NeuroLOG ANR-06-TLOG-024
Software technologies for integration of process, data and knowledge in medical imaging

Deliverable L8: processings composition on a distributed system

Javier Rojas Balderrama MODALIS (I3S) javier@i3s.unice.fr
Johan Montagnat MODALIS (I3S) johan@i3s.unice.fr

Abstract

The NeuroLOG platform supports the integration of new image processing tools and the building of processing chains describing neurology experiments. The underlying execution of such experiments may be distributed over the resources provided by multiple sites participating to the federation or remote grid resources. This deliverable documents the processing tool integration mechanism adopted to achieve these goals.

1 Introduction

The NeuroLOG middleware provides image analysis tools integration into their platform in order to support image analysis pipelines design. The solution proposed is based on several software components that have been integrated into the NeuroLOG prototype platform currently deployed. This document gives an overview of the existing system and it is meant as a technical documentation for users of these services.

The objectives of WP4 (*design and execution of distributed application on a distributed infrastructure*) are to:

- enable the integration of image analysis tools into the platform;
- enable the remote invocation of these tools;
- publish these tools for discovery and usage by neurologists;
- provide access control to tools execution;
- enable the design of image analysis pipelines; and
- control the execution of complete pipelines over data sets.

The image analysis tools developed by the NeuroLOG neuroscience centers are command-line applications meant for atomic run on an (or a group of) image. To address WP4 goals, the software developed is structured in three components:

1. A toolbox for service management, namely ***j*GASW** (*java Grid Application Service Wrapper*), embedding traditional command-line application into a service-oriented framework by providing a Web Service invocation interface to the wrapped application. Moreover, *j*GASW provides additional instrumentation concerns such as tools relocation and client interfaces.
2. A tools container, currently the **Apache Tomcat** services container, for exposing the wrapped tools to the NeuroLOG clients. This server includes the Sun Metro Web services stack to run the Web services.
3. A workflow designer and manager, **Moteur2**, which can handle large data flows and provide efficient enactment.

The *j*GASW toolbox is described in section 2 and its graphical user interface in section 3. The *j*GASW remote execution capability is discussed in section 4 and its invocation interface is documented in section 5. The simple web services catalog designed to date is then described in section 6. The Moteur2 workflow engine is finally documented in section 7.

2 *j*GASW toolbox

Most image analysis tools developed in the neuroscience community are neither designed for integration in a standard environment nor for tool relocation and remote invocation. The *j*GASW toolbox aims at providing a wrapper shell for applications using a *Command-Line Interface* (CLI), using the Web Service (WS) standard invocation interface, in order to ease the integration of these tools into the NeuroLOG platform.

*j*GASW provides much more than a mere invocation interface to the service. This tool provides a complete mechanism to package tools and their dependencies into an archive, relocate this archive on a given server, deploy and publish it. As a submission interface, *j*GASW is also involved with security (access control to the service execution) and grid invocation (including files transfer for proper processing on a remote resource). *j*GASW therefore provides a full range of functionality, making the applications autonomous, relocatable and grid-compliant. It also addresses grid execution performance.

The *j*GASW toolbox is composed by three elements: (1) a service wrapper, (2) libraries to manage the relocation of resources, and (3) a client programmatic interface to invoke the wrapped applications. The wrapper and the libraries are based on a data modeling describing CLI applications and the way to execute them. In contrast, the client interface is an autonomous service consumer.

2.1 Data modeling

The description of a CLI application is made using a XML schema listed in the Appendix C. This representation allow us to declare all options and dependencies associated to a CLI application. *j*GASW uses Java objects to access the instances of this schema. The mapping between Java objects and serialized representation in XML are performed through an automated data binding transformation.

2.1.1 Data binding

Data binding gives and object view of underlying XML data without losing access to the original information, and delivers performance benefits using unmarshalling and efficient methods to access XML schema build-in data types.

Several Java data binding tools including JAXB, Castor, EMF and JiBX are available. However, they are partial implementations of the W3C Schema specification or case oriented like relational mapping. On the other hand Apache XMLBeans[1], an open source tool based on the StAX specification, provides a data binding mechanism by automatically creating a mapping between elements of a XML schema to bind, and members of a class to be represented as objects in memory. XMLBeans takes advantage of the richness and features of XML giving a full schema support and the corresponding Java classes; provides a XML fidelity keeping the full infoset after unmarshalling in a XML instance; and honors schema constraints. The election of XMLBeans is based on these arguments, the large adoption in the community, and support.

The XML schema is the starting point for XMLBeans development. This schema provides a data model that enables to express the structure and constrains designed in the schema. The XMLBeans technology is used to generate Java classes and interfaces associated to all elements defined in the XML schema. The resulting XMLBeans classes are able to parse any instance document that conforms to the schema. Also, and instance document can be created by manipulating these classes.

After the XMLBeans code generation, an intermediate step consists in an code adjustment by means of a adapter pattern to reflect the interfaces declared as high level representation of the resource. This step simplifies the access to declarations leaving the complexity of the original data binding hidden for later instances declarations.

[1] http://xmlbeans.apache.org

2.2 Service wrapper

The process to expose an application by way of Web services begins with the generation of personalized code that reflects the resource description: Java bean code to invoke the business code of the service; intermediate method stubs; and configuration files to deploy the resulting byte-code on the service container.

2.2.1 Template processor

The description of a CLI application of section 2.1 abstracts a resource. Now the principle consists to transform that description into a WS interface. In *j*GASW this transformation is based in a template engine that provides Java code and the necessary files to let the service container interpret this code.

Apache Velocity[2], is an open source tool that defines a simple template language used to create and render documents that format and present a data model. Velocity aims at ensuring a clean separation between the representation and the business tiers using *context* objects and merging them with the data using a template to produce the resulting document. The context object is a central concept to Velocity. This context is the carrier of data between the Java layer and the template. Since the data model is represented as objects, Velocity takes them directly becoming accessible via the references defined in the template and substitutes the values with the instance of the description.

The template-based procedure generates (1) the WS interface using the JAX-WS specification (see 2.2.2); (2) the configuration file needed by the WS messages interpretation engine in order to publish the service; and (3) the configuration file of the server container to associate the Java code to the WS servlet engine. Each (set of) file(s) has its own template and is combined with the description of the application using the data modeling representation described in the section 2.1.1.

2.2.2 WS implementation and stubs generation

Basically Web services are created according to two paths: the top-down or "Contract first" based on the initial declaration of the WSDL document; and the bottom-up or "Implementation first" working with the code and later generating the WSDL associated to that code. The bottom-up approach is a suitable scenario for *j*GASW because the service interface is generated in a Java bean and the data model is already defined.

Under Java different specifications exist to build services. Some of them are isolate efforts such as Apache Axis implementations, and others are based on Java Specification Requests. The latest specification for WS applications and clients is the Java API for XML Web services (JAX-WS). This specification replaces the JAX-RPC API reflecting the move away from RPC-style. JAX-WS is the "modern" Java SOAP implementation of Web services making extensive use of the annotations mechanism introduced in Java 5 and strategically aligns itself with the current trend towards a more documented-centric messaging model.

The use of annotations simplifies the implementation and eases the service development. Based on a Plain Old Java Objects, containing the implementation of the WS interface, the annotations are included in the code describing details such as Service identification, SOAP binding, namespace and operation descriptions, among others. All these details are instantiated during the merging step of the code generation and they are used to be compiled into byte-code assuring better platform independence for Java applications.

[2] http://velocity.apache.org

JAX-WS uses JAXB as default data binding to process the message marshal/unmarshalling. These operations map the Java types into WSDL types and vice-versa. The resulting mapping called WS method stubs are part of the final service and they are used to communicate with the client all along the invocation. We use the `wsgen` utility to generate those stubs. This pragmatic choice requires an external call of the utility after the code generation during compilation. In the future, another programmatic approach using APT-Jelly[3] could be implemented to reduce external dependencies of *j*GASW.

It is important to quote that some circumstantial adoptions are inevitable. While the use of URI Java type ensures the correct manipulation of any protocol, unlike the URL Java type which is limited to a set of predefined protocols, it is not possible to map directly *j*GASW parameters defined as `URI`s into URI Java objects. Indeed, JAXB maps a URI to a `xs:string` instead of `xs:anyURI`, making it indistinguishable from any string. Alternatively, *j*GASW internally uses URLs. This use carries some consequences: the URL Java type does not support grid protocols, so an extension of the URL handler is required.

In terms of solutions supporting JAX-WS basically three products contest the audience: Sun Metro, Apache CXF and Apache Axis2. The first two implements all the JAX-WS specification and they are distributed on major application servers. Axis2, on the other hand, provides an interesting alternative but is based on its Axis services approach giving to JAX-WS a marginal dedication. Each of these products needs to be configured on the basis of Tomcat and the deployed services have to fulfill the formatting and configuration for deployment.

*j*GASW uses Sun Metro[4], the JAX-WS Reference Implementation, that provides the stack engine to publish services. It is easily integrated with Tomcat and supports additional needs as MTOM, useful for the service attachments manipulation.

2.2.3 Packaging and deployment

Following the Tomcat server schema, all services are deployed in form of a *Web Application Archive* (WAR), a special JAR file used to distribute a standard Web application. In the case of *j*GASW this archive includes configuration files (`sun-jaxws.xml` and `web.xml`); the description of the resource (`description.jgasw`); the wrapped CLI application; the dependencies; and the Java classes representing the WS interface and the stubs. This WAR file is created automatically using one of the *j*GASW interfaces of the wrapper as is detailed in the Section 3.

The Tomcat server is not configured to support JAX-WS out of the box. Metro should be installed with Tomcat before hosting any service. Additionally the set of *j*GASW libraries has to be accessible in the class path. These libraries implements all the logic associated to the relocation described in the Section 4.

The last action to publish a service is to deploy the WAR archive on the Tomcat `webapps` directory. Tomcat sets up services at run-time without perturbing its normal operation (*i.e.* there is no need to restart the server). This quality is known as *hot deployment*. Just after the deployment, the WSDL describing the service is available through the annotations inserted in the Java code and the service is ready for invocation. Removing the deployed services releases safely the reference to the service from the container; and from the catalog.

[3] APT-Jelly is a Java engine for generating artifacts from source code by providing and interface for Sun's Annotation Processing Tool to a template engine. See http://apt-jelly.sourceforge.net

[4] https://metro.dev.java.net/

3 User interfaces

The user interfaces of *j*GASW allows us to create and deploy new services. The procedure aims at being as simple as possible, filling the description form that includes eventual dependencies of the application and the details of all arguments. This section describes the graphical interface shown in Figure 1 and the equivalent command-line utility.

3.1 Graphic interface

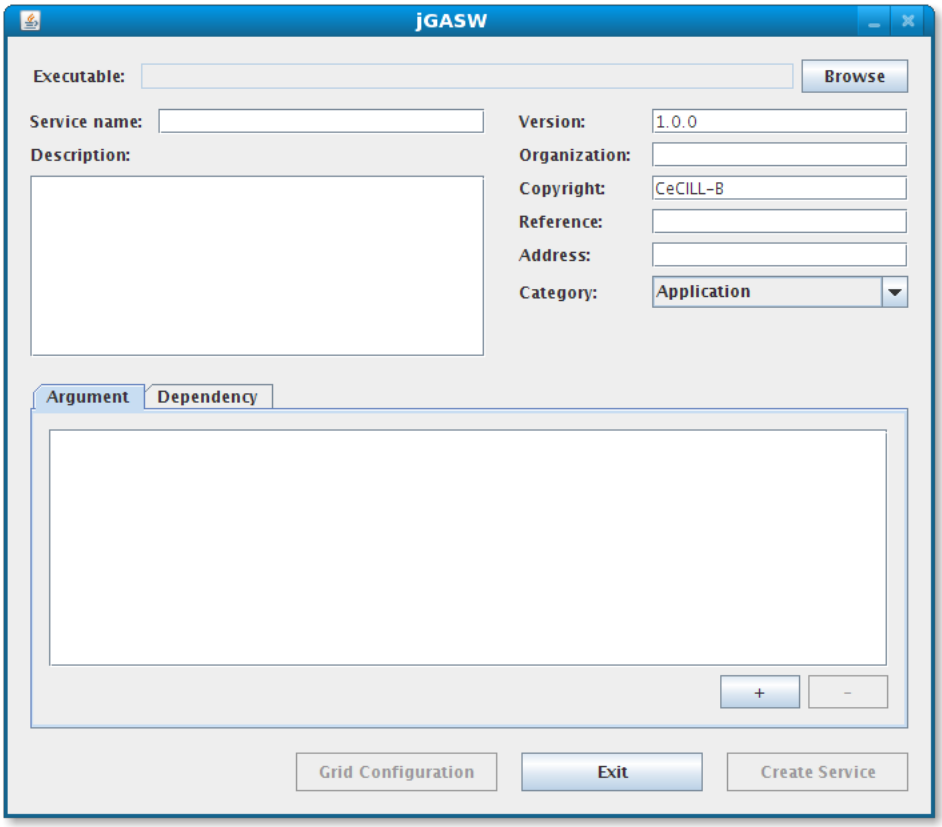


Figure 1: Main *j*GASW GUI panel

The main window includes all relevant fields of the CLI application. All these fields are used to generate the description. Below there is a brief description of each element of this form.

Service name is the symbolic name of the service. This name is part in the construction of the WSDL URL path.

Version is the declared version of the application. This version as the **Service name**, is used to build the WSDL URL.

Description is the extended description of the application. Even if is purely informative it should describes a detailed description of the CLI application including a command-line example.

Organization is the name of the organization who owns the application.

Copyright is the licence category of the original application.

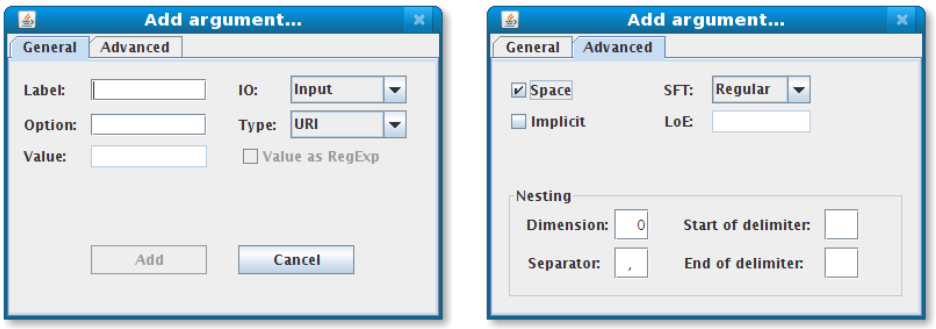
Reference is a reference or key name associated to the application.

Address is the contact email address of the person in charge of the wrapped service.

Category is the application's file type. The possible values are `Application` or `Command`.

Executable is the path of the application (loaded using the Load button).

Arguments can be associated to the application in the description using the add/remove (+/−) buttons in the Argument tab pane. Once the add button, is clicked the dialog shown in Figure 2(a) is displayed.



(a) General options pane (b) Advanced options pane

Figure 2: Argument declaration dialog

An argument can be described using the following fields:

Label the reference's name of the argument. This label is just a reference freely named by the user. If the **Type** is declared as `Input` this label is used as argument name of the Web service.

Option the command-line option associated to the argument. For instance the POSIX-style defines options with simple dash character followed by a letter or double dash followed by the option name.

Value is the exact content of the argument. The value is only taken into account when the **IO** is not declared as `Input` because input values are provided invoking the service. The value can contain a case sensitive string or a regular expression (a Perl Compatible Regular Expression indeed). A second alternative is designed for output arguments with variable number of entries in the Advanced tab pane.

IO the argument's type, namely: `Input`, `Output` or `Flag`. The `Flag` type can be used to provide constant values of arguments that are not inputs nor outputs. When the **IO** is declared as `Output` the argument's **Type** is always a `URI` because the WS output are files all the time. This is the mechanism to present results and the output of the execution.

Type the primitive type class associated to the argument. The possible values are `string`, `double`, `integer` or `URI`.

Additionally, advanced argument options (shown in the Figure 2(b)) may be set on the Advanced pane. These options add richness to the classical description of an argument and is provided for special situations as implicit arguments, special file types and the declaration of nested arrays. The advanced options includes:

SFT or Special File Type assumes as default that an argument of type `URI` is a simple `Regular` file. The behaviour is modified using the `Expand`, `Replace` or `Directory` options. `Expand` and `Replace` are used in combination with the list of extensions **LoE**. When `Expand` is selected the argument declared with a specific `Value` is concatenated to each extension of the **LoE** and then add all the combinations to the command-line argument. If the `Replace` option is set the argument declared with the `Value` is used in the command-line but the reference a this file is resolved using all the extensions of the **LoE** (see example at section 8). Finally when the option `Directory` is selected the declared `Value` is considered as a directory file name and all files included inside are associated to this argument.

LoE or List of Extensions is used to declare all file extensions associated to the declared `Value` in combination with the **SFT**.

Space In some cases an argument **Value** is declared next to the **Option** without any space between them. This type of declaration is set checking the box.

Nesting All elements needed to declare the argument as a array of a given **dimension** are declared on the nesting frame. For that the **begin** and **end** delimiters (blank space by default) and the **separator** character are set.

Like arguments, we can also include dependencies. Typically dependencies are external libraries to run an application. Sometimes other binaries are included for the execution, for example the binaries to be run in the main script. To achieve dependencies inclusion click the add button on the Dependency tab pane. A simplified dialog to the main one is displayed as shown in Figure 3.

All dependencies are "resources" so the description is similar to the main application with an appropriate **Category** on the combo-box. Finally when the button Create Service is clicked, the service is deployed on the container directory of Tomcat. If the configuration is not set, using the *j*GASW properties listed on the Appendix B, a Save file chooser appears to save the corresponding file on the system. It means, services can be created without a server and then we can deploy the service manually into a server.

There is also an alternative method to load a service description from the command line. The option '`-l`' with the name of the file when the *j*GASW application is lauched will load the description on the graphical interface. If some paths of the description are invalid these will be highlighted on red.

3.2 Command-line interface

The command-line utility is an alternative interface to create services using *j*GASW without using the graphical interface. This alternative is appropriate for testing purposes or users that only want to modify their service slightly without filling again all the description. The use of this interface however, supposes accurate knowledge of the *j*GASW XML schema.

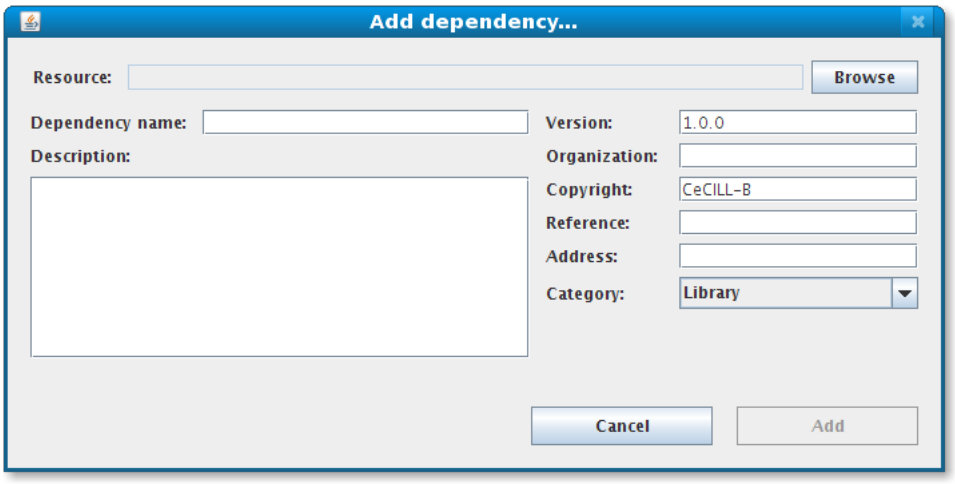


Figure 3: Dependency declaration panel

The utility is an option to wrap an application from scratch combining the use of the command-line tools provided by XMLBeans[5]. The utility `xsd2inst` is useful to create a empty description instance based on the schema. Then the `validate` tool helps to verify the correctness of the description. The list of steps to create service from CLI is the following:

1. Run the command: `xsd2inst resource.xsd -name description.jgasw` to create a empty description.
2. Complete the generated description file.
3. Validate the description running: `validate resource.xsd description.jgasw`
4. Copy the description and all necessary resources into a directory.
5. Execute: `java -jar jgasw-ui-<version>.jar -c <directory name>`

This procedure creates a WAR file using the information found at the description, encapsulating all resources and generating the code to wrap the application as a Web service. This service can be deployed on the Tomcat server as usual.

4 Tools relocation

The core functionality of *j*GASW beyond the description of CLI application is the instrumentation of the logic related with the interpretation of arguments, dependencies configuration and the execution. This operative process is organized in three groups: data management, local execution and execution on the grid.

[5] http://xmlbeans.apache.org/docs/2.0.0/guide/tools.html

4.1 Data management

One fundamental feature of *j*GASW in case of file transfers is the delegation of this task to external tools. It means, the instantiated service does not transfer the results declared as files, and only provides references to them. The design of this feature responds to the necessity of avoiding the potential bottleneck of large amount of data transfers that certainly it is not part of the service execution. By contrast, if the service receives references to files as arguments, these are fetched to the effective point of execution managing the different protocols supported by *j*GASW, namely: FILE, HTTP, HTTPS, FTP, LFN or GSIFTP.

After the execution of an application two scenarios are figured out regarding data. In the fist scenario, if the resulting files are potentially inaccessible to the remote client *j*GASW puts available into a public space all the resulting files. Usually this context happens in a local execution where the outputs are defined using the FILE protocol so, a translation of the reference is possible in favor of other protocol such as HTTP. In the second scenario, *j*GASW registers all resulting files product of a grid execution on a Storage Element, then references to those files are reported to the client. Naturally, in both cases files can be delivered to the client using an additional data transfer operation.

4.2 Local execution

The instrumentation of the local execution interprets the description of the application building the command-line to execute. On the server, a isolated sandbox is created and the execution is then performed retrieving all necessary data to the sandbox and configuring the dependencies properly. The local execution is the simplest *j*GASW run-time instrumentation. The application runs in the same place where the service is hosted, for this reason multiple instances of heavy-demanding applications are not suitable and a remote relocation on a production grid is contemplated.

4.3 Grid execution

The grid execution allows several remote executions of an application transparently but requires remote relocation of resources. This execution implies use of several components of the grid infrastructure such as Workload Management System (WMS), Storage Elements, the Logging and Bookkeeping service, etc. The correct execution on the grid involves strategies from submission to monitoring procedure.

A sequence diagram of all steps during the grid execution is presented in the Figure 4. The diagram shows a the user invocation of the *j*GASW operation corresponding to the grid execution (1). Next the application is submitted to using a WMS (2) and the real execution is delegated to a computer element (3). The Web service acting as submitter obtains a job identifier to trace the execution progress (4). Later, the grid execution begins getting the necessary data from the source (5). Immediately the application runs and the Web service checks the status periodically (loop). After a normal execution the results are saved on a Storage Element (12) and finally the references to those results are returned to the client (15).

4.3.1 Submission channels

We contemplate two submission channels for the grid execution. Job submissions through the programmatic approach using the gLite interface to the WMS (WMProxy API) and

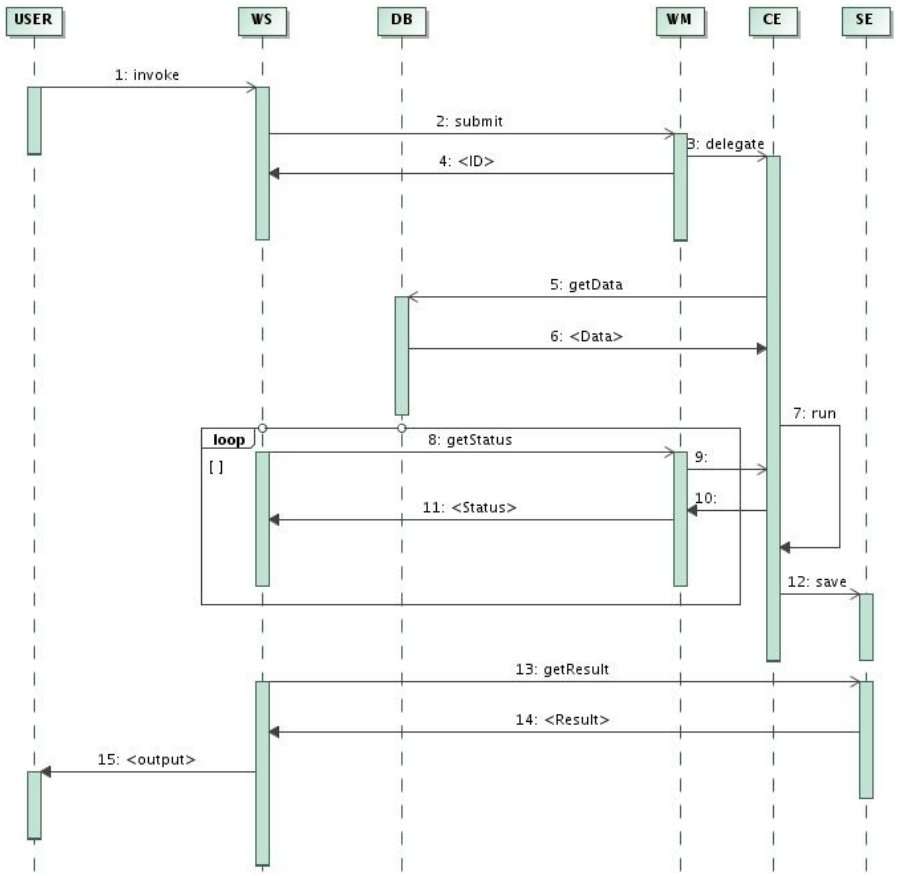


Figure 4: *j*GASW grid execution

alternatively the job submissions by means of a grid UI. The programmatic channel at fist sight suitable for *j*GASW since everything is managed from the JVM, presents scalability weaknesses because the service container resources quickly reaches their limits. Likely, this is an issue to be fixed but is out of the *j*GASW scope. In the same way, the submission by means of a grid UI reveals other challenges. The job description file is transferred alongside the application and all the other dependencies to the grid UI and then copied to the grid. This transfer requires the management of a new security layer to use the SSH pathways that assures the connection between the grid UI and the submission manager. Moreover, this last submission channel is not efficient because data is transferred twice: from users to the grid UI and from grid UI to the grid.

4.3.2 Job submission policies

Job submission policies improve the rate of successful executions. We refers basically to three policies [3]. First, the single resubmission using a timeout before submitting again the job and cancel the previous submission. Second, the multiple submission using a collection of the same job and cancelling all but the first job that starts the execution. Third, the delayed resubmission. This last policy submit periodically a copy of the job without cancelling and it iterates the process until at least one job starts. We are interested in the implementation of these policies because latency has an important impact in normal

execution on grids. This issue may be overcome using job submission policies replacing the default job submission mechanism used by the WMS. As a result, we may reduce the final execution time that is the result of the addition of effective execution time of the job and the latency associated to the submission. Alternatively meta-schedulers like GridWay[6] enable users efficient and interoperable executions but add a layer to the technology stack.

4.3.3 Fault tolerance

Fault tolerance is other property to reach in favor of quality of executions. Usually, executions on the grid face major inconvenient like system incompatibilities and resource unavailability. System incompatibility appears when the application requires special characteristics to run, for instance memory size or installed applications. In some cases, the WMS does not match compatible computers or the filtered results are not sufficient to run the application either. The resource unavailability is present when the grid is overloaded or the server responses are slow so the submission and later execution cannot start. Quality of Services using monitoring and site selection help to improve results. White and black WMS lists may be used to probe compatibility and availability before real submissions selecting reliable and fast computers.

4.4 Automatic execution

The final goal of *j*GASW is to provide an automatic execution type. Users without technical skills could find difficult the selection of execution type because they do not have a clear idea about the implications of relocation. Moreover users do not know the load status of the server nor the health of the grid. These arguments draw the necessity to provide an operation to choose automatically the type of execution in behalf of the user. The selection between local or grid execution is based on the application description, specific requirements and resource availability. The execution time and the requirement needs are the most important factors however, overhead or remote responses can also determine the type. The explicit operations still remain pertinents though.

4.5 Results manipulation

A service has arguments described with different data types and structures. In the same way the results of an execution should match the description of provided outputs. *j*GASW takes the original results of an execution and forwards them preserving that structure and data typing. The notions about such types and structures are explained in detail in the GWENDIA language proposal [6]. Briefly, the input and output manipulation of data is based on the array programming principles. *j*GASW uses primitive types to map results with data types. Scalars and arrays represent all application parameters and results. Scalars are simple elements representing a plain primitive type. In contrast, (nested) arrays are groups of elements of same type structured according to the dimension of the array. In the same way, the workflow enactor processes data according to the GWENDIA specification using the *j*GASW programmatic client described in the section 5.

Most of the time, when the results are not files they are presented in the standard output as sequences of strings. After execution is imperative to interpret these outputs using the description of arguments. This task involves parse, cast and map the result in

[6] http://gridway.org

the right structure and finally provide this structure as service result. *j*GASW reproduces as much as possible the structure organization resulting from an execution. Nevertheless this is not trivial and some agreement of structure is defined using delimiters of arrays and element separators. If an application provides a different result format the output should be adapted to a *j*GASW-compatible interpretation.

4.6 Extensions

Implementation of other mechanisms of submission on grids, using for example the DIET[7] toolbox, are natural extensions of the execution process for future milestones. The design of the core module of *j*GASW abstracts the notion of an executor enabling the possibility to add new instances. These instances do not affect the rest of the project organization and improve at the same time the re-usability of other *j*GASW components. Nevertheless, where the relocation of resources is not exclusively yielded to *j*GASW and depends on other mechanisms of deployment the effort to implement an extension is not really profitable.

Other kind of extensions are possible inside *j*GASW. The integration of external concerns (non-functional requirements like access control or logging) have been implemented during the integration of *j*GASW with the NeuroLOG middleware. In fact, changes do not affect the data model nor the core application and it is just reflected during the generation of the Web Service interface. Only the template interface, have to be adapted injecting the modules of the middleware that implement those requirements. Then the byte-code is compiled by setting a comprehensive class path to those modules. In NeuroLOG, to carry such libraries to the client side do not bring any benefit, for that reason the generation of the WAR file includes only Java source code. The code is compiled on the server and immediately deployed. Specifically, the NeuroLOG security framework (cf. [1]) has been reused to extend *j*GASW with strong and distributed security policies to prevent unauthorized invocations. Support another concerns such as semantic knowledge inclusion or monitoring are good sources of additional potential addendums.

5 Invocation client interface

A generic client API to invoke services is the third element of the *j*GASW toolbox after the client used to wrap the CLI applications and the core libraries installed on the server to enable the instrumentation. The methods of the API parses the service description, interprets the contents and creates dynamically the SOAP messages to submit the information to the server. Besides, it is possible to use the same API to invoke external Web services interpreting the operations and arguments declared in the WSDL as long as such services met the requirements of data management used by the GWENDIA proposal. This capability of generalization let's the workflow manager use the API to build pipelines with *j*GASW services combined with other pre-existent services provided by third-parties.

5.1 WSDL and schema parsers

The interpretation of different formats and styles provides all the relevant information from a WS description. A WSDL binding describes how services are bound to a messaging protocol. The WSDL binding can be either a RPC-style binding or a document style binding. It can also have an encode use or a literal one. This gives several style/use models; and

[7] http://graal.ens-lyon.fr/~diet/

an extra pattern should be added to this collection when the binding is wrapped. Other characteristic of services description associated to the models is the inclusion of the schema describing the messages and types of service. This also varies importing the contents as an external declaration or including the schema as a part of the description. All these details are considered trying to propose a simple interface.

Two parsers are implemented in order to obtain the list of services, ports and operations, the description of messages and sequences. A WSDL parser to handle the high level details of the service description, and a schema parser to manage the intrinsic details of primitive types as namespaces or dimensions of variables. From the final developers point of view only the fist parser is useful to obtain the sequence element of the operation to call. The schema parser is used by the WSDL parser internally. In the same way, the content details of the expected results are processed using the WSDL. In other words, a client can obtain all the necessary information to invoke and process an operation based on the description.

5.2 Service invocation

Using Web services the interoperability is granted between clients an server thanks to the messaging protocol independence. Consumers dispatch a well-defined message and wait for the result. This action is possible creating SOAP messages with the references retrieved from the description associated to their corresponding values and send them to the server.

The Java specification provides the dynamic and static methods to consume Web services. The conventional static method creates stubs and the exclusive use of objects takes place using the WSDL file to unmarshal the arguments. The second method involves a dynamic dispatch client that is more generic and offers more flexibility. The dynamic method is a pure XML messaging oriented client and requires advanced use of SOAP message construction and interpretation, especially if different types of response messages are considered.

The *j*GASW API client implements the dynamic method because a generic procedure for invocation is mandatory. Each service is described using a specific and different WSDL therefore a direct manipulation of messages is done during execution. Similarly the process mechanism of the server response is performed, despite the different message formats since each server provides a message that is not necessary defined in the same manner. Nevertheless to pay special attention to that concern has limits because in practice is not possible to test all types of server messages. The *j*GASW approach works based mainly on the message responses of the JAX-WS reference implementation provided by Sun Metro but gSOAP and Axis2 are supported partially.

6 Image processing tools catalog

Command-line applications wrapped through *j*GASW are deployed on the Tomcat server of each site in the NeuroLOG federation. This deployment scenario ensures reliable application services management (Tomcat is a thoroughly tested application container used in many production environments) and coherency with the rest of the NeuroLOG platform (the middleware services being hosted in the same container and sharing functionality such as access control and file transfer facilities).

Tomcat is mostly an application container, managing the life cycle of the services and providing a single entry point for clients. The Tomcat server deployed with the

NeuroLOG middleware is configured to use the HTTPS protocol for all communications with the application services: only clients with a valid NeuroLOG federation certificate are able to pass the SSL handshaking operation and communicate with the service. However, Tomcat does not provide finer access control (it does accept all valid certificates, not making any difference between different platform users), nor cataloging capabilities (it does not provide service discovery nor listing). The fine-grain access control is handled by each application service as described in section 4.6. The cataloging capability is provided by the NeuroLOG middleware. A local Tomcat server introspection method is provided: it parses the Tomcat application repositories and deployment information stored by Tomcat at runtime to build a list of applications available. The cataloging method is exposed as one of the middleware web service operations. It returns to clients the list of available services and their corresponding WSDL document, accessible from the Tomcat server through the HTTPS protocol. Clients can therefore browse the list of services and discover their interface through each service WSDL description. The *j*GASW client handles the WSDL files parsing for easy integration into the NeuroLOG software client.

The processing tools catalog provides only a limited functionality so far. It is limited to site-wise cataloging and it does not provide any high level interface. In the longer run, a catalog federation mechanism (possibly using relational information on services stored in the NeuroLOG site databases and federated through the middleware Data Federator services) will be developed to enable simple browsing of all tools available through the federated platform. In addition, tools need to be identified through more significant information than a simple short name. In a longer term, the image processing tools will be annotated with semantics information (describing service function, input and output data classes). This information will be exploited to provide a higher level tools query interface. Standard cataloging techniques (*e.g.* UDDI) will be considered if they provide sufficient support for handling and querying semantic information related to services.

7 Image analysis pipelines design

Image analysis pipelines are designed by pipelining analysis tools in order to assemble a complete data analysis chain. The *Moteur* workflow designer and execution manager [2], now in version 2, is used for that purpose. Moteur2 provides a user friendly interface for graphical design of application workflows, a very expressive framework to design data intensive applications and a client/server framework to handle multiple workflows executions.

The Moteur2 graphical editor is detailed in the NeuroLOG User Guide [4]. The designer is interfaced to the NeuroLOG middleware through a simple web service browser that enables listing of services. Users can drag and drop WS operations listed on the catalog into the workflow canvas. Through this simple mechanism, analysis pipelines are built easily embedding the image analysis tool and composing the outputs and inputs.

An application workflow is defined as a graph of application tools interconnected by data dependency links. The Moteur2 workflow enactor controls the flow of data items along the graph links and invokes each application tool as soon as any data is available for all inputs of this service. This data-flow oriented approach ensures high performance (application tools are enacted concurrently when no dependency exist between them), and expression of application parallelism in a transparent way from the user point of view (no explicit parallelism construct is needed: parallelism is implicit in the workflow graph topology and through the use of independent data sets). Moteur2 provides rich data

flow manipulation operators, known as *iteration strategies* [5] to enable the description of complex image analysis pipelines.

The Moteur2 engine only interfaces with the application tools through *j*GASW client API that facilitates the web service interface parsing and invocation. Through the *j*GASW wrapper, Moteur2 is shielded both from details of the command-line tools invocation and from the grid invocation interface (including data transfers and grid security credentials management). Its role is focused on analysing the data flow and enforcing the coherent execution of the application in a distributed environment.

In the current version of the middleware, the choice between a local or a grid execution of each service is explicitly defined through the invocation of one of the methods of the *j*GASW wrapper. Since the Moteur2 workflow services are described each by a specific web service endpoint and the operation to invoke, this information is hard-coded in the workflow description.

Although integrated into the NeuroLOG middleware as a binary dependency, the Moteur2 workflow engine is implemented independently from the NeuroLOG software stack and it can be downloaded from the Trac server at https://nyx.unice.fr/projects/Moteur2.

8 Concrete example

Some advanced characteristics of *j*GASW cannot be observed at a glance. The manipulation of special file formats, hidden arguments or dependencies, and the interpretation of the resulting outputs requires an example to look in detail the necessity of the proposed description of resources and easy invocation.

8.1 Context

BrainVISA[8] is a software that allows users to trigger sequences of treatments in series of images. These treatments are performed by calls to command-lines. These tools, are the building blocks on which is built an assembly line. One of these applications for calculation of images is AimsLinearComb[9]. It performs a sum of two brain activation maps. For instance, AimsLinearComb performs a linear combination using the formula $I_{1+2} = aI_1/b + cI_2/d + e$ obtaining a fusion of two binary functional-analysis activation in form of a new volume. An example of this command-line is:

```
$AimsLinearComb -i lwlebge.img -a 200.0 -b 1.0 -j lwdupje.img \
 -c 20.0 -d 1.0 -e 0.0 -o lwtest.img
```

In practice this tool is more complex due to some suppositions the user should know:

- The input and output use Analyze images. This kind of image consists in two files with IMG and HDR extensions. In the command-line however, only the IMG file name appears explicitly representing the *image*.
- The tool execution produces a text file with the extension MINF. This file is not expressed in the command-line but have to be included in the results with the Analyze image.

[8] http://brainvisa.info
[9] http://brainvisa.info/doc/aimsdata-3.1/aims_training/en/html/ch04.html

- AimsLinearComb needs several libraries for standalone execution: `libaimsalgo.so.3.0`, `libaimsalgopub.so.3.0`, `libaimsdata.so.3.0`, `libcartobase.so.3.0`, `libcatodata.so.3.0`, `libgraph.so.3.0`, `libsigc-2.0.so.0` and `libstdc++.so.5`. The user should configure the environment to include them in the list of the system. Typically this is possible in UNIX-like systems adding to the `LD_LIBRARY_PATH` the directory path where those dependencies are located.

Additionally the user could be aware of the standard output or error messages, so all these concerns have to be considered when the tool is wrapped as service.

8.2 Wrapping and publishing

Using the *j*GASW interface, the creation and execution of this kind of CLI application as service is possible without adding intermediate scripts to adapt arguments manipulation or file formats interpretations. Although special attention to the description declaration is expected. The procedure is summarized as follows:

- Respect the order of the arguments, *j*GASW replace them in the same order. And take into account the case of letters for the options, extensions, and values because of case sensitiveness.
- Declare arguments 'a', 'b', 'c', 'd' and 'e' as `Input` of type `double`.
- Declare arguments 'i' and 'j' as `Inputs` of type `URI` setting the **STF** value on the Advanced pane to `Replace`; and include the HDR extension to the **LoE** text field.
- Declare the argument 'o' as `Output` setting the **STF** value to `Replace` and include the HDR extension to the **LoE** text field as well.
- Declare an implicit argument (from Advanced pane) as `Output` and set the **Value** as regular expression checking the box. Then include the expression "`.*minf`" on the value field to get the generated text file in the list of results.
- Add one by one all library dependencies on the Dependencies tab setting the category of each one to `Library`.

Thereafter the service is generated and hot-deployed on the Tomcat container. The service includes the AimsLinearComb binary, libraries, the description of the application, the server-side configuration files, and the compiled code representing the WS interface and the stubs. Finally, this service is ready for execution by client invocation. The complete descriptor generated of this example is presented in the appendix D.

8.3 Execution

The service WSDL is interpreted with the description parser operations. The arguments below are identified as inputs:

- `i1;{http://www.w3.org/2001/XMLSchema}anyURI;0`
- `num1;{http://www.w3.org/2001/XMLSchema}double;0`
- `den1;{http://www.w3.org/2001/XMLSchema}double;0`

- `i2;{http://www.w3.org/2001/XMLSchema}anyURI;0`
- `num2;{http://www.w3.org/2001/XMLSchema}double;0`
- `den2;{http://www.w3.org/2001/XMLSchema}double;0`
- `cst;{http://www.w3.org/2001/XMLSchema}double;0`

Each item describes a semicolon separated list with the label of the argument; the data type with the respective namespace; and the dimension of the input.It means the request message should provide a sequence carrying out all the operation arguments.

The expected values of AimsLinearComb are two elements: the text MINF file and the Analyze *image*. By default *j*GASW add the contents of standard output and standard errors as files to the results. In short, after successful invocation a list of five file references are present by the service consumer.

9 Conclusions

The *j*GASW provides a rich description of CLI applications and execution procedure generating a Web service that encapsulates all the associated resources to be instrumented using different interfaces. This approach enables compositions of pipelines using Moteur2 with strong type mapping and complex structures. This solution is a step forward the conciliation of CLI applications with modern service oriented architectures providing a clean and simple set of tools to assist scientists that are not computer specialists to build, run, combine, and share their work.

References

[1] Alban Gaignard, Johan Montagnat, and David Godard. Security policies set-up. NeuroLOG Deliverable 10, I3S Laboratory, Sophia Antipolis, 2009.

[2] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR. *International Journal of High Performance Computing Applications (IJHPCA)*, 22(3):347–360, August 2008.

[3] Diane Lingrand, Johan Montagnat, and Tristan Glatard. Modeling user submission strategies on production grids. In *International Symposium on High Performance Distributed Computing(HPDC'09)*, June 2009.

[4] Franck Michel, Alban Gaignard, Johan Montagnat, David Godard, and Javier Rojas Balderrama. NeuroLOG client user guide. Technical report, I3S Laboratory, Sophia Antipolis, 2009.

[5] Johan Montagnat, Tristan Glatard, and Diane Lingrand. Data composition patterns in service-based workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'06)*, Paris, France, June 2006.

[6] Johan Montagnat, Benjamin Isnard, Tristan Glatard, Mireille Blay Fornarino, and Ketan Maheshwari. A data-driven workflow language for grids based on array programming principles. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'09)*, Portland, USA, November 2009.

A *j*GASW Java project

The *j*GASW Java project is organized in six modules: data binding, core, common, grid, UI, util and client API. The data binding module implements the process of mapping from the XML schema to Java objects. The core module implements the server-side business logic including the description interpretation and the relocation of resources. The common module declares the global properties. The grid module implements the remote relocation and grid communication. The UI module implements the interface with the client using graphical components. The util module implements the additional functionality of all operations used by the other modules. Finally, the client API module is used to invoke dynamically all deployed services. The code is available under CeCILL-B licence at `https://nyx.unice.fr/svn/jgasw/`

B Configuration properties

All *j*GASW configuration is set using 'properties'. The default values are defined in the `commons` module. Those values may be overriden using a jgasw.properties file located at `.jasw` directory on the user's home using java properties format `<property name> = <property value>`. System environment variables are used on runtime if these are not overridden using a *j*GASW property defined on the properties file, in that case the environment variable is ignored.

Property name	Description
*General j*GASW *properties*	
`jgasw.core.file.description`	XML descriptor's name of the CLI application. Default value: `description.jgasw`
`jgasw.service.file.metro`	Name of the Metro configuration file. Default value: `sun-jaxws.xml`
`jgasw.service.file.tomcat`	Name of the Tomcat configuration file. Default value: `web.xml`
`jgasw.template.file.metro`	Velocity template used to create the Metro configuration file. Default value: `metro.vm`
`jgasw.template.file.tomcat`	Velocity template used to create the Tomcat configuration file. Default value: `tomcat.vm`
`jgasw.template.file.service`	Velocity template used to create the WS interface of *j*GASW. Default value: `template.vm`

continued on next page

Property name	Description
FILE_SYSTEM_PATH	Temporal directory used to host the *j*GASW artifacts. Default value: <java.io.tmpdir>/jgasw
TMP_WORKING_PATH	Server's temporal directory used to relocate a CLI application and the associated inputs. It is used in local executions. Default value: <java.io.tmpdir>/_jgasw
jgasw.ws.file.stdout	Name of the standard output file. Default value: std.out
jgasw.ws.file.stderr	Name of the standard error file. Default value: std.err
jgasw.service.flag.deploy	Option to enable auto-deploy on $CATALINA_HOME/webapps. Default value: false
jgasw.service.flag.compile	Option to enable the compilation after the code generation. Default value: true
jgasw.ws.uri.flag.translation	Option to translate FILE URIs in Web URLs. Default value: true
Path translation properties	
jgasw.ws.uri.string.protocol	Protocol name. Default value: http
jgasw.ws.uri.string.host	Server name. Default value: localhost
jgasw.ws.uri.path.public	URL path. Default value: ∼/<user name>
jgasw.ws.uri.number.port	Server' port number. Default value: 80
jgasw.ws.file.data	Host directory of resulting files. Default value: $HOME/public_html
Grid properties	
jgasw.grid.lb.number.interval	Logging and Bookkeeping status interval in seconds. Default value: 30

continued on next page

Property name	Description
jgasw.grid.config.path.globus	User's Globus home directory. Default value: $HOME/.globus
jgasw.grid.config.path.glite	User's gLite home directory. Default value: $HOME/.glite
X509_USER_CERT	User's certificate from which the proxy credential is created. Default value: $GLOBUS_HOME/usercert.pem
X509_USER_KEY	User's private key from which the proxy credential is created. Default value: $GLOBUS_HOME/userkey.pem
PKCS12_USER_CERT	File name representing the usercert .p12 file. Default value: $GLOBUS_HOME/usercert.pk12
PKCS12_USER_KEY_PASSWORD	The password for unlocking the PKCS12 user certificate. Default value: Not set.
VOMSES_LOCATION	Directory where voms specification files are located. Default value: $GLITE_HOME/vomses
VOMSDIR	Directory where voms certificates are located. Default value: Not set.
CADIR	Directory where CA certificates are located. Default value: Not set.
X509_USER_PROXY	User's proxy certificate file path. Default value: <java.io.tmpdir>/x509up_u_<user name>

C Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://modalis.cnrs.fr/jgasw/xml/resource"
    xmlns:tns="http://modalis.cnrs.fr/jgasw/xml/resource"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="bundle" type="tns:bundleType"/>
    <xs:attribute name="category">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="application" />
                <xs:enumeration value="library" />
                <xs:enumeration value="command" />
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="osname">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="aix" />
                <xs:enumeration value="win32" />
```

```
                <xs:enumeration value="bsd" />
                <xs:enumeration value="os2" />
                <xs:enumeration value="solaris" />
                <xs:enumeration value="hpux" />
                <xs:enumeration value="irix" />
                <xs:enumeration value="linux" />
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="name">
        <xs:simpleType>
            <xs:restriction base="xs:string" />
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="io">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="in" />
                <xs:enumeration value="out" />
                <xs:enumeration value="nio" />
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="type">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="int"/>
                <xs:enumeration value="string"/>
                <xs:enumeration value="boolean"/>
                <xs:enumeration value="double"/>
                <xs:enumeration value="URI"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="brand">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="regular" />
                <xs:enumeration value="directory" />
                <xs:enumeration value="expand" />
                <xs:enumeration value="replace" />
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="regex">
        <xs:simpleType>
            <xs:restriction base="xs:boolean"/>
        </xs:simpleType>
    </xs:attribute>

    <xs:attribute name="implicit">
        <xs:simpleType>
            <xs:restriction base="xs:boolean"/>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="separator" >
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:length value="1"/>
```

```
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="initDelimiter">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:length value="1"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="endDelimiter">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:length value="1"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:complexType name="bundleType">
        <xs:complexContent>
            <xs:extension base="tns:resourceType">
                <xs:sequence>
                    <xs:element name="arguments"
                                type="tns:argumentsType"
                                minOccurs="0"/>
                    <xs:element name="dependencies"
                                type="tns:dependenciesType"
                                minOccurs="0"/>
                    <xs:element name="configuration"
                                type="tns:configurationType"
                                minOccurs="0"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="resourceType">
        <xs:sequence>
            <xs:element name="target" type="xs:string"/>
            <xs:element name="version" type="xs:string"/>
            <xs:element name="symbolicName" type="xs:string"/>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="localization" type="xs:anyURI"/>
            <xs:element name="organization" type="xs:string" minOccurs="0"/>
            <xs:element name="copyright" type="xs:string" minOccurs="0"/>
            <xs:element name="reference" type="xs:string" minOccurs="0"/>
            <xs:element name="contactAddress" type="xs:string" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute ref="tns:category" use="required" />
        <xs:attribute ref="tns:osname" default="linux" />
    </xs:complexType>
    <xs:complexType name="nestingType">
        <xs:sequence>
            <xs:element name="dimension" type="xs:int"/>
        </xs:sequence>
        <xs:attribute ref="tns:separator" default=","/>
        <xs:attribute ref="tns:initDelimiter" default=" "/>
        <xs:attribute ref="tns:endDelimiter" default=" "/>
    </xs:complexType>
    <xs:complexType name="configurationType">
        <xs:sequence>
```

```
            <xs:element name="variable"
                        type="tns:variableType" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="variableType">
        <xs:sequence>
            <xs:element name="content" type="xs:string" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute ref="tns:name" use="required"/>
    </xs:complexType>
    <xs:complexType name="valueType">
        <xs:sequence>
            <xs:element name="value" type="xs:string" minOccurs="0"/>
            <xs:element name="extensions" type="xs:string" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute ref="tns:regex" default="false"/>
        <xs:attribute ref="tns:brand" default="regular"/>
    </xs:complexType>
    <xs:complexType name="argumentsType">
        <xs:sequence>
            <xs:element name="argument"
                        type="tns:argumentType"
                        minOccurs="1"
                        maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="argumentType">
        <xs:sequence>
            <xs:element name="label" type="xs:string" />
            <xs:element name="option" type="xs:string" minOccurs="0" />
            <xs:element name="value" type="tns:valueType"/>
            <xs:element name="space" type="xs:boolean" minOccurs="0" />
            <xs:element name="nesting" type="tns:nestingType"/>
        </xs:sequence>
        <xs:attribute ref="tns:io"  use="required"/>
        <xs:attribute ref="tns:type" default ="URI"/>
        <xs:attribute ref="tns:implicit" default ="false"/>
    </xs:complexType>
    <xs:complexType name="dependenciesType">
        <xs:sequence>
            <xs:element name="dependency"
                        type="tns:resourceType"
                        minOccurs="1"
                        maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

D Descriptor example

```
<?xml version="1.0" encoding="UTF-8"?>
<res:bundle res:category="application"
            res:osname="linux"
            xmlns:res="http://modalis.cnrs.fr/jgasw/xml/resource">
    <res:target>AimsLinearComb</res:target>
    <res:version>1.0.0</res:version>
    <res:symbolicName>AimsLinearComb</res:symbolicName>
```

```
        <res:description/>
        <res:localization>/opt/jgasw/alc</res:localization>
10      <res:organization/>
        <res:copyright>CeCILL-B</res:copyright>
        <res:reference/>
13      <res:contactAddress/>
        <res:arguments>
            <res:argument res:io="in" res:type="URI" res:implicit="false">
16              <res:label>i1</res:label>
                <res:option>-i</res:option>
                <res:value res:regex="false" res:brand="replace">
19                  <res:value/>
                    <res:extensions>hdr</res:extensions>
                </res:value>
22              <res:space>true</res:space>
                <res:nesting res:separator=","
                             res:initDelimiter=""
25                           res:endDelimiter="">
                    <res:dimension>0</res:dimension>
                </res:nesting>
28          </res:argument>
            <res:argument res:io="in" res:type="double" res:implicit="false">
                <res:label>num1</res:label>
31              <res:option>-a</res:option>
                <res:value res:regex="false" res:brand="regular">
                    <res:value/>
34                  <res:extensions/>
                </res:value>
                <res:space>true</res:space>
37              <res:nesting res:separator=","
                             res:initDelimiter=""
                             res:endDelimiter="">
40                  <res:dimension>0</res:dimension>
                </res:nesting>
            </res:argument>
43          <res:argument res:io="in" res:type="double" res:implicit="false">
                <res:label>den1</res:label>
                <res:option>-b</res:option>
46              <res:value res:regex="false" res:brand="regular">
                    <res:value/>
                    <res:extensions/>
49              </res:value>
                <res:space>true</res:space>
                <res:nesting res:separator=","
52                           res:initDelimiter=""
                             res:endDelimiter="">
                    <res:dimension>0</res:dimension>
55              </res:nesting>
            </res:argument>
            <res:argument res:io="in" res:type="URI" res:implicit="false">
58              <res:label>i2</res:label>
                <res:option>-j</res:option>
                <res:value res:regex="false" res:brand="replace">
61                  <res:value/>
                    <res:extensions>hdr</res:extensions>
                </res:value>
64              <res:space>true</res:space>
                <res:nesting res:separator=","
                             res:initDelimiter=""
```

```
                    res:endDelimiter="">
            <res:dimension>0</res:dimension>
        </res:nesting>
    </res:argument>
    <res:argument res:io="in" res:type="double" res:implicit="false">
        <res:label>num2</res:label>
        <res:option>-c</res:option>
        <res:value res:regex="false" res:brand="regular">
            <res:value/>
            <res:extensions/>
        </res:value>
        <res:space>true</res:space>
        <res:nesting res:separator=","
                    res:initDelimiter=""
                    res:endDelimiter="">
            <res:dimension>0</res:dimension>
        </res:nesting>
    </res:argument>
    <res:argument res:io="in" res:type="double" res:implicit="false">
        <res:label>den2</res:label>
        <res:option>-d</res:option>
        <res:value res:regex="false" res:brand="regular">
            <res:value/>
            <res:extensions/>
        </res:value>
        <res:space>true</res:space>
        <res:nesting res:separator=","
                    res:initDelimiter=""
                    res:endDelimiter="">
            <res:dimension>0</res:dimension>
        </res:nesting>
    </res:argument>
    <res:argument res:io="in" res:type="double" res:implicit="false">
        <res:label>cst</res:label>
        <res:option>-e</res:option>
        <res:value res:regex="false" res:brand="regular">
            <res:value/>
            <res:extensions/>
        </res:value>
        <res:space>true</res:space>
        <res:nesting res:separator=","
                    res:initDelimiter=""
                    res:endDelimiter="">
            <res:dimension>0</res:dimension>
        </res:nesting>
    </res:argument>
    <res:argument res:io="out" res:type="URI" res:implicit="false">
        <res:label>fileout</res:label>
        <res:option>-o</res:option>
        <res:value res:regex="false" res:brand="replace">
            <res:value>alcresult.img</res:value>
            <res:extensions>hdr</res:extensions>
        </res:value>
        <res:space>true</res:space>
        <res:nesting res:separator=","
                    res:initDelimiter=""
                    res:endDelimiter="">
            <res:dimension>0</res:dimension>
        </res:nesting>
```

```
            </res:argument>
127         <res:argument res:io="out" res:type="URI" res:implicit="true">
                <res:label>minf</res:label>
                <res:option/>
130             <res:value res:regex="true" res:brand="regular">
                    <res:value>.*minf</res:value>
                    <res:extensions/>
133             </res:value>
                <res:space>true</res:space>
                <res:nesting res:separator=","
136                          res:initDelimiter=""
                             res:endDelimiter="">
                    <res:dimension>0</res:dimension>
139             </res:nesting>
            </res:argument>
        </res:arguments>
142     <res:dependencies>
            <res:dependency res:category="library" res:osname="linux">
                <res:target>libaimsalgo.so.3.0</res:target>
145             <res:version>1.0.0</res:version>
                <res:symbolicName>libaimsalgo.so.3.0</res:symbolicName>
                <res:description/>
148             <res:localization>/opt/jgasw/alc</res:localization>
                <res:organization/>
                <res:copyright>CeCILL-B</res:copyright>
151             <res:reference/>
                <res:contactAddress/>
            </res:dependency>
154         <res:dependency res:category="library" res:osname="linux">
                <res:target>libaimsalgopub.so.3.0</res:target>
                <res:version>1.0.0</res:version>
157             <res:symbolicName>libaimsalgopub.so.3.0</res:symbolicName>
                <res:description/>
                <res:localization>/opt/jgasw/alc</res:localization>
160             <res:organization/>
                <res:copyright>CeCILL-B</res:copyright>
                <res:reference/>
163             <res:contactAddress/>
            </res:dependency>
            <res:dependency res:category="library" res:osname="linux">
166             <res:target>libaimsdata.so.3.0</res:target>
                <res:version>1.0.0</res:version>
                <res:symbolicName>libaimsdata.so.3.0</res:symbolicName>
169             <res:description/>
                <res:localization>/opt/jgasw/alc</res:localization>
                <res:organization/>
172             <res:copyright>CeCILL-B</res:copyright>
                <res:reference/>
                <res:contactAddress/>
175         </res:dependency>
            <res:dependency res:category="library" res:osname="linux">
                <res:target>libcartobase.so.3.0</res:target>
178             <res:version>1.0.0</res:version>
                <res:symbolicName>libcartobase.so.3.0</res:symbolicName>
                <res:description/>
181             <res:localization>/opt/jgasw/alc</res:localization>
                <res:organization/>
                <res:copyright>CeCILL-B</res:copyright>
184             <res:reference/>
```

```
            <res:contactAddress/>
        </res:dependency>
        <res:dependency res:category="library" res:osname="linux">
            <res:target>libcartodata.so.3.0</res:target>
            <res:version>1.0.0</res:version>
            <res:symbolicName>libcartodata.so.3.0</res:symbolicName>
            <res:description/>
            <res:localization>/opt/jgasw/alc</res:localization>
            <res:organization/>
            <res:copyright>CeCILL-B</res:copyright>
            <res:reference/>
            <res:contactAddress/>
        </res:dependency>
        <res:dependency res:category="library" res:osname="linux">
            <res:target>libgraph.so.3.0</res:target>
            <res:version>1.0.0</res:version>
            <res:symbolicName>libgraph.so.3.0</res:symbolicName>
            <res:description/>
            <res:localization>/opt/jgasw/alc</res:localization>
            <res:organization/>
            <res:copyright>CeCILL-B</res:copyright>
            <res:reference/>
            <res:contactAddress/>
        </res:dependency>
        <res:dependency res:category="library" res:osname="linux">
            <res:target>libsigc-2.0.so.0</res:target>
            <res:version>1.0.0</res:version>
            <res:symbolicName>libsigc-2.0.so.0</res:symbolicName>
            <res:description/>
            <res:localization>/opt/jgasw/alc</res:localization>
            <res:organization/>
            <res:copyright>CeCILL-B</res:copyright>
            <res:reference/>
            <res:contactAddress/>
        </res:dependency>
        <res:dependency res:category="library" res:osname="linux">
            <res:target>libstdc++.so.5</res:target>
            <res:version>1.0.0</res:version>
            <res:symbolicName>libstdc++.so.5</res:symbolicName>
            <res:description/>
            <res:localization>/opt/jgasw/alc</res:localization>
            <res:organization/>
            <res:copyright>CeCILL-B</res:copyright>
            <res:reference/>
            <res:contactAddress/>
        </res:dependency>
    </res:dependencies>
</res:bundle>
```