as an operation in WSDL could have a sync or async representation according to the output message. WSDL overloading is not a commonly supported feature and neither Taverna nor BPEL support it. Therefore, the second option was adopted using the decision tree (Listing 12).

**Listing 12.** Decision tree adopted by PyWPS to determine if a WPS process should have one WSDL operation (async) or two WSDL operations (async and sync).

```python
if process supports (status=True and storeExecuteResponse=True):
    - One service called ExecuteProcess_\{Process1 \}
      - Normal input and output
    - One service called ExecuteProcessAsyn_\{Process1 \}
      - Normal input
      - statusURL as only output
else:
    - One service called ExecuteProcess_\{Process1 \}
      - Normal input and output
```

Fig. 3. Workflow structure: green boxes represent WPS services, purple boxes are I/O XML generators, dark pink boxes are internal beanshell service (data transformation), grey boxes are constant values and blue boxes represent outputs.
The client/user decides if it is necessary for the service to be called in a sync mode. Taverna allows the use of conditional clauses (if-else) and error detection during web services calls and a workflow could call the async process if the sync process had a time-out error caused by high computational needs. Async services have to be queried in a loop structure, meaning the returned statusURL has to be constantly queried until the status response document is populated with content.

3.2.3. WSDL generation

The new PyWPS release automatically generates a WSDL description from any WPS function (which merely needs to be a suitable Python process). From both the user's and process provider's perspective, this is ideal—it minimises developer effort and ensures consistency between WPS and WSDL interfaces to the same process.

The WSDL XML document is automatically generated using XSLT (Extensible Stylesheet Language Transformation), by converting a WPS XML document obtained from a describeProcess request of all the processes being served (request=describeProcess&identifier=all). This approach is different from the proxy method presented by Sancho-Jiménez et al. (2008), where a proxy intercepts the services and dynamically generates a WSDL, but similar to method used by Brauner and Schaeffer (2008) to generate a WSDL document that will then be used in a BPEL engine. When a client requests the generic WSDL document (http://hostname/WPSname?WSDL), PyWPS launches an internal describeProcess and applies an XSLT to it resulting in an updated WSDL document. I/O identification names are checked using regular expressions (as XSLT extension) and any illegal start characters are dropped.

3.2.4. PyWPS SOAP processing

PyWPS checks the client input for a SOAP envelope: if found, it calls an internal SOAP module that checks the message element content. If the message contains a standard WPS XML request the SOAP structure is removed and the WPS XML request is then

[Fig. 4. Taverna workflow accumulation result view. In the result view the final workflow results are presented.]

17 http://wiki.rsg.pml.ac.uk/pywps/Async_Request.
forwarded and processed. In the case of compressed SOAP content, PyWPS uses XSLT to transform the content into a default WPS XML request. XSLT transformation checks which process is being called and gathers information from PyWPS classes (using XSLT extension functions) to determine the correct I/O identification names: this is necessary to deal with cases where processes have I/O identification names that are not XML compliant. If a request uses SOAP then the response is also given in a SOAP envelope.

4. Workflow example

A simple workflow example (Fig. 3) has been set up using Plymouth Marine Laboratory—Remote Sensing Group WPS services. These comprise three WPS instances: generic, vector and raster processes. Raster and vector WPS instances are based on the WPS-GRASS-Bridge project RC2. This project uses the WPS module description of GRASS 7.0 (that each module outputs) to automatically generate PyWPS processes, wrapping the GRASS modules in PyWPS.

The example workflow is directly comparable to that used by Li et al. (2010) demonstrating the use of GRASS module r.watershed as an orchestrated web service using WSDL and Java code wrapping. In contrast, the approach presented here does not require manual programming of wrappers, since WSDL/SOAP descriptions are automatically generated for the GRASS modules ported by the WPS-GRASS-Bridge project. The final result is identical to Li et al. (2010).

The workflow accepts a DEM (Digital Elevation Model) and a threshold value that is the minimum size of the exterior watershed basin. A DEM and numerical threshold value is sent to r.watershed for calculation, the accumulation result is used to feed two auxiliary processes: r.math and geotiff2png. The first service is identical to GRASS r.mapcalc and is used to filter pixels below a certain value by setting them to null using the GRASS equation output = (if(a > 100, a, null)), where a is an input raster and output the final result, the geotiff2png process is a service that converts geotiff file formats into PNG, facilitating the visualization of results.

The final resulting images are presented in two output boxes, outAccum (Fig. 4) and outAccumNull (Fig. 5).

5. Conclusions

WPS orchestration using non-BPEL tools/approaches is possible and feasible, as shown in this paper, by extending WPS to use properly developed SOAP and WSDL. The use of compressed SOAP allows for I/O identifiers inside WSDL, identical to the ones used in a WPS service description. This, in combination with the existing capabilities of PyWPS, allows pre-existing Python code to be simply exposed as a SOAP enabled WPS service. The resulting WSDL enables a better graphical representation in generic workflow GUI software like Taverna (as compared to the use of a single XML structure as process I/O), facilitating the development of complex scientific models using multiple WPS processes without requiring programming skills and orientating the scientist’s effort into problem solving instead of computer code development. Existing eScience systems such as myExperiment enable the developed workflows to be published and reused by different users.

---

Fig. 5. Detailed workflow accumulation result. The filtering of pixels below a threshold (using r.math) provides a more comprehensive result that could be used to generate a vectorial content using r.to.vect command/service.
Async and sync calls to a WPS are possible by splitting a single process into two WSDL services, where one service returns the WPS output (sync) and the second service returns the status document URL that will be used to retrieve the final WPS output document. Error propagation of WPS exceptions is feasible when using SOAP fault structure associated to a WSDL fault message. The use of compressed SOAP structure raises implementation problems when the I/O identifier text contains illegal start characters, but these problems were overcome by using regular expressions and XSLT extension functions means it is possible to overcome the problem. Requesting an output as reference is not possible (in the current development): future developments will focus in this specific problem and can constitute a workflow bottleneck when dealing with high volumes of data.

Future developments of WPS should take into consideration the WSDL approach of tight content definitions, where XML I/O description should follow the OGC formats and their object types and not a generic simple anyType. Future WPS identifier elements should restrict the character content to those allowed in W3C XML element name, this would make any XSLT transformation more robust.

The Taverna/SCUFL approach to orchestration, where data flow through web services as they become available, is a viable alternative to BPEL (which is more process oriented), and is more flexible with changing workflow structures. SCUFL and its new 2.0 version should be considered as a viable direction in future OGC service orchestration research.

A web based graphical editor is being developed based on the WireIt! API.27 The XML output of this editor will be compatible with the Taverna server allowing it to be used as a web based orchestration engine. This web editor will be found in the future European Marine Information System (EUMIS) pilot, part of the NETMAR project.

Acknowledgments

The NETMAR project (no. 249024) is partially funded by the European Commission under Theme ICT-2009.6.4 ICT for environmental services and climate change adaptation of the Information & Communication Technologies FP7 Programme.

We are grateful to Dr. Bastian Schäffer for providing bibliography, used in this paper.

References


Cepicky, J., 2006. GRASS goes web: PyWPS. In: Free and Open Source Software for Geoinformatics, Lausanne, Switzerland.


27 http://neyric.github.com/wireit/.