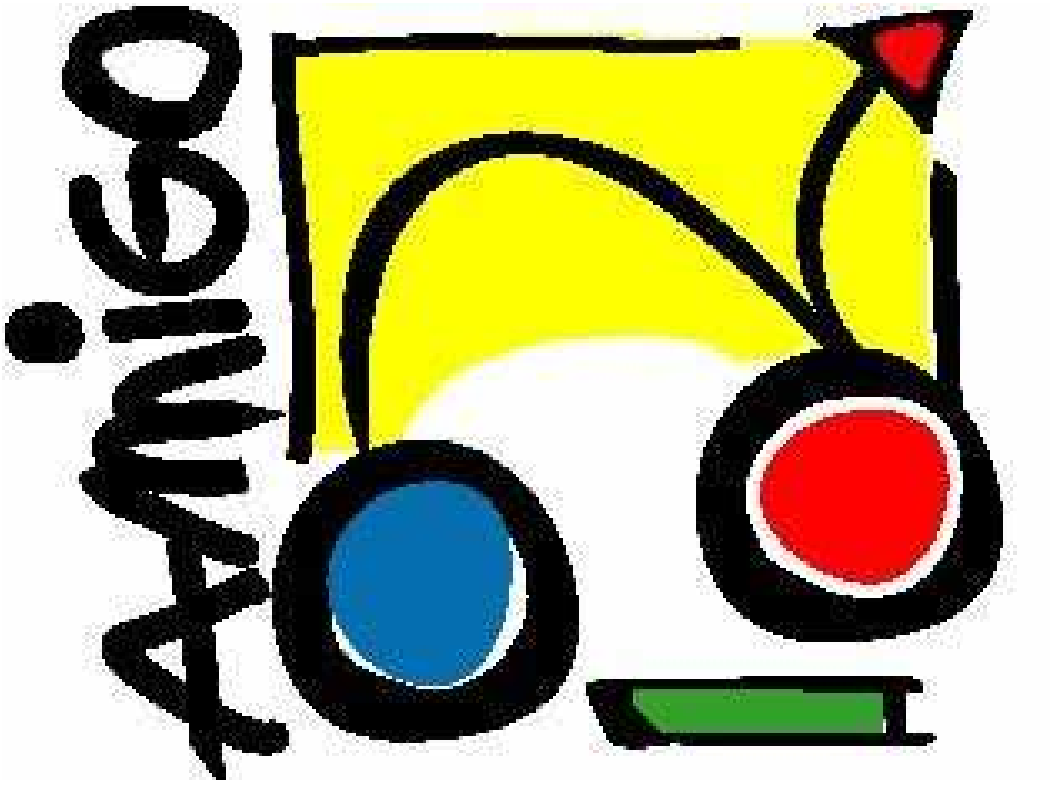

IST Amigo Project

SD-SDCAE
Software Developer's Guide

Public

Table of content

Table of illustrations

1.SD-SDCAE Architecture

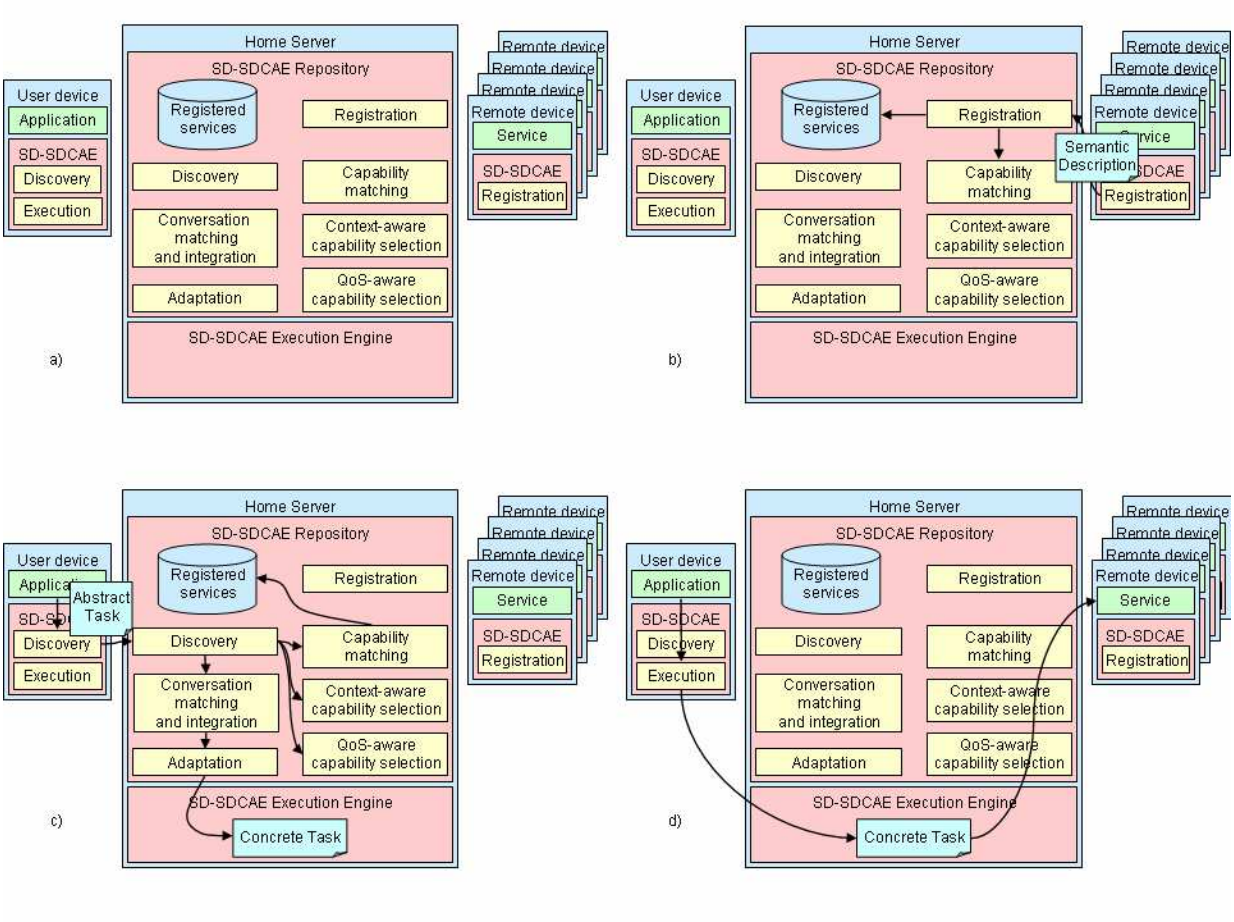

Figure 1-1: The SD-SDCAE high-level architecture

Figure 1-1 shows the high-level architecture of SD-SDCAE. Figure 1-1 a) shows the main components within SD-SDCAE. These include a user's device, which hosts the application which exploits SD-SDCAE, other remote devices, which host the services that the application wishes to use, and the Amigo home server, which host the SD-SDCAE Semantic Service Repository and Execution Engine. This configuration is flexible however, and can be reconfigured to suit the need of a deployment. The repository and execution can be hosted on independent machines, for example, should a particular deployment require it. Similarly, common services and/or application could be hosted on the Amigo home server.

Figure 1-1 b) to d) show the basic steps involved in registering services, dynamically composing and application using the available services, and executing the application, respectively.

Figure 1-1 b) shows a service registering itself with the Amigo Semantic Service Repository. The first step in this phase is that the service discovers the repository. The repository appears in the Amigo home network as a basic Amigo service, and can be discover using the standard Amigo basic service discovery mechanisms [Amigo-D3.5]. Next the service registers itself by passing its Amigo-S semantic service description to the repository.

Providing an Amigo-S semantic service description for a service provides several features and advantages to the Amigo application developer:

- It allows semantic service descriptions to be created for basic services, where the service provides a collection of atomic capabilities. This allows the service to be discovered via semantic matching at both the service and capability levels, increasing the service's availability and promoting its interoperability.
- Semantic service descriptions can provide semantic concepts for a whole service, for each capability and its inputs and outputs, and optionally its pre-conditions and effects.
- It allows semantic descriptions of conversation-based services, where the capabilities the service provides are described as a workflow of sub-capabilities. This allows the expression of data and control dependencies between the service’s capabilities.
- Complex workflows can be created by composing capabilities using a variety of control constructs, e.g. Sequence, Choice.
- Context parameters can be described for a service, both at the level of individual capabilities, and for the service as a whole. Then, the service can be incorporated by the context-aware discovery mechanisms~\cite{AmigoD34}, which enables an application to optimize service selection based on services’ current context, and frees the application developer from the need to actively search for the optimal service.
- The quality of service parameters a service supports can be described both at the individual capability level, and for the service as a whole. This makes the service available to applications which employ the priority-based quality of service selection model provided by the Quality of Service Aware Service Selection Tool (QASST)~\cite{AmigoD34}, which allows the Amigo middleware to serve as many requests as possible while satisfying the majority of Amigo users.
- Multiple groundings can be supported, enabling interoperability between different service technologies.

Figure 1-1 c) shows an application being dynamically composed, based on the services currently available in the repository. Again, the first step in this phase is that the application discovers the repository using standard Amigo basic service discovery. Then the application passes an abstract task, which describes the parts of the application that rely on external services available in the environment to run in an abstract way, to the repository. The repository then attempts to select suitable services based on matching the capabilities, context-aware parameters, and QoS parameters declared for the provided services, against those requested in the abstract task. If suitable services are found, the repository than attempts to compose and if necessary, adapt the available services to fit the workflow of capabilities required by the abstract task. Finally, if a successful composition is found, the concrete task - the version of the abstract task now instantiated with real service capabilities, is deployed on the home server.

Figure 1-1 d) shows an application executing the dynamically composed service, the concrete task, that realizes the abstract task in the previous phase. The concrete task appears as a regular Amigo service, and be discovery and executed as such. In fact, other external applications could use the newly-composed service, should it match their task description. When a capability of the task is executed, the SD-SDCAE Execution Engine automatically orchestrates the execution of each of the composite service capabilities that feature in the task capability's workflow, one or more remote devices.

Describing the service-based parts of application as a task offers the following features and advantages to the user:

- A task provides an abstract description of the required capabilities of an application. In doing so, we are not bound to any particular remote service in terms of the capabilities provided or the specific orchestration of these capabilities, increasing the availability and promoting interoperability of the potentially matching services.
- The required capabilities of a task can be identified by the semantic of the capability and of its inputs and outputs. This allows concrete details of the services' provided capabilities used in a composition to be reliably and automatically adapted to the needs of the required capabilities in the absence of an exact match.
- An application may invoke capabilities of varying complexity, from lightweight atomic calls to complex, interleaved conversations (i.e. service workflows), for one or many tasks from within the same application.
- The conversation (workflow) of a required capability of a task is reconstituted by weaving together the conversations (workflows) of the provided capabilities of the available services. This offers automatic and reliable service composition, while offering fine-grained control over the placement of capabilities in the task, and guaranteeing that the data and control dependencies of each of the provided capabilities are preserved.
- The resulting (composed) service is generated as an executable ActiveBPEL bundle and automatically deployed. Orchestration of the execution of the composed service is handled automatically by the ActiveBPEL execution engine.

2.Global architecture of sources

For a better reusability of the developed source code, we decide to divide the code into 2 main part: a part concerning the repository himself (repository structure, algorithms for Service Discovery, algorithms for Service Composition) – *SD-SDCAE Semantic Repository* – and another part concerning the data model – *SD-SDCAE Service Model*.

2.1. SD-SDCAE Service Model

This part of source code contains the data model and some extension as QOS, encoded type… but also some tools as a parser or automata model that will be use during composition.

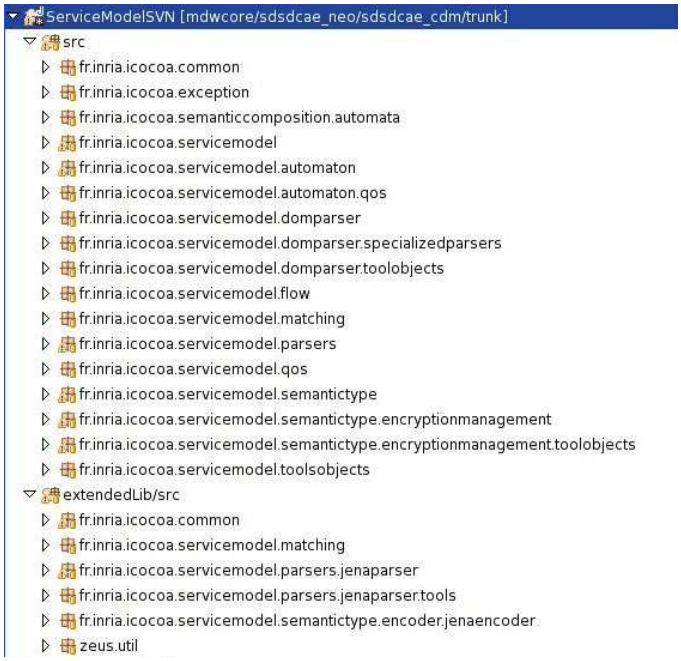

Figure 2-1 - Source architecture of SD-SDCAE Service Model

As you can see figure 2, the source code is divided into 2 folders. The common one (*./src/*) that contains the main part of code including service model, and another one (*./extendidLib/src/*) that contains the code that refer to *Jena.* This has been done to avoid the main part of code to require a huge number of libraries. By the way, the main part of service model code required only one external library (*Log4J*).

The source code is available on SVN track on gforge.inria.fr with URL http://yourname@scm.gforge.inria.fr/svn/amigo/mdwcore/sdsdcae_neo/sdsdcae_cdm

2.1.1. The base library

This “library” contains the data model use for all algorithms. If you use the given ant build file, the *jar* target generate the file *SD-SDCAE_ServiceModel_V*.*.jar*. The only external library required is *log4j-*.jar*.

This library is composed of data model (*fr.inria.icocoa.servicemodel.**), some tools that can be use by all components (*fr.inria.icocoa.common.**) and the package of exception (*fr.inria.icocoa.exception.**).

The last package (*fr.inria.icocoa.semanticcomposition.automata.**) contained into the library is here for convenience. Indeed, for composition algorithm, each services and capabilities required to be converted into an automaton. For easily access it, we decide to put it as a field of Service/Capability object. So the class Automaton (and so, the tools that permit to use it) required to be in the same library to avoid all compilation troubles.

2.1.2. The extended library

This extension of the library completes the basic library with all classes that required *Jena*[1]. Indeed, *Jena* (the ontology manager used) is an external library of functionalities. This library required to works a number of other external libraries. So as to use *Jena* you required the following list of jar files, which represents an over-cost of 35Mo:

- icu4j_3_4.jar
- arq.jar
- jena.jar
- relaxngDatatype.jar
- aterm-java-1.6.jar
- pellet.jar
- xsdlib.jar
- iri.jar
- jenatest.jar
- commons-logging-1.1.jar
- xercesImpl.jar
- arq-extra.jar

This extension contains the Amigo-S parser (*fr.inria.icocoa.servicemodel.parsers.jenaparser.**), the Semantic Matcher that use *Jena* to get some relation between concept (*fr.inria.icocoa.servicemodel.matching.**) and the classes that permit to get a tree of concept from a loaded ontology (*fr.inria.icocoa.servicemodel.semantictype.encoder.jenaencoder.**).

2.2. SD-SDCAE Service Repository

This set of package is the source code of the repository himself. It contains the code for the repository architecture and implementation of all high-level algorithms: service discovery, service composition, BPEL generation. It also contains some classes that implement some useful tools to help the developer to generate Amigo-S description file and to test the repository.

---

[1] http://jena.sourceforge.net

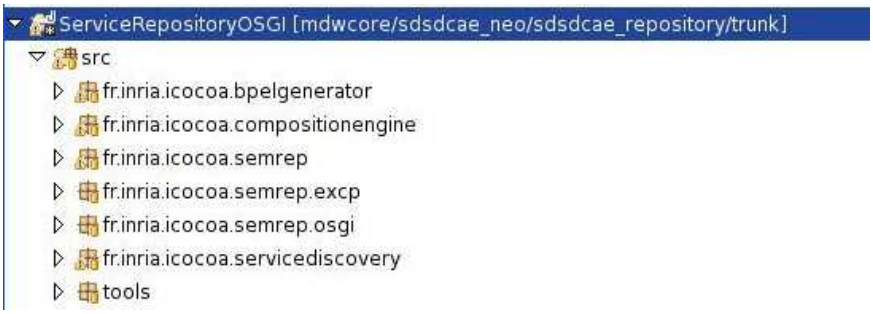

Figure 2-2 - Source architecture of SD-SDCAE Service Repository

As you can see on Figure 3, the service repository code is divided in many packages. The one named *fr.inria.icocoa.semrep.** contains all code for the global architecture of the repository (data storage, procedure to access data,…) including the specifics exceptions that have been defined (*fr.inria.icocoa.semrep.excp.**), and all the material required to convert this repository into an OSGi bundle (*fr.inria.icocoa.semrep.osgi.**). This possibility of conversion explains the fact that most of the interface return and required parameters with simple type. Indeed, troubles occurred during networks connections when we try to use more complex type with OSGi.

Algorithm used during the process of service composition are the service discovery (*fr.inria.icocoa.servicediscovery.**), the service composition itself (*fr.inria.icocoa.compositionengine.**) and the BPEL generator (*fr.inria.icocoa.bpelgenerator.**).

Some tools are available in package *tools*. The class *amigosgen* permit to generate an Amigo-S description skeleton from a WSDL description of a service. All other tools are more useful during debugging and testing of the repository.

The source code is available on SVN track on gforge.inria.fr with URL http://yourname@scm.gforge.inria.fr/svn/amigo/mdwcore/sdsdcae_neo/sdsdcae_repository

3.Some details about implementations

This part of the document will focus on few details of class architecture.

3.1. Data Model

The service repository has been done to implements the algorithm of Service Composition. This algorithm works at Capability level.

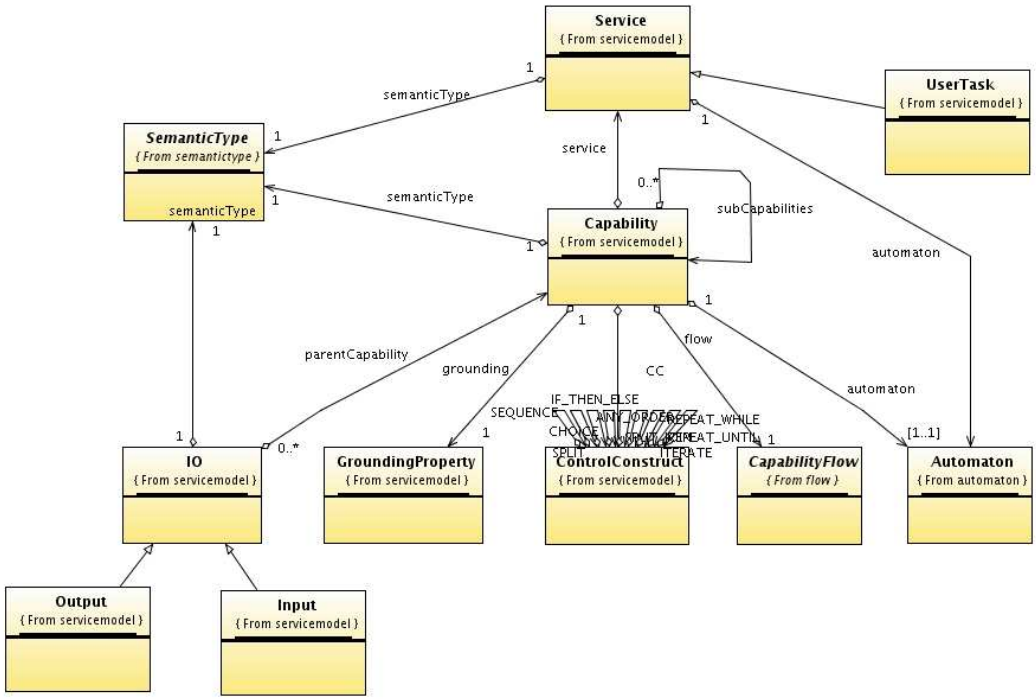

Figure 3-1 - Diagram of classes for the Service Model

Figure 4 shows that the data model is build around Capabilities. Those one can be Atomic or Composite (and so having some sub-capabilities).

A service object is a set of capabilities. Theses capabilities could represents some single operations (in that case, a capability is directly linked with a Web Service operation, so the grounding property is defined, so flow is atomic, as the automaton) ; or some composite ones. In that case, a flow describes the link between the composite capability and the set of capability that compose it.

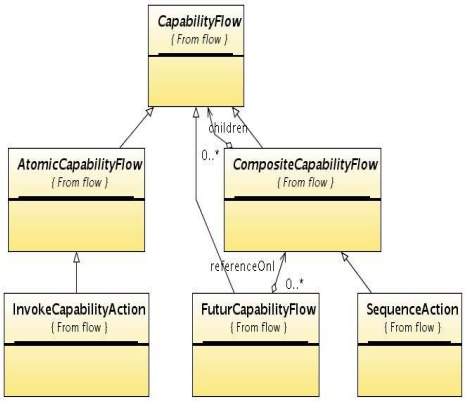

Figure 3-2 - Diagram of class focus on CapabilityFlow

The description of the flow follows the design pattern *Composite*. The class FuturCapabilityFlow is a tool class. Indeed, the flow of capability is done during the parsing process. Nevertheless, since a CompositeFlow can be compose of another composite flow, troubles can occurs due to order in the Amigo-S description file (a process A could required a process B, but the process B is unknown at the current moment). In that case, an instance of FuturCapabilityFlow is used to describe completely the flow. At the end of the process, all instance of FuturCapabilityFlow are replaced by the up-to-date CapabilityFlow instance, using the list of references of the FuturCapabilityFlow object.

3.2. Semantic Type

All algorithms are based on use of semantic Types. Those semantic types represents data type, but also the “action” of a capability, service… To improve the performance of algorithm, those semantic types can be encoded.

Figure 6 shows how classes are organized. In the Amigo-S description file, the semantic type is described by a String. This information is stored whatever coded you use during algorithm. Encoded type added a list of codes for the associated type.

In fact, the EncodedSemanticType class is associated with two another classes:

- A subclass of Code. Instance of this class will be created to be associated with EncodedSemanticType object to represent the semantic type.
- A subclass of Encoder. An instance of this Encoder will be dynamically loaded when ontology has to be encoded to works with the selected EncodedSemanticType class.

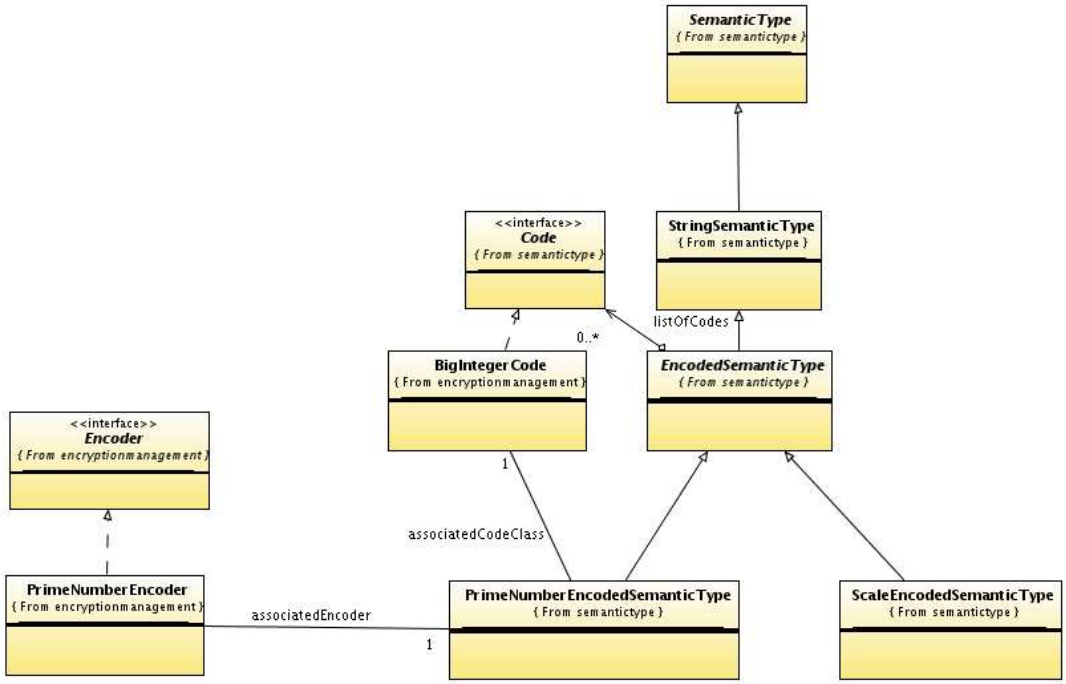

Figure 3-3 - Class diagram of Semantic Type

4.The configuration file

The all properties that can be set by the user are gathered into the configuration file. Here is an example of the file that path **must be** “./config/config.xml” where “.” represents the path from where the application has been started.

4.1. Global Organisation

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This file contains all extra data to permit that the cocoa middleware works well. Please make change with caution.  -->
<config>
   <!-- This part is about the configuration of the bpel server-->
   <bpel_configuration>
      <!-- The address where the server is running (nameOfComputer:port) -->
       <bpel_server_address>chico1:9080</bpel_server_address>
      <!-- This is the url of the Deployement web service for Bpel script -->
      <bpel_deployement_service>/active-bpel/services/DeployBPRService</bpel_deployement_service>
       <!-- This is the name of the operation call to deploy the bpel script.
          The corresponding web service is the value of "bpel_deployement_service" -->
      <bpel_deployement_operation>deployBpr</bpel_deployement_operation>
   </bpel_configuration>

   <!-- This part permit to configurate the different path
      BE AWARE : theses have to be absolut path or need to start with a ".". In this case, the "." will be change with the path where the application start !
   -->
   <path_configuration>
      <!-- The path of the folder where the ontology are stored -->
      <ontology_folder_path>./ontologies</ontology_folder_path>
      <!-- The path of the folder where the generated bpel files will be stored -->
      <bpel_folder_path>./bpel_generatedFiles</bpel_folder_path>
      <!-- The path of the folder where the generated repository will be stored -->
      <repository_folder_path>./repository</repository_folder_path>
      <!-- The path of the folder where the library are stored -->
      <lib_folder_path>./lib</lib_folder_path>
      <!-- The path of the folder where temp files will be stored -->
      <tmp_folder_path>./temp</tmp_folder_path>
      <!-- The path of the folder where the source code files for new operation type are stored -->
      <operation_folder_path>./OperationType</operation_folder_path>
      <!-- The path of the configuration path for log4j. You can use both "basic" or "xml-based" configuration file.
          WARNING : The software recognize xml configuration file only if its name ends with ".xml"  -->
      <log4j_configuration_file_path>./config/log4j.prop</log4j_configuration_file_path>
   </path_configuration>
```

```
  <!-- Set the name of the class that will be use as Service Parser (mandatory, no default value)-->
   <system_property key="serviceparser.classname"
value="fr.inria.icocoa.servicemodel.parsers.jenaparser.SDL2ServiceParser"/>
   <!-- Set the name of the class that will be use as Semantic Type implementation (default =
fr.inria.icocoa.servicemodel.semantictype.StringSemanticType) -->
   <system_property key="semantictype.classname"
value="fr.inria.icocoa.servicemodel.semantictype.PrimeNumberEncodedSemanticType"/>
   <!-- Set the number of primes generated for encode ontologies (default = 100000 ) -->
   <system_property key="primes.requirednumber" value="100000"/>
   <!-- Set the name of the class that will be use as Matcher implementation (default =
fr.inria.icocoa.servicemodel.matching.SyntacticMatcher) -->
   <system_property key="matcher.classname" value="fr.inria.icocoa.servicemodel.matching.WeakSemanticMatcher"/>
   <!-- Indicate if time measure need to be save and display (default = false) -->
   <system_property key="measurementtool.savetrace" value="true"/>
    <!-- Set the name of the class that will be use as GraphGeneration implementation (default =
fr.inria.icocoa.servicemodel.semantictype.encoder.jenaencoder.JenaOntologiesGraphGenerator) -->
   <system_property key="graphgenerator.classname"
value="fr.inria.icocoa.servicemodel.semantictype.encoder.jenaencoder.JenaOntologiesGraphGenerator"/>
   <!-- Set the name of the class that will be use as ServiceBag implementation (default =
fr.inria.icocoa.semrep.dataset.ServiceBagImplementation) -->
   <system_property key="servicebag.classname" value="fr.inria.icocoa.semrep.dataset.ServiceDAGImplementation"/>
</config>
```

Figure 4-1 - Configuration file content

This configuration file is divided into three parts. The first one address the configuration of the BPEL Engine. You can configure there the URL of the web service, the name of the server and so one…

The second part concerns the configuration of the path using by the application. Those paths must be global path (ie C:\myWindoswPath or /myUnixPath) but can be relative if it started with a ".". In that case, the path is relative, based on the path where the application is started from.

The third and last part is relative to system properties. Those system properties are used to manage the different features of the middleware.

4.2. System properties

serviceparser.classname: This property is used to select the parser main class that must be loaded. There is no default value, so if this property is missing, the application will quickly crash, with a message asking for this property.

semantictype.classname: This property is used to select the class that will be use to represent the semantic type of the concept. By default, StringSemantcType will be used. This class just stores the semantic type as String and permits no encryption of concept.

matcher.classname: This property is used to set the class that will be used as matcher between semantic type. Be careful that the matcher is compatible with the selected SemanticType object represent (cf property semantictype.classname) because no check is performed by the application. If

the property is not set, the default classes that will be use is SyntacticMatcher that only check the equality of the two Strings that represent both concept (method equals of String class).

graphgenerator.classname: This property is used to set the class that will be used as GraphGenerator. This graph generator is required to encode the semantic type. This property is used only when an "EncodedSelectedType" is used as SemanticType object. The default value is JenaOntologiesGraphGenerator. As the name indicates, this class uses Jena, so it required the extended library to works.

servicebag.classname: This property is used to set the class that will be used as Service Bag. This property is used only when using the Repository. The Service Bag instance will store all capabilities registered among the repository. Those capabilities can be ordered or not.

measurementtool.savetrace: This property indicate if we want that the software store all time measurement of the application into a file when the application ended. The measurement output file is "./out_measurement.txt" You can add some measure into the code by using MeasurementTool.displayTime(…)

primes.requirednumber: This property indicates the number of primes that has to be generated to encrypt the all ontologies.