Report on the implementation of the integrated ACGT environment and workflows

Project Number:	FP6-2005-IST-026996
Deliverable id:	D 9.2
Deliverable name:	Report on the implementation of the integrated ACGT environment and workflows
Date:	December 27, 2007

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	ACGT
Project Full Name:	Advancing Clinico-Genomic Clinical Trials on Cancer: Open Grid Services for improving Medical Knowledge Discovery
Document id:	D 9.2
Document name:	Report on the implementation of the integrated ACGT environment and workflows
Document type (PU, INT, RE)	INT
Version:	0.5 (Final)
Date:	27/12/2007
Editors: Organisation: Address:	Stelios Sfakianakis FORTH-ICS Foundation for Research and Technology-Hellas (FORTH) Institute of Computer Science N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece

Document type PU = public, INT = internal, RE = restricted

ABSTRACT:

This document aims to provide information on the initial implementation of the integrated ACGT environment and the service composition environment which realizes and validates it. Description of the integration requirements based on the initial reference implementation of the services is given. Furthermore the prototype implementation of a Workflow Editing and Enacting environment is described and experience on this effort is presented. Finally, areas that need improvement and work directions that should be the focus of future efforts are explored.

KEYWORD LIST: workflow; service composition; rich internet applications; semantic interoperability; integration

MODIFICATION CONTROL			
Version	Date	Status	Author
0.1	20/11/2007	Draft	Stelios Sfakianakis
0.3	6/12/2007	Draft	Stelios Sfakianakis, Lefteris Koumakis, Johan Karlsson
0.4	19/12/2007	Pre-Final	Stelios Sfakianakis, Giorgos Zacharioudakis
0.5	27/12/2007	Final	Stelios Sfakianakis

List of Contributors

- Stelios Sfakianakis, FORTH-ICS
- Lefteris Koumakis, FORTH-ICS
- Giorgos Zacharioudakis, FORTH-ICS
- Johan Karlsson, UMA

Contents

Table of Figures

Acronyms

AJAX	Asynchronous Javascript and XML
API	Application Programming Interface
ASP	Active Server Pages
BPEL	Business Process Executable Language
CA	Certification Authority
CGI	Common Gateway Interface
CSS	Cascading Style Sheets
DMS	Data Management Service
DOM	Document Object Model
GridFTP	Grid File Transfer Protocol
GSI	Grid Security Infrastructure
GT4	Globus Toolkit 4
GUI	Graphical User Interface
HTML	HyperText Markup Language
JSP	Java Server Pages
ODE	Orchestration Director Engine
OWL	Web Ontology Language
PHP	PHP: Hypertext Preprocessor
RDF	Resource Description Framework
RIA	Rich Internet Application
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
SPARQL	SPARQL Protocol and RDF Query Language
SVG	Scalable Vector Graphics
URI	Uniform Resource Identifier
VO	Virtual Organization
WEEP	Workflow Enactment Engine Project
WFMC	Work Flow Management Coalition
WS	Web Services
WS-I	Web Services Interoperability
WSDL	Web Service Description Language
WSRF	Web Services Resource Framework
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Executive Summary

This document aims to provide information on the initial implementation of the integrated ACGT environment and the service composition environment which realizes and validates it. Description of the integration requirements based on the initial reference implementation of the services is given. Furthermore the prototype implementation of a Workflow Editing and Enacting environment is described and experience on this effort is presented. Finally, areas that need improvement and work directions that should be the focus of future efforts are explored.

1 Introduction

In the present document we reflect on the first implementation of the ACGT integrated environment. The definition of such an environment encompasses the selection of interoperable standards, the implementation of the tools and services in conformance to these standards and other quality criteria such as security and performance, and finally the specification and implementation of the necessary infrastructure for the synthesis of the available “islands” of functionality to form higher level and more complex added value reusable software entities.

In Section 2 we describe the experiences with respect to the integration and interoperability of services in ACGT. A reference implementation of one of the ACGT services and the efforts for integrating it with other services in the context of a workflow have been used to gain valuable insights to guide the implementation of future ACGT services. In Section 3 the ACGT Workflow Environment is analyzed and its various aspects (Workflow Editor, Workflow Enactor, Metadata) are studied and scrutinized in the light of the prototype implementations. Finally in Section 4 we conclude by providing information about open issues and further enhancements are proposed.

2 Integration in ACGT

The general guidelines for achieving integration and interoperability in ACGT have been described in D9.2. Here we give some additional insight into the implementation of services and tools based on the prototype version of the workflow environment and the service reference implementation.

2.1 The ACGT Architecture

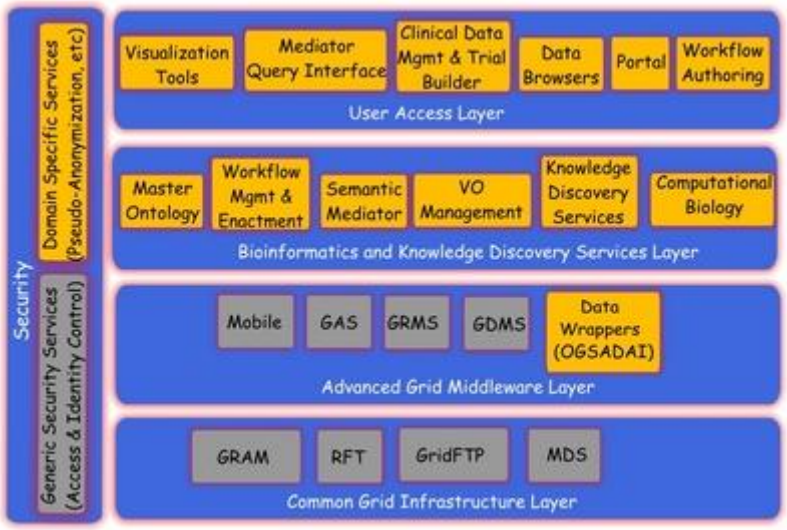

Figure 2-1 The ACGT Layered Architecture

The ACGT platform is designed according to the following technologies and standards:

- The Service Oriented Architecture (Web Services)
- The Grid
- The Semantic Web

These technologies coexist synergistically and complementary to each other in order to support the interoperability and integration requirements of the ACGT end user scenarios and applications. In essence, the Grid provides the computational and data storage infrastructure, the general security framework, the virtual organization abstraction and relevant user management mechanisms etc. The machine to machine communication is performed via XML programmatic interfaces over web transport protocols, which are commonly referred as Web Services interfaces. Finally the Semantic Web adds the knowledge representation mechanisms through the means of OWL ontologies, the implementation-neutral query facilities with the SPARQL “universal” query language and the associated query interfaces, etc.

2.2 Service Reference Implementation

As part of the initial implementation plan a reference implementation of a service has been started and the process of building and deploying such a service in conformance to the ACGT Grid infrastructure, and security architecture had been documented in the project's wiki pages (http://decenturl.com/wiki.healthgrid/refimpl). The provision of a new service mostly involves the implementation of the service's functionality in adherence to the ACGT guidelines but in some cases it may require its deployment as well. For the latter case of the service deployment, the first thing that a service provider should do in order to become part of the ACGT Grid is to get a server certificate signed by the ACGT Certification Authority (CA). This will approve the authenticity of its interactions and a proof of its unique identity[1]. After that the necessary software components should be installed and configured appropriately. These components include the Globus Toolkit 4 (GT4) and the Apache Tomcat container and when these are installed the new server site is part of the ACGT Grid.

The implementation of the service requires for it to be implemented as a Web Service and its core logic to be wrapped in a way that makes possible the reuse of the existing ACGT infrastructure. The Globus Toolkit incorporates facilities and tools like 'wsdl2java' to ease the implementation but to large extent this depends on the specific requirements of the service. For example:

- There may be cases where the service needs to know the identity of the final end-user that calls it
- The data input need not be given directly to the service; instead a *reference* to the data may be passed and the service should download the ultimate data form the ACGT data grid
- The service needs to further invoke additional services in the name of the user. This implies the support of the delegation of the user's credentials to the other services.

These requirements and more other are supported by the Grid middleware and the Gridge toolkit. In addition to all of the above the service need to be *published* in order to be later on discovered and selected among the other services based on its functionality, performance, stability, etc. The need for metadata annotations of services is therefore quite important and the preparation of these service descriptors and their validation by the ACGT quality committees are the final steps in the process of creating a new ACGT service.

2.2.1 Service implementation best practices

In the process of building the reference implementation and testing it with the workflow environment a number of requirements and "best practices" for the implementation of services have been revealed:

- **Use of the so called Document/Literal style in WSDL**. There are two main styles used in the service description documents: the RPC/Encoded and the

[1] The server certificate also covers the "non-repudiation" property, i.e. so that the service can not deny its involvement in a certain interaction and claim forgery of its identity.

Document/Literal[2]. The first one is considered deprecated by the Web Services Interoperability Organization (WS-I, http://www.ws-i.org/) since it leads to a more object oriented (or remote procedure oriented) view of the XML communication. In the Document/Literal case the exchanged messages do not follow a pre-established set of encoding rules like the SOAP Encoding used in the RPC/Encoded case and they can be validated by XML Schemas, parsed with common XML parsers, transformed with XSLT stylesheets, etc. It is therefore the preferred and recommended way to build Web Services. A variant of this style is the Wrapped Document/Literal convention popularized by Microsoft .NET that creates more programmer friendly view of the services interface while maintaining the virtues of the Document/Literal model in the wire XML messages. In conclusion the Document/Literal with the "Wrapped" convention is the most interoperable and programmer friendly way for describing and implementing Web Services and it is supported by the *java2wsdl* tool used in Globus Toolkit 4 with the "`--style WRAPPED`" and "`--use LITERAL`" options.

- **Wrapped Arrays**. Even with the Document/Literal style specified in a WSDL, it seems that the *wsdl2java* tool of Java Axis 1.x that is included in Globus Toolkit 4 does not produce a serialization of arrays (sequences of the same element type) in conformance to the WS-I Basic Profile [WSBP]. Entering the "`-w`" (or "`--wrapArrays`") undocumented option in the command line will ensure the correct behaviour.

- **"Call-by-reference**". Being a text format, XML is well known for its unfriendliness for transferring binary data. There are a couple of solutions for this ranging from encoding the binary data in hexadecimal or, most often, in Base64 text format to using "attachments" in the SOAP messages. The architecture of ACGT is based on the Grid technologies and there's an alternative way which is to transmit *references to data* as part of the Web Services interaction while the data itself can be transferred through "out of band" channels, e.g. by the means of GridFTP. This approach offers the advantage of "quicker" XML interactions, easier and more performant service composition since there's no need to "get" (download) a huge binary data set in order to "give" (upload) it to another service, identity of the data so that they can associated with metadata through their references, etc. The only foreseen drawback of this approach is that the services need a better metadata schema for describing their functionality so that, for example, we can register the fact that a service accepts a reference to a microarray expression data file as its parameter.

- **Asynchronous interfaces**. Some services may need a lot of time to accomplish their requests. An example of such a service is GridR [D6.2] where the time required for an invocation to it is not known a priori since it depends on the input script and this time can be quite long. In such cases, the request – response model of interaction is not the best one for practical reasons: the network connection could time out, the resources used (e.g. open "file" descriptors) accumulate and this can be the source of "denial of service" problems, etc. It is therefore better for the service to expose an *asynchronous* interface instead of the synchronous (request – response) one. In the asynchronous interface instead of a single long-lasting operation we have multiple short operations. The first one ("submit") is used for the initial invocation and the

[2] A good description of these styles and some less popular ones is at http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/

submission of the parameters and it returns some unique reference id (a token) to this specific invocation. There's also a final operation that based on the reference id it returns the results of the requested work if it has finished and possibly others that check the status of the work submitted or possibly cancel it[3]. From the Grid middleware side the Web Services Resource Framework (WSRF) has been proposed as a more or less standard way to cover also these cases.

The above guidelines constitute more or less "strong" recommendations for interoperability and performance and will be further explored in the future development of the ACGT services and tools.

[3] The interface described corresponds to the *polling* or *pull* model where the client originates each communication. The *push* model could be also used where the server, actively, notifies the client when the work/job has finished execution.

3 The ACGT Workflow Environment

3.1 Introduction

The Workflow Management Coalition (WFMC, http://www.wfmc.org/) defines a workflow as "The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules". In other words a workflow consists of all the steps and the orchestration of a set of activities that should be executed in order to deliver an output or achieve a larger and sophisticated goal. In essence a workflow can be abstracted as a composite service, i.e. a service that is composed by other services that are orchestrated in order to perform some higher level functionality.

The term “ACGT Workflow Environment” collectively describes the end user interfaces and the grid infrastructure for the design, archiving, execution, and management in general of the ACGT scientific workflows. The aim of this environment is to assist the users in their scientific research by supporting the composition of different data access and knowledge extraction and analytical services into complex workflows. This way the users can extend and enrich the functionality of the ACGT system by reusing existing ACGT compliant services and producing “added value” composite services. This reuse and composition of services is in some sense a programming task where the user actually writes a “program” to realize a scenario or to test a scientific hypothesis.

The major objectives of the ACGT workflow environment are:

- The definition (*design*) of new scientific workflows based on the user’s experimental scenarios or exploratory data analysis tasks.

- The management of the workflows as *reusable* functional components. The workflows are functional entities that can be reused either in the same way or with slight modifications (“repurposed”) and therefore they should be persisted and easily located and retrieved.

- The provision and maintenance of the necessary *metadata* descriptions that will enable not only the discovery of relevant workflows in a custom task but also to support the extraction of knowledge about the way services can be combined and operate in a collaborative manner.

- The execution (“*enactment*”) of available workflows taking into account knowledge about the ACGT platform and its Grid fabric. In particular the execution of the workflows in the Grid and the availability of services like the Grid Data Management Service (DMS) offer additional degrees of freedom both in terms of performance and in terms of functionality.

- The bookkeeping of the results of the workflows execution and the maintenance of the back links to the initial inputs. The so called “*provenance*” information should be maintained so that results can be reproduced or better documented as part of the scientific process that the workflow participated in.

3.2 Design of the ACGT Workflow Environment

Following a component based methodology we have identified the following software entities for the realization of the ACGT Workflow Environment:

- The *Workflow Editor* is the end user tool whose major functionality is the definition of new workflows
- The *Workflow Engine* ("Enactor") is the network service that is responsible for the enactment of the stored workflows

These two components are able to communicate through a common *workflow description language* that is able to fully describe the composition of services into the workflow.

3.2.1 Rationale for having two separate software components

We have made the decision that we should separate the two phases of the workflow authoring and the enactment. Well known workflow editors such as the Taverna Workbench or Triana do not make this distinction[4] and they incorporate both functionalities in a single application. There are of course advantages as well as disadvantages to this line of thought. The advantages for having the workflow editor supporting also the enactment is that the user has more control of the workflow execution, it's easier to use the local resources (e.g. a file in his/her desktop machine), etc. On the other hand the separation of the execution from the authoring phase offers the following advantages:

- The workflow can be executed in a remote machine, which hopefully is more powerful, or even in a cluster of machines in the Grid, depending of course on the implementation of the Workflow Enactor. Nevertheless, in every case, the user should not be worried about the specific implementation details of the workflow execution mechanisms and there's no need to install additional software on his/her workstation.
- There is no burden imposed on the user's local machine since the majority of computation and data transfer of the intermediate results are taken place in the Grid where the services are run.

If we may use a metaphor we could say that having two separate components, one for the authoring and another for the enactment of the workflows, is reminiscent of the way a compiled programming language (usually) works: The user writes and "compiles" the program and after that, in a later step, he/she can execute ("run") it. On the other hand the approach of having a common environment for authoring and executing the workflows is similar to the "Read Evaluate Print Loop" (REPL, http://en.wikipedia.org/wiki/REPL) interactive environment of some interpreted languages (e.g. Lisp or Scheme). Without delving into religious arguments about which of the two is better, it is generally agreed that the compiled version offers better performance and easier deployment whereas the interpreted environment minimizes the "edit-run-debug" cycle and offers better interactivity because the outputs of each processing step are readily available to the user.

[4] This is the case for the versions available at the time of the writing of this report. At least for Taverna there are plans to move towards to the direction that ACGT is heading in the next major release (2.0) of the software.

- As a consequence of the above, the user is allowed to leave or even shut down his/her machine should the execution of a workflow take too long. The outputs of the workflow can be stored in temporary locations in a user specific area and later on retrieved by the user.

- The workflows by default will be saved outside the user's premises so it's easier to enable the sharing and reuse of the workflows. This of course requires the user's consent and all the security constraints should be satisfied.

- The evolution of the workflow engine and the installation of a new version is easier because the upgrade can be done centrally.

There are of course some disadvantages:

- There is an additional step before the workflow is able to be executed: the workflow should be deployed in the Workflow Enactor. This means that the workflow can not be run before it is validated, transformed into the appropriate format, and stored in the enactor's database. Of course much of this additional step can be done in the background and shunned away by friendly user interfaces.

- The interactivity is severely affected because the workflow is run outside the user's control. For example additional mechanisms should be implemented in order to provide feedback to the user about the status of the execution or to enable the user to stop the execution at some point. This kind of functionality can of course be implemented and realized by additional programmatic interfaces to the enactor's core but even in this case the network latency could affect negatively the user's experience.

The decision taken in ACGT was to uncouple the two phases - the "think" and the "enact" phase - and to provide the tools and the mechanisms that properly handle and serve each one in the best possible way. This decision is also taken based on the project's technological directions, i.e. the grid and service oriented foundations seem to be more inline to the definition of a more component based architecture.

3.3 Workflow Editor

The ACGT Workflow Editor is a graphical tool that allows a user to combine different ACGT services into complex workflows. This tool is accessible through the ACGT Portal and therefore has a web based graphical user interface. It supports the searching and the browsing of the available services and data sources and their composition through some intuitive and user friendly interface. The workflows created can be stored in a user's specific area and later retrieved and edited so new versions of them can be produced. The publication and sharing of the workflows are also supported so that the user community can exchange information and users benefit from each other's research. Finally the workflow editor supports the execution of the workflows and the monitoring of their enactment status.

3.3.1 Functional Requirements

The following functional requirements have been identified:

- Present a navigable presentation of the available services and workflows. Faceted classification is a preferred way to support the efficient browsing of a large number of

entities and the utilization of the service and data type ontologies will leverage this kind of functionality.

- Support the graphical composition and linking of services based on the descriptions of their inputs and outputs. The Workflow Editor supports the syntactic analysis of services and provides feedback to the user for the compatibility of two services when connected together. The possibility of runtime errors should therefore be largely decreased when the workflow editing environment itself discovers syntactic inconsistencies and notifies the user about them through visual indications.
- Provide for the semantic validation of the workflows assuming that the participating services have been annotated semantically. This kind of functionality will try to eliminate the "logic errors" and support the creation of workflows that model useful and meaningful scenarios.
- Support the persistence and retrieval of workflows. Workflows are stored centrally (in the ACGT Data Grid) and can be designated either as private, which means that only the creator has access to them, or public, which means that all the ACGT users have the rights to view and edit them. The publication facility leverages the reuse or "repurpose" of the workflows by other users to accomplish a similar but slightly different scientific problem, e.g. by changing a parameter's value or a data input.
- Versioning support is required, especially in the cases where a workflow is made public. The user during the editing of a private workflow will be given the choice to store the altered workflow either as a new version of it or in the old version (overwriting the existing copy). When the edited workflow is a public one then the user is forced to store it under a different name so that the previous public version remains the same and other users could continue using it.
- The workflow editor supports the BPEL workflow description language. This is required in order to deploy the workflow into a BPEL compliant workflow engine for its subsequent execution. Since BPEL lacks the ability to represent visually the workflows graphs, an additional language is needed to describe the visual representation of the workflow. For this language an XML format like SVG could be used or, since this is an internal representation of the workflows, some custom (proprietary) format could be devised.
- Import/Export functionality is needed to permit the interoperation with other workflow editing environments, e.g. Taverna. Especially for the import facility, two are the currently anticipated input formats: BPEL and Taverna's Scufl. Nevertheless, it is impossible to support the import of all Taverna workflows because of the additional, non Web Service based "processors" that Taverna supports, e.g. "local" Java classes.

3.3.2 External interface requirements

Being a web based application the Workflow Editor is hosted in the ACGT Portal in custom made portlet. This requirement has a number of implications as far as the design of the user interface is concerned (e.g. the same fonts, colors, etc. used in the rest of the portal) but it also has some consequences on the implementation; for example, the workflow editor's portlet should support "window minimization" and "maximization". In general, this tool should comply with the requirements and restrictions set by the Portal environment in terms of the presentation and the functionality.

Additionally, the Workflow Editor requires interaction with the following ACGT components:

- The Workflow Engine (Enactor), which is responsible for the execution of the workflows. The Workflow Editor is responsible for the deployment of an abstract workflow as a new concrete workflow which later can be invoked as a service and participate in other workflows. The communication between the workflow editor and the workflow engine depends on the interface offered by the engine and how the "hot" deployment of workflows is implemented.
- The majority of information required for the listing of the available service and the syntactic and semantic validation of workflows comes from the Metadata Repository. In order to support these advanced features the Workflow Editor requires a flexible and transparent interface and query language and therefore the choice is clearly the SPARQL RDF query language used over the corresponding access protocol .
- The Grid Data Management and VO specific Services are used in order to store and maintain the workflow definitions and distinguish between the public and private storage areas.

3.3.3 The architecture of the Workflow Editor

The Workflow Editor has been designed as an application consisting of two components:

- The client side, which runs as a "rich internet application" inside the browser, using familiar and well known web technologies like HTML and Javascript
- The server side, which is located in a central point and is responsible for more computational heavy or the core "business-logic" of the workflow editor.

These two components communicate with each other in a unidirectional, "firewall friendly" way, i.e. the client side makes requests to the server side and not the other way round. An implication of this is that implementation of "server push" scenarios, i.e. cases where the server needs to notify the client for an event of some sort, is difficult although there are a number of solutions. Currently though we don't have a need for such a functionality.

3.3.4 Technology Background

ACGT Workflow Editor is an internet application. Like all these applications Workflow Editor is highly accessible, require no installation, can be upgraded at any time, and offer access to large amounts of data without complex networks. These are the main advantages compared to a native, desktop application. Even though Internet applications usually have poorer usability due to their simpler, less interactive interfaces and slow update times, they are replacing native applications everywhere you look[5].

A Rich Internet Application (RIA) [AJAX] is an Internet application that attempts to bridge the usability gap between native applications and normal Internet ones. It contains more code on the browser, which offers higher levels of interactivity and an experience similar to native applications. With RIAs, it's possible to use many technologies, such as Flash, Java, and ActiveX, but the most important one is JavaScript. JavaScript is provided directly by the

[5] For example Google Docs (http://docs.google.com/) is a complete office suite similar in functionality to Microsoft Office that runs exclusively on the browser.

browser instead of being an add-on like the other technologies. Technologies used for the realization of ACGT Workflow Editor are: HTML, JavaScript, Ajax, and PHP.

3.3.4.1 JavaScript

Netscape invented JavaScript [JSAX] as a way to control the browser and add interactivity to Web pages. JavaScript referred to as a "scripting language" and has no connection at all the Java programming language. A JavaScript script is a program that is either contained internally in an HTML page (the original method of scripting) or resides in an external file (the now-preferred method).

There are many things that can be done with JavaScript to make Web pages more interactive and provide site's users with a better, more exciting experience. JavaScript can create an active user interface, giving the users feedback as they navigate to web pages.

But JavaScript is a client-side language; that is, it is designed to do its work on user's machine, not on the server. Because of this, JavaScript has some limitations built-in, mostly for security reasons:

- JavaScript does not allow the reading or writing of files on client machines. That's a good thing, because nobody wants a Web page to be able to read files off of its hard disk, or be able to write viruses onto the disk, or be able to manipulate the files on the computer. The only exception is that JavaScript can write to the browser's cookie file, and even then there are limitations.
- JavaScript does not allow the writing of files on server machines. There are a number of ways in which this would be handy (such as storing page hit counts or filled-out form data), but JavaScript isn't allowed to do that. Storing and handling data to serer can be done with a CGI written in a language such as Perl or PHP, or a Java program.
- JavaScript cannot close a window that it hasn't opened. This is to avoid a situation where a site takes over user's browser, closing windows from any other sites.
- JavaScript cannot read information from an opened Web page that came from another server. In other words, a Web page can't read any information

3.3.4.2 Ajax

Ajax [JSAX] is shorthand for "Asynchronous JavaScript and XML", and it is a term that was first coined in early 2005 by Jesse James Garrett[6], a Web developer and author. Strictly speaking, Ajax is just a small (although particularly popular) part of JavaScript. As commonly used, though, the term no longer refers to a technology by itself (like, say, Java or JavaScript).

In the larger scheme of things, what's generally referred to as Ajax is the combination of these technologies:

- XHTML
- CSS (Cascading Style Sheets)
- The DOM (Document Object Model) accessed using JavaScript
- XML, the format of the data being transferred between the server and the client
- XMLHttpRequest to retrieve data from the server

[6] http://www.adaptivepath.com/ideas/essays/archives/000385.php

By working as an extra layer between the user's browser and the web server, Ajax handles server communications in the background, submitting server requests and processing the returned data. The results may then be integrated seamlessly into the page being viewed, without that page needing to be refreshed or a new one loaded.

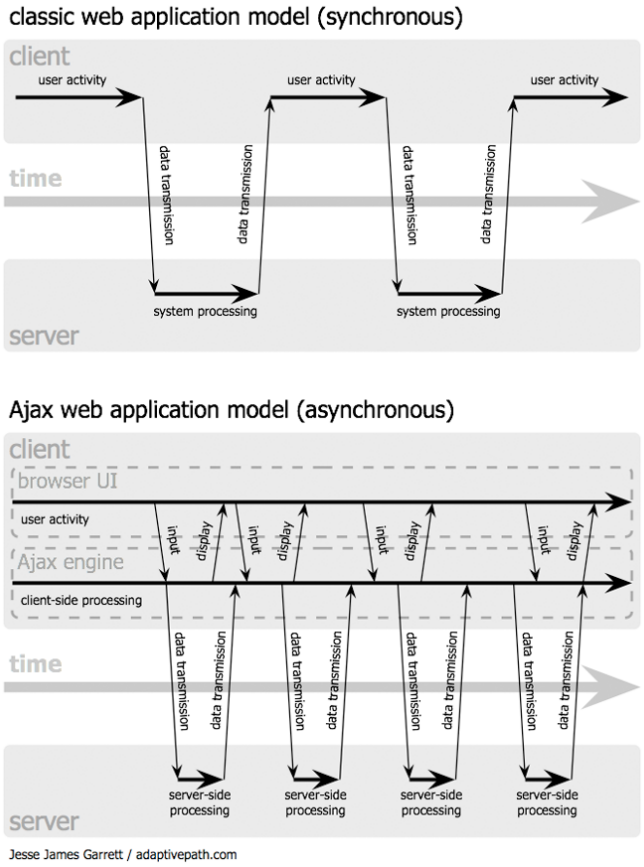

Figure 3-1 The Ajax asynchronous programming model (image courtesy of Jesse James Garrett)

As we can see in Figure 3-1, Ajax builds an extra layer of processing between the web page and the server. This layer, which is often referred to as an Ajax Engine or Ajax Framework, intercepts requests from the user and handles server communications in the background, quietly, unobtrusively, and asynchronously. By this we mean that server requests and responses no longer need to coincide with particular user actions but they may happen at any time convenient to the user and to the correct operation of the application.

In the traditional style of web page, when the user issue a server request via a hyperlink or a form submission, the server accepts that request, carries out any required server-side processing, and subsequently serves to the user a new page with content appropriate to the action you have undertaken.

While this processing takes place, the user interface is effectively frozen. User is made quite aware of this, when the server has completed its task, by the appearance in the browser of the new or revised page.

With asynchronous server requests, however, such communications occur in the background, and the completion of such a request does not necessarily coincide with a

screen refresh or a new page being loaded. The browser does not freeze and await the completion by the server of the last request but instead lets the user carry on scrolling, clicking, and typing in the current page.

The benefit to Ajax is that most of the processing for the application is happening within the user's browser, and requests to the server for data are usually short. So with Ajax, you can give users the kind of rich applications that depend on Web-based data, without the performance penalty of older approaches, which required that the server send back entire pages of HTML in response to user actions.

3.3.4.3 PHP

PHP stands for “PHP: Hypertext Preprocessor” [PHP]. PHP is a reflective programming language originally designed for producing dynamic web pages. It is used mainly in server-side scripting, but can be used from a command line interface or in standalone graphical applications.

PHP is a widely-used general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. PHP generally runs on a web server, taking PHP code as its input and creating Web pages as output.

Five important characteristics make PHP’s practical nature possible:

- Familiarity. Many of the language’s constructs are borrowed from C and Perl, and in many cases PHP code is almost indistinguishable from that found in the typical C or Pascal program.
- Simplicity: PHP engine simply begins executing the code after the first escape sequence (<?) and continues until it passes the closing escape sequence (?>). If the code is syntactically correct, it will be executed exactly as it is displayed.
- Efficiency: PHP 4.0 introduced resource allocation mechanisms and more pronounced support for object-oriented programming, in addition to session management features.
- Security: PHP provides developers and administrators with a flexible and efficient set of security safeguards. These safeguards can be divided into two frames of reference: system level and application level.

Flexibility: PHP is an embedded language, so it is extremely flexible. Although PHP is generally touted as being used in conjunction solely with HTML, it can also be integrated alongside languages like JavaScript, WML, XML, and many others. Additionally, as with most other mainstream languages, wisely planned PHP applications can be easily expanded as needed.

In our case PHP is the technology chosen for the implementation of the server side of the workflow editor. The important thing here is that this choice has no impact on the implementation of the client side: we could have chosen any other server side dynamic web page generating technology like Java Servlets or JSP, or ASP.Net, or even the traditional and currently not so popular CGI scripts. The communication between the client and the server side of the Workflow Editor is based on prearranged wire formats and the two sides are agnostic to each other’s implementation details.

3.3.5 The Workflow Editor in Action

Setting as background the Javascript, AJaX and PHP technologies we created the ACGT Workflow Editor. External, third party, open source libraries that are used in Workflow Editor are Ext JS (http://extjs.com/) and draw2d (http://www.openjacob.org/draw2d.html). Figure 3-2 shows the user interface when we invoke the Workflow Editor.

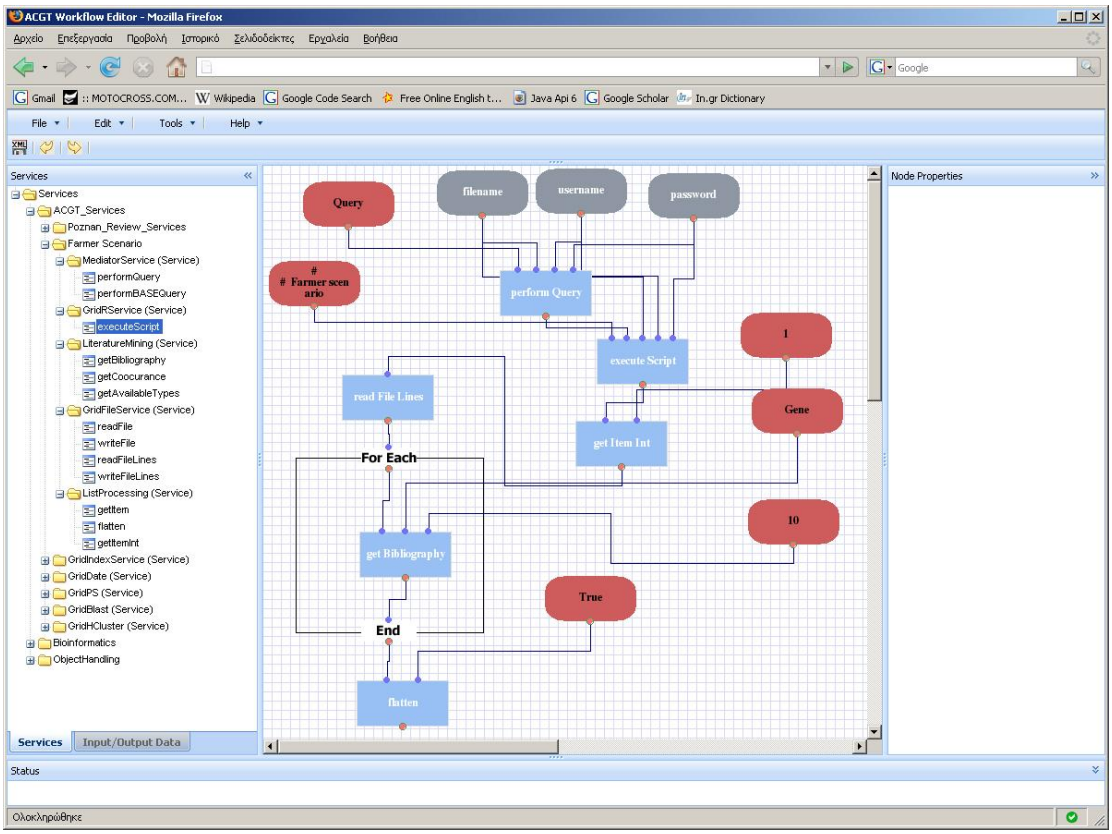

Figure 3-2 A general view of the workflow editor

As we can see the application consists of five frames:

- "North" frame contains a menu bar and menu buttons.
- "East" frame contains all the available services (according to the ACGT web service repository) that the user can use to create a workflow and like input/output variables, constants, loop branches.
- The centric frame is the main frame where the workflow is drawn.
- "West" frame contains information about the selected (if any) items that we have in the workflow.
- The "south" frame contains information about the workflow status (e.g. response from enactor, errors during deploy/run etc).

Interaction with frames is asynchronous, using Ajax technology, in order to gain flexibility and reduce the delay to the end user.

From the menu bar (Figure 3-3) the users have the options to create a new workflow (New button), open an existing workflow from the workflow repository, save the current workflow, and export the workflow to BPEL. Furthermore, the user during the workflow design can undo or redo operations, can deploy the current workflow or "run" the workflow (a workflow must have been deployed before it can be executed). Some of these operations (e.g. Open, Save, Export to BPEL, Deploy, Run) require from the Workflow Editor (client) communication with the server-side. PHP scripts are responsible for the client – server communication.

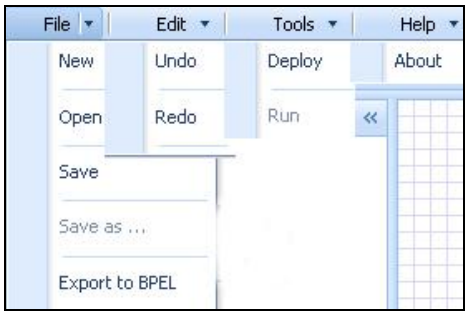

Figure 3-3 Menus in the Workflow Editor

From the east frame the user can select the objects that she/he wants to participate in the workflow. In the bottom of this frame the user can select the view "Services" or "input/output data". From the view "Services" the user can select all the available web services from the ACGT services repository. The services are hierarchically organized as a tree view (Figure 3-4, left side). A service can be added in the workflow by double clicking on it or using the drag and drop operation to the workflow area (centre frame). Then the service is dynamically generated (using asynchronous call to the service repository) and is visible in the workflow area. The "input/output data" view of the east frame gives the ability to the user to add objects in the workflow compliant with the BPEL standard, such as input/output objects, constants and loop operations.

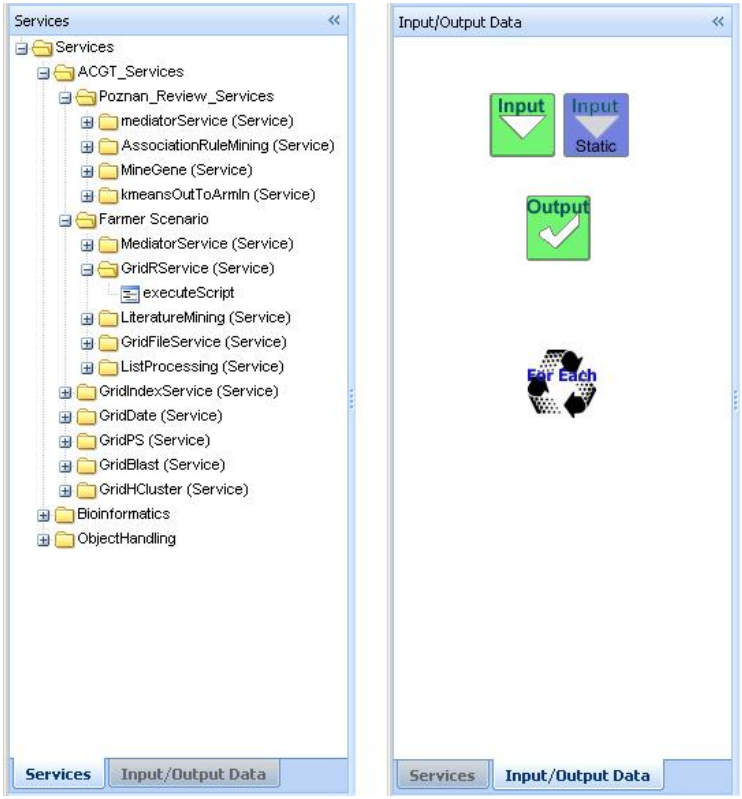

Figure 3-4 Left and right panel

West frame contains information about the selected (if any) items. This information varies depending on the object. If the object is an input then the information that the user get is the name and the data type for the specific input object. Same holds for the output. For the constants the information is the data type and the default value. For the loop object the information that is available to the user is the input/output data types and the name of the loop. For the services the information that the user gets is the service name, the input names – data types, the output names - data types. An example can be shown in Figure 3-5 where information about the GridR service of ACGT is shown.

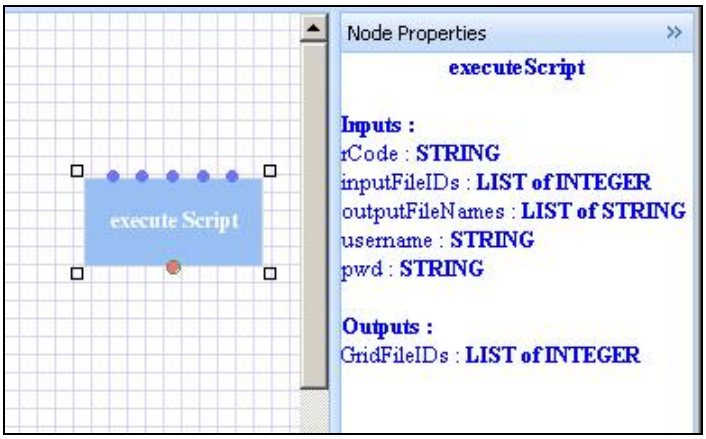

Figure 3-5 Context specific information

In Figure 3-6 the south frame is shown. It contains a status bar with information about the workflow status (e.g. response from enactor, errors during deploy/run etc).

Figure 3-6 Status bar

The central frame contains the workflow area where the user draws its workflow. The user can add as many objects as he/she wants and connect them respectively. The editor using an efficient algorithm prevents the user from connecting objects with different data types. If the user tries to connect objects with conflicting data types then a warning message appears (Figure 3-7).

Figure 3-7 Syntactic validation of services connections

Furthermore, the user can change the data type of the constants, the input, the output and the loop input/output objects simply by double clicking on the object. Then a new message box with a list of available data types appears and the user can select the appropriate. An example of this message box can be shown in Figure 3-8.

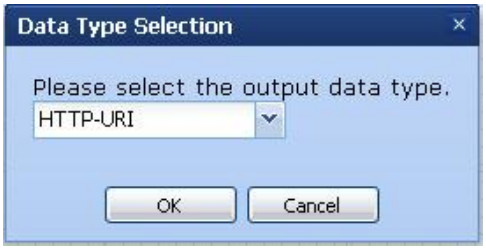

Figure 3-8 Data type annotation

Depending on the needs of the workflow the user can connect as many services as he/she wants, add loops, constants and input/output variables. An example of a workflow concerning the Farmer scenario can be shown in Figure 3-9. At any time the user can save the workflow and continue working on it at a different time. All the workflows are stored in the server side so the user can handle its workflow from any place using only the credentials to access the portal ACGT.

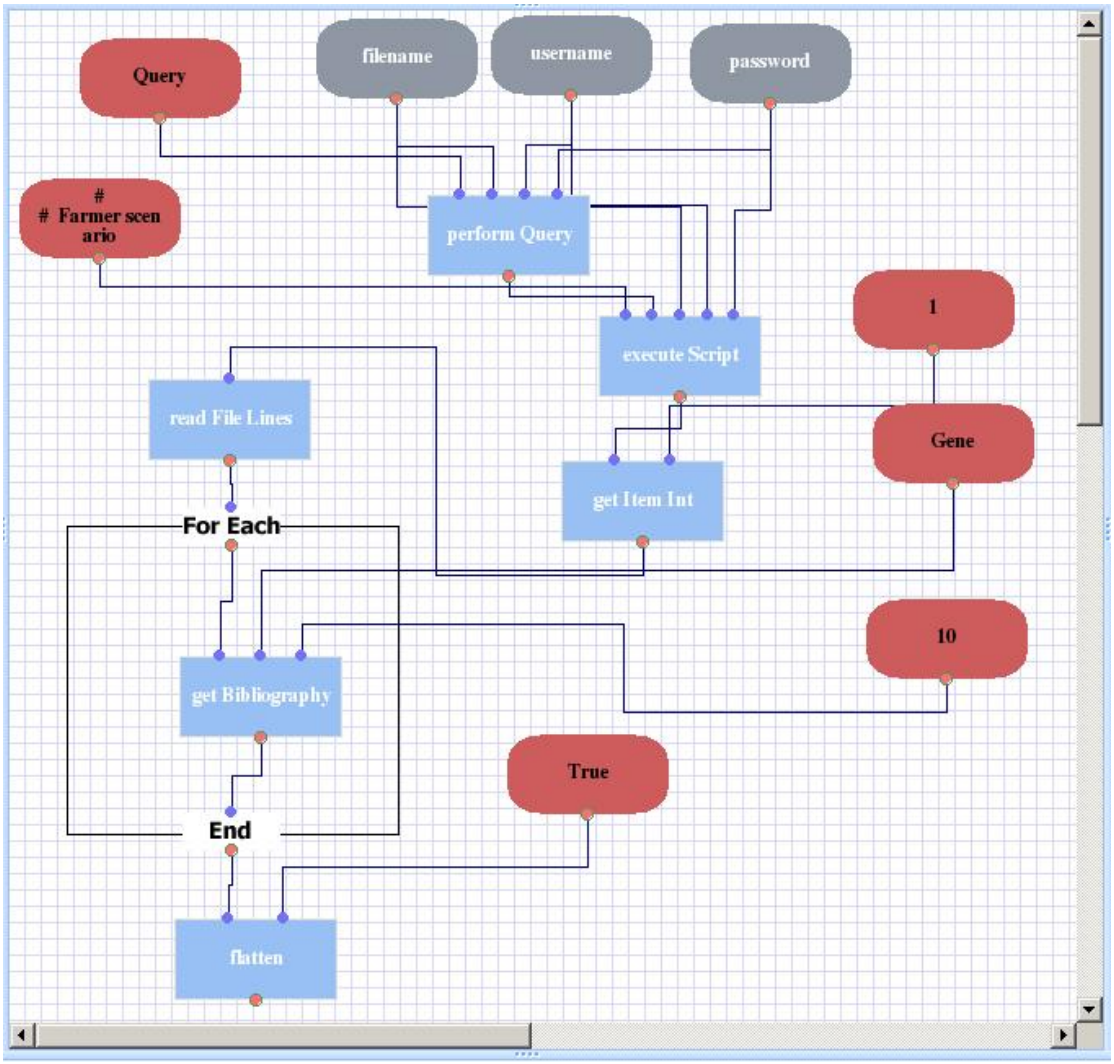

Figure 3-9 An example of a complex workflow

Move, delete, undo, redo operations are also available.

3.3.6 Performance Requirements

In terms of performance, the development of the Workflow Editor as a Rich Internet application, i.e. an application that is run inside the web browser, requires a great deal of forethought and good analysis and design. The presence of the network usually cannot be hidden due to its inherent latency so a good policy is to try to minimize the interactions (e.g. queries) with the server side or at least to overlap them with some user actions by doing them in the background. Minimization of the communication with the server side implies that as much as possible state is kept on the client, i.e. in the user's machine. An example of this is that the navigation of the available services should not require re-fetching of the same information if it has been retrieved in the past and can be reused. But also the user should not wait for the whole classification of services to be retrieved from the server and loaded in her browser. So another important policy is the "lazy evaluation", which means that

computations are delayed until their results are actually needed[7]. In the same example, an implementation of this policy will be that the hierarchy of services is loaded one level at a time and additional branches of the tree are retrieved only when the user requests for them, e.g. by clinking in their “root”.

So in conclusion the following design policies should be consider with respect to the performance and the user experience:

- Minimize communication with the server and keep as much as possible business logic in the client
- Perform the network communication interleaved with the user actions, i.e. concurrently in the background
- Cache expensive computations or the results of the interactions with the server side for as much as possible
- Adopt a lazy evaluation of computations and client – server communications to reduce the user perceived delay.

3.3.7 Security

The Workflow Editor as part of the portal is subject to the same access policies and security constraints: The users are authenticated and authorized according to their specific roles and based on the Grid security mechanisms.

3.3.8 Portability

Being web based the Workflow Editor does not require any special installation in the client side, except some modern web browser. Its implementation is based on state of the art technologies that are supported by the majority of the contemporary browsers, like Javascript, (X)HTML, XML, SVG. Nevertheless, since there are always slight incompatibilities, it is important to keep the code portable to at least the following browsers and versions: Internet Explorer 7, Mozilla Firefox 2, and Apple’s Safari 3.

3.3.9 Maintainability

The code is kept as modular as possible to permit its maintenance and evolution. The selection of some high quality libraries and tools for web development will support this modularity and also will keep the code clean by abstracting away the minor vagaries of the web browsers and operating systems.

3.4 Workflow Enactor

Each workflow is deployed as a "higher order" composite service and the Workflow Enactor is the Grid enabled component responsible for the invocation, monitoring, and management of running workflows. The standard workflow description language WS-BPEL (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel) has been selected as the workflow description format and being a standard it enables the separation of the workflow editing environment from the workflow enactor and facilitates their communication

[7] This is also known as a *Call-by-need* evaluation strategy.

and integration. Also Apache ODE (http://ode.apache.org/) is the workflow enactor chosen for the initial demonstration of the workflow environment.

3.4.1 BPEL

BPEL is an XML language for describing business process behaviour based on Web services. The BPEL workflow description language supports flow control, variables, concurrent execution, input and output, transaction scoping/compensation, and error handling.

Each BPEL document specifies the behaviour of a business process. BPEL processes often invoke Web services to perform functional tasks and can be either *abstract or executable*.

- Abstract processes are similar to library APIs: they describe what the process can do and its inputs and outputs but do not describe how anything gets done. Abstract processes are useful for describing a business process to another party that wants to use the process.

- Executable processes do the "heavy lifting" - they contain all of the execution steps that represent a cohesive unit of work.

A process consists of *activities* connected by *links.* (A process sometimes only contains one activity, which is usually a container for more activities.) The path taken through the activities and their links is determined by many things, including the values of variables and the evaluation of expressions.

The starting points are called start *activities*. When a start activity is triggered, a new business process instance is created. From then on, the instance is identified by data called *correlation sets*. These data uniquely identify a process, but they may change over time. For example, the correlation set for a process may begin as a purchase order number retrieved from a customer order. Later, when an invoice is generated, the correlation set may be the invoice number.

BPEL is layered on top of other Web technologies such as WSDL, XML Schema, XPath, XSLT, and WS Addressing.

3.4.2 BPEL Enactors

A BPEL Runtime Engine (Enactor) is the core software module which translates the BPEL description from files (e.g. BPEL documents, WSDLs, XML schemas, etc) to executable entities. Each executable BPEL process not only consists of Web Services but is also a Web Service itself, so each Engine has essentially to provide a SOAP stack for its processes. Another common fact for BPEL Engines is that they have to provide a means of persistent storage for their state, usually a relational database.

Various BPEL enactors exist, distributed both as open source tools or sold as commercial products. Below we outline a list of such enactors along with some brief information for each one. This is not a complete list, but we strongly believe that it contains the most prominent currently available.

- The Workflow Enactment Engine Project (**WEEP**, http://weep.gridminer.org/) is open source and freely distributed under the “Apache Software License, v2.0”. The WEEP project is also accepted as a Globus Incubator project. The WEEP enactor aims to

implement an easy to use and manage workflow enactment engine for WS-I/WSRF services and orchestrate the services as described by WS-BPEL 2.0. The WEEP project seems very promising as far as the integration with Grid Technologies is concerned, due to its tight connections with the Globus Toolkit. It is the only available enactor to express support both for WS-I and WSRF Web Services and we feel that WEEP could more easily be integrated to the ACGT Security Architecture. Nevertheless, at the moment WEEP lacks of general community support and also has only partially implemented the WS-BPEL 2.0 standard. As the project evolves and full compliance with BPEL 2.0 is achieved, WEEP seems to be a good alternative enactor to be used by ACGT.

- The **ActiveBPEL** (http://www.active-endpoints.com/active-bpel-engine-overview.htm) open source engine is a commercial-grade open source implementation of the Business Process Execution Language (BPEL) standard. The ActiveBPEL engine comes in two different distributions, and is made available either as a commercial product or as a feature limited product distributed under the GNU General Public License (GPL). The ActiveBPEL runtime enactor seems to be the most mature implementation of BPEL 2.0. Also, ActiveBPEL does not translate BPEL to a different internal format but the core architecture is build up based on the notions of BPEL. Furthermore, ActiveBPEL expresses its commitment to keep updating its implementation to support subsequent revisions of BPEL and also offers for free a visual BPEL editor (ActiveBPEL Designer). However, being a by-product of a commercial product we hesitate to integrate ACGT with it and we prefer to keep it as valuable alternative in case the other implementations fail to meet our expectations.

- The **Apache Orchestration Director Engine (ODE)** is an open source BPEL enactor that is written in Java, relies on JDK v5.0 features and is freely distributed under the "Apache Software License". It supports both the WS-BPEL 2.0 OASIS standard and the legacy BPEL4WS 1.1 vendor specification. It also features "hot-deployment" of processes, a compiled approach to BPEL that provides detailed analysis and validation at the command line or at deployment, and management interfaces for processes, instances and messages. Finally it bases its implementation on an efficient and sound theoretical model of concurrency [ACTORS].

In our initial evaluation of the available BPEL Enactors we have concluded that ODE is the preferred one based on its feature set, its open source development model and the strong community support, and the scientifically documented approach for its architecture. These reasons don't imply that other existing implementations don't have some of these features (or even better), but we feel that ODE represents the "golden ratio".

Nevertheless, although in the current phase of the ACGT project integration we decided to adopt the ODE Enactor, as long as other BPEL compliant enactors exist there should be easy to turn over to another solution, should the implementation of ODE shows signs of incompetence. This is also the strong advantage of adopting Open Standards such as BPEL, which offer the ability to choose among different implementations without having to tightly integrate with only one of them.

3.5 Metadata

In order to support the composition of the ACGT services the workflow environment should have access to the metadata descriptions of these services. These descriptions play a critical role in the integration in the ACGT ecosystem because they provide useful information about

the functional and non functional characteristics of the ACGT components. The metadata make it possible to achieve the discovery, composition, validation, and quality of service verification of the services and tools and therefore enrich the whole platform with advanced functionality and ease and facilitate the user interactions.

In the context of the Workflow Environment there are two metadata related scenarios: the use of existing metadata and the provision of additional metadata related to the workflows. We are further describing each of them in the following paragraphs.

3.5.1 Metadata used by the Workflow Environment

In the current configuration the metadata that are required by the Workflow Editor are used for the following purposes:

- The implementation of the browsing and navigation to the available services
- The syntactic composition of the services

In the first case we need to have information about what are the available processing elements that can be put in a workflow. In the current version of the workflow editor the available services are shown in a hierarchical view based on their *functional categories* i.e. their functionality as classified in the metadata repository (see Figure 3-4, left panel). Additional ways of presenting the same information can be the combination of browsing and searching based on multiple classification schemes (*faceted classification*), e.g. by functional category and then by input or output parameter data types.

3.5.1.1 Workflow Editor and Data Types

For the syntactic composition of services the metadata needed is mostly about the parameter data types. The data types used currently are a subset of the basic (primitive) data types proposed by the XML Schema [XSD]:

- String
- Boolean
- Integer
- Float
- Double
- AnyURI

The only compound data type is the so called[8] LIST which is used to represent sequences (arrays, streams) or collections of elements with the same data type, for example "LIST of STRING", "LIST of BOOLEAN", etc.

[8] As a tribute to the Lisp family of programming languages and the functional programming.

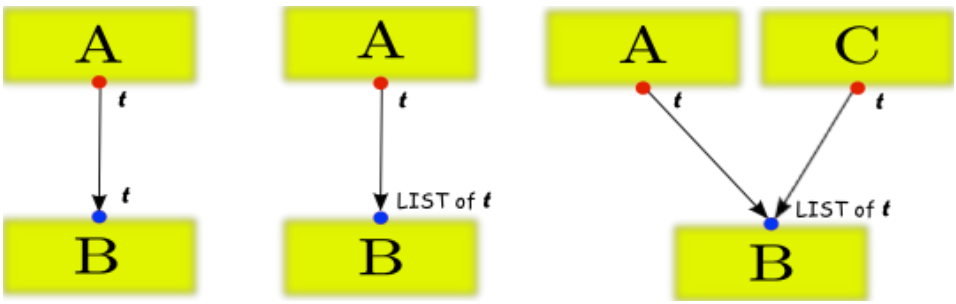

Figure 3-10 Typing rules for the Workflow Editor

In the current prototype version of the workflow editor two services A and B can be connected through a direct link if an output parameter of A has the same data type with an input parameter of B or if B's input parameter has a data type that is a LIST of A's output parameter data type ("lifting"). In the latter case a data type ***t*** can be automatically "lifted" to a data type "LIST of ***t***" because data of type ***t*** is also a sequence (LIST) of a single element of type ***t*** (Figure 3-10). This automatic list construction is also performed when multiple outputs of type t are connected to a single input of type "LIST of ***t***" as shown in the right of Figure 3-10.

3.5.2 Workflow Metadata

Each workflow is registered in the metadata repository as two entities: Firstly a new workflow is a new service that can be executed and participate in more complex workflows as any other atomic service. Furthermore, being a workflow, it should have additional information recorded, such as the services that it's consisted of or its machine friendly description (i.e. the BPEL document). In the future, we expect that more metadata will be needed, such as monitoring information, quality criteria such a performance, etc.

3.5.3 Metadata Repository

The tool metadata repository in ACGT provides functionality to publish, discover and maintain metadata about tools (web services and workflows). An initial proposal for metadata was part of Deliverable 6.1 (section 3.1.9) and currently most of this metadata is supported by the tool metadata repository. Examples of metadata for services include operations, parameters (input/output), data type of parameters, functional category for services and workflows and exportable definitions of services/workflows (WSDL/BPEL).

This metadata is used to provide several necessary functionalities, for example registration, discovery, invocation and documentation of tools. An application programming interface (API) has been developed to provide these functionalities. The API has been implemented as SOAP web services using Axis2. We have used Hibernate to implement persistent java objects that map to database tables. These java objects are used to hold the metadata information.

4 Open issues and Future Work

There are a number of open issues that should be addressed in the final ACGT integrated environment. First and foremost, the integration of the Grid security to the workflow enactor is the most challenging task. Currently none of the enactors we have studied supports the Grid Security Infrastructure (GSI) set of standards. In particular the enactor should support the ***delegation*** of user rights [D11.2] so that all the services that participate in a workflow are contacted by the enactor *in the name of the end user* with no need for the user to be present during the workflow execution. An additional thing for the enactor and the workflow execution layer is the provision of *monitoring functionality* that will enable the supervision and control of the workflow enactment by the end user through the portal environment. This will provide a feedback on the status of the execution, useful information in the presence of errors, etc.

The work on the Workflow Editor will be continued and we foresee the following major directions to be followed:

- **Support for more control structures**. Currently the *ForEach* iteration structure is the only control mechanism that augments the sequential flow and invocation of services in a workflow. Additional structures such as choice ("*if/then/else*", "*switch*") or other repetition forms ("*repeat/until*", "*while*") seem to be needed to support the end user scenarios. The driving force for the inclusion of such additional constructs in the workflow editing environment will be the simplicity of "data flow" paradigm but it may be proved that these control structures require a more *imperative* model[9] of visual programming, e.g. the introduction of variables and assignment operators.

- **Support of a more user friendly view of XML structures and messages.** We expect that certain ACGT web services will need data structures more complex than the primitive ones. As described in Section 3.5.1 currently the `LIST` construct is the only complex data structure but we may need more advanced ones in the future. This issue is related to the Type Theory of programming languages [TAPL] and recent advancements in the incorporation of XML in programming languages (e.g. [Mei03]) could be sources for inspiration.

- **Support and facilitate the *semantic* composition of services.** The validation and checking performed by the workflow editor in the prototype version is limited in the syntactic data type information of services' parameters. This of course is not enough because, for example, in a rather simplistic case, the output of a service A that corresponds to the *height* of a person can be given to the input of a service B that corresponds to the *weight* of a person. Therefore syntactic information (i.e. the literal form and structure of the data in the XML or other serialization) is not sufficient for verifying the correctness of a workflow: semantics (i.e. the "meaning", the high level concepts that data represent) is also needed. Semantic information is also related to the services' metadata and therefore this imposes the requirement for the metadata

[9] Imperative programming, in contrast with the declarative programming, is a programming paradigm that describes computation as statements that change a program state. State is usually represented by variables that change their values after the execution of assignment statements. FORTAN, C/C++, and Java are examples of imperative programming languages whereas functional or logic programming languages like Haskell and Prolog follow the declarative programming paradigm.

repository to store also the semantic concepts that the parameters represent[10]. If this information is available then an additional constraint for the linear composition of two services will be added: the two services' parameters that are connected should have the same semantic type *or* the semantic type of the first (output) should be more specific (i.e. a "subtype"/"subclass") than the semantic type of the second (input)[11].

- **Compliance with the Taverna Workbench**. Taverna is more or less the *de facto* standard for designing and executing workflows in the domain of bioinformatics. Hence the implementation of functionality for the import and export of workflows in the Scufl, Taverna's internal format is something that should the focus of future work. This is again a demanding task since the semantics of Scufl and BPEL do not match exactly. For example in Taverna the user can have workflows that contain, in addition to standard Web Services, plenty of other "activity types", e.g. Biomart services, Beanshell scripts, etc.

- **Portal integration**. The workflow editor should be seamlessly integrated into the ACGT portal environment in order to be easily accessible by the end user and to take advantage of the portal's facilities (e.g. authentication of the users). This does not seem to be a painless task since the portal technology ("Java Portlets") requires a strict discipline in code development and the most part of logic should be implemented in the server side while in the Workflow Editor there's a big portion of the "business logic" implemented in the client side.

In conclusion, the work on the integration and interoperability of services in relation to the security and performance quality criteria and the refinement of metadata and the metadata infrastructure should continue to be among our top priorities.

[10] This does not mean that the whole ACGT Master Ontology (http://www.ifomis.org/acgt/1.0) should be kept inside the metadata repository. According to the Semantic Web technologies each concept described in the Master Ontology is identified by a URI and therefore only this information (link) needs to be kept in the repository for each parameter.

[11] This second case is the application of the subsumption relation or the *Liskov Substitution Principle (http://en.wikipedia.org/wiki/Liskov_substitution_principle)* used in the object oriented programming.

References

[ACTORS] Gul Agha “ACTORS: A Model of Concurrent Computation in Distributed Systems”. Doctoral Dissertation, MIT Press. 1986, http://hdl.handle.net/1721.1/6952

[AJAX] Joshua Eichorn. Understanding AJAX ISBN: 978-0132216357

[D6.2] ACGT Deliverable 6.2 “Demonstration and report of data mining and knowledge discovery tools and services”

[D11.2] ACGT Deliverable 11.2 “Implementation of the ACGT core security services & initial implementation of the Pseudonymisation tool”

[JSAX] Tom Negrino JavaScript and Ajax for the Web ISBN: 978-0321430328

[Mei03] Meijer, E., Schulte, W. & Bierman, G., Programming with circles, triangles and rectangles, in Proceedings of the XML Conference, Philadelphia, United States, 7-12 December 2003 http://research.microsoft.com/~emeijer/Papers/XML2003/xml2003.html

[PHP] W. Jason Gilmore A Programmer's Introduction to PHP 4.0 ISBN: 978-1893115859

[TAPL] Pierce, Benjamin C. *Types and Programming Languages*. MIT Press, March 2002.

[WSBP] Web Service Interoperability Organization *Basic Profile version 1.1* http://www.ws-i.org/Profiles/BasicProfile-1.1.html

[XSD] XML Schema Part 2: Datatypes Second Edition, W3C Recommendation 28 October 2004, http://www.w3.org/TR/xmlschema-2/

Appendices

Appendix A - Installation and configuration of ODE

ODE comes in two different distribution types, either as a WAR (web archive) file or a JBI (Java Business Integration) package. The WAR distribution can be used inside an application container such as Apache Tomcat, while the JBI distribution can be used in a JBI container such as Apache ServiceMix. The rest of the configuration guidelines apply to the WAR distribution, deployed inside the Tomcat application container so please refer to the ODE documentation if you are interested in the other types of software. Furthermore, the information below refers to the distribution of the described software modules which we had available at the time when this document was edited.

- Apache Tomcat v5.5.25
- Apache ODE v1.1, web archive (WAR) distribution
- MySQL DBMS v5.0.27
- MySQL Connector/J v5.1.5

Installation of ODE into a Tomcat container

To install ODE inside Tomcat please follow these steps[12]:

- Install Tomcat. We'll refer to the installation directory as `$CATALINA_HOME`.
- Unzip the ODE distribution package and copy the `ode.war` to the `$CATALINA_HOME/webapps/` directory.
- Download the Apache Commons FileUpload library and copy the .jar file to the `$CATALINA_HOME/common/lib/` directory.

Configuration of the database

The default installation of ODE contains an internal database (Apache Derby) for persistent storage of information such as processes, jobs, events etc. The Derby database has many advantages like its small footprint, fast and easy deployment and an embedded JDBC driver. However, it is not suitable for large installations where robust functioning is demanded due to heavy load and many deployed processes. Below we outline the necessary steps to configure an installation of ODE to function with an external MySQL database. Similar steps apply also for other external databases (all the major DBMS are supported) and the relevant information can be found at the documentation of ODE.

1. Create a database (e.g. named "`ode`") to your MySQL instance, and a user account ("`user`", "`password`") with appropriate access rights to this database.

[12] In the installation and configuration instructions we assume that a Linux operating system is used. If a Microsoft Windows system is used instead the path separator should be changed to "`\`" and the environment variables should start with "`%`" instead of "`$`"

2. Load to the "ode" database the schema of the database, which is described in Appendix B.

Configuration of the ODE installation

1. Install ODE (see above).

2. Start/restart the container (Tomcat) at least once so that the ODE web application has been deployed in the container.

3. Drop a copy of the .jar file of the MySQL JDBC Driver (MySQL Connector/J) library to the $CATALINA_HOME/common/lib/ directory of Tomcat.

4. Edit the configuration file of Tomcat ($CATALINA_HOME/conf/server.xml) inside the <Host> element and add the following stanza:

```
<Context path="/ode" docBase="ode" debug="5" reloadable="true" crossContext="true">
          <Resource name="jdbc/ODEDB" auth="Container" type="javax.sql.DataSource"
                    maxActive="100" maxIdle="30" maxWait="10000"
                    username="user" password="password"
                    driverClassName="com.mysql.jdbc.Driver"
                    url="jdbc:mysql://hostname:3306/ode?autoReconnect=true"/>
```

5. Add a file named ode-axis2.properties under the $CATALINA_HOME/webapps/ode/WEB-INF/conf/ directory with the following content:

```
ode-axis2.db.mode=EXTERNAL
ode-axis2.db.ext.dataSource=java:comp/env/jdbc/ODEDB
```

6. Start the container (Tomcat). Always check the log files for errors or warnings which could indicate possible malfunctioning.

Deploying a Process in Ode

Each deployment is a directory with all relevant deployment artefacts. At the minimum it will contain the deployment descriptor, one or more process definitions (BPEL or .cbp), WSDL and XSDs (excluding those compiled into the .cbp). It may also contain other files, such as SVGs or XSLs. The deployment descriptor is a file named deploy.xml (see the documentation for its description).

During deployment, the process engine loads all documents from the deployment descriptor. Loading documents allow it to reference processes, service and schema definitions using fully qualified names, and import based on namespaces instead of locations.

To deploy in Ode, just copy the whole directory containing your artifacts (the directory itself, not only its content) in the path $DEPLOYMENT_ROOT/WEB-INF/processes (in Tomcat it would be $CATALINA_HOME/webapps/ode/WEB-INF/processes).

Another option is to compile an Axis archive (.aar) and upload it using the Ode web service management interface (DeploymentService), which essentially has the same result as above: to create a directory with all necessary artifacts and deploy the service.

Appendix B - ODE and MySQL

In order to load the schema to your MySQL DB follow the steps below.

- Store the SQL code (see below) to a file (e.g. “ode-schema.sql”)
- Connect to your database with an account having appropriate access privileges
- Execute from the command prompt the “ode-schema.sql”.

```
mysql> \. ode-schema.sql
mysql>
```

The SQL code that builds the ODE database is the following:

```
CREATE TABLE ODE_ACTIVITY_RECOVERY (ID BIGINT NOT NULL, ACTIONS VARCHAR(255), ACTIVITY_ID
BIGINT, CHANNEL VARCHAR(255), DATE_TIME DATETIME, DETAILS TEXT, REASON VARCHAR(255), RETRIES
INTEGER, INSTANCE_ID BIGINT, PRIMARY KEY (ID)) TYPE = innodb;

CREATE TABLE ODE_CORRELATION_SET (CORRELATION_SET_ID BIGINT NOT NULL, CORRELATION_KEY
VARCHAR(255), NAME VARCHAR(255), SCOPE_ID BIGINT, PRIMARY KEY (CORRELATION_SET_ID)) TYPE =
innodb;

CREATE TABLE ODE_CORRELATOR (CORRELATOR_ID BIGINT NOT NULL, CORRELATOR_KEY VARCHAR(255),
PROC_ID BIGINT, PRIMARY KEY (CORRELATOR_ID)) TYPE = innodb;
CREATE TABLE ODE_CORSET_PROP (ID BIGINT NOT NULL, PROP_KEY VARCHAR(255), PROP_VALUE
VARCHAR(255), CORRSET_ID BIGINT, PRIMARY KEY (ID)) TYPE = innodb;

CREATE TABLE ODE_EVENT (EVENT_ID BIGINT NOT NULL, DETAIL VARCHAR(255), DATA BLOB, SCOPE_ID
BIGINT, TSTAMP DATETIME, TYPE VARCHAR(255), INSTANCE_ID BIGINT, PROCESS_ID BIGINT, PRIMARY KEY
(EVENT_ID)) TYPE = innodb;

CREATE TABLE ODE_FAULT (FAULT_ID BIGINT NOT NULL, ACTIVITY_ID INTEGER, DATA TEXT, MESSAGE
VARCHAR(255), LINE_NUMBER INTEGER, NAME VARCHAR(255), PRIMARY KEY (FAULT_ID)) TYPE = innodb;

CREATE TABLE ODE_MESSAGE (MESSAGE_ID BIGINT NOT NULL, DATA TEXT, TYPE VARCHAR(255),
MESSAGE_EXCHANGE_ID VARCHAR(255), PRIMARY KEY (MESSAGE_ID)) TYPE = innodb;

CREATE TABLE ODE_MESSAGE_EXCHANGE (MESSAGE_EXCHANGE_ID VARCHAR(255) NOT NULL, CALLEE
VARCHAR(255), CHANNEL VARCHAR(255), CORRELATION_ID VARCHAR(255), CORRELATION_KEYS
VARCHAR(255), CORRELATION_STATUS VARCHAR(255), CREATE_TIME DATETIME, DIRECTION INTEGER, EPR
TEXT, FAULT VARCHAR(255), FAULT_EXPLANATION VARCHAR(255), OPERATION VARCHAR(255),
PARTNER_LINK_MODEL_ID INTEGER, PATTERN VARCHAR(255), PIPED_ID VARCHAR(255), PORT_TYPE
VARCHAR(255), PROPAGATE_TRANS BIT, STATUS VARCHAR(255), CORR_ID BIGINT, PARTNER_LINK_ID
BIGINT, PROCESS_ID BIGINT, PROCESS_INSTANCE_ID BIGINT, REQUEST_MESSAGE_ID BIGINT,
RESPONSE_MESSAGE_ID BIGINT, PRIMARY KEY (MESSAGE_EXCHANGE_ID)) TYPE = innodb;

CREATE TABLE ODE_MESSAGE_ROUTE (MESSAGE_ROUTE_ID BIGINT NOT NULL, CORRELATION_KEY
VARCHAR(255), GROUP_ID VARCHAR(255), ROUTE_INDEX INTEGER, CORR_ID BIGINT, PROCESS_INSTANCE_ID
BIGINT, PRIMARY KEY (MESSAGE_ROUTE_ID)) TYPE = innodb;
CREATE TABLE ODE_MEX_PROP (ID BIGINT NOT NULL, PROP_KEY VARCHAR(255), PROP_VALUE VARCHAR(255),
MEX_ID VARCHAR(255), PRIMARY KEY (ID)) TYPE = innodb;

CREATE TABLE ODE_PARTNER_LINK (PARTNER_LINK_ID BIGINT NOT NULL, MY_EPR TEXT, MY_ROLE_NAME
VARCHAR(255), MY_ROLE_SERVICE_NAME VARCHAR(255), MY_SESSION_ID VARCHAR(255), PARTNER_EPR TEXT,
PARTNER_LINK_MODEL_ID INTEGER, PARTNER_LINK_NAME VARCHAR(255), PARTNER_ROLE_NAME VARCHAR(255),
PARTNER_SESSION_ID VARCHAR(255), SCOPE_ID BIGINT, PRIMARY KEY (PARTNER_LINK_ID)) TYPE = in-
nodb;

CREATE TABLE ODE_PROCESS (ID BIGINT NOT NULL, GUID VARCHAR(255), NUMBER_OF_INSTANCES INTEGER,
PROCESS_ID VARCHAR(255), PROCESS_TYPE VARCHAR(255), VERSION BIGINT, PRIMARY KEY (ID)) TYPE =
innodb;
```

```
CREATE TABLE ODE_PROCESS_INSTANCE (ID BIGINT NOT NULL, DATE_CREATED DATETIME, EXECUTION_STATE
BLOB, LAST_ACTIVE_TIME DATETIME, LAST_RECOVERY_DATE DATETIME, PREVIOUS_STATE SMALLINT,
SEQUENCE BIGINT, INSTANCE_STATE SMALLINT, FAULT_ID BIGINT, INSTANTIATING_CORRELATOR_ID BIGINT,
PROCESS_ID BIGINT, ROOT_SCOPE_ID BIGINT, PRIMARY KEY (ID)) TYPE = innodb;

CREATE TABLE ODE_SCOPE (SCOPE_ID BIGINT NOT NULL, MODEL_ID INTEGER, SCOPE_NAME VARCHAR(255),
SCOPE_STATE VARCHAR(255), PARENT_SCOPE_ID BIGINT, PROCESS_INSTANCE_ID BIGINT, PRIMARY KEY
(SCOPE_ID)) TYPE = innodb;

CREATE TABLE ODE_XML_DATA (XML_DATA_ID BIGINT NOT NULL, DATA TEXT, IS_SIMPLE_TYPE BIT, NAME
VARCHAR(255), SCOPE_ID BIGINT, PRIMARY KEY (XML_DATA_ID)) TYPE = innodb;
CREATE TABLE ODE_XML_DATA_PROP (ID BIGINT NOT NULL, PROP_KEY VARCHAR(255), PROP_VALUE
VARCHAR(255), XML_DATA_ID BIGINT, PRIMARY KEY (ID)) TYPE = innodb;

CREATE TABLE OPENJPA_SEQUENCE_TABLE (ID TINYINT NOT NULL, SEQUENCE_VALUE BIGINT, PRIMARY KEY
(ID)) TYPE = innodb;

CREATE TABLE STORE_DU (NAME VARCHAR(255) NOT NULL, DEPLOYDT DATETIME, DEPLOYER VARCHAR(255),
DIR VARCHAR(255), PRIMARY KEY (NAME)) TYPE = innodb;

CREATE TABLE STORE_PROCESS (PID VARCHAR(255) NOT NULL, STATE VARCHAR(255), TYPE VARCHAR(255),
VERSION BIGINT, DU VARCHAR(255), PRIMARY KEY (PID)) TYPE = innodb;

CREATE TABLE STORE_PROCESS_PROP (id BIGINT NOT NULL, PROP_KEY VARCHAR(255), PROP_VAL
VARCHAR(255), PRIMARY KEY (id)) TYPE = innodb;

CREATE TABLE STORE_PROC_TO_PROP (ProcessConfDaoImpl_PID VARCHAR(255), element_id BIGINT) TYPE
= innodb;

CREATE TABLE STORE_VERSIONS (id BIGINT NOT NULL, VERSION BIGINT, PRIMARY KEY (id)) TYPE = in-
nodb;

CREATE INDEX I_D_CTVRY_INSTANCE ON ODE_ACTIVITY_RECOVERY (INSTANCE_ID);
CREATE INDEX I_D_CR_ST_SCOPE ON ODE_CORRELATION_SET (SCOPE_ID);
CREATE INDEX I_D_CRLTR_PROCESS ON ODE_CORRELATOR (PROC_ID);
CREATE INDEX I_D_CRPRP_CORRSET ON ODE_CORSET_PROP (CORRSET_ID);
CREATE INDEX I_OD_VENT_INSTANCE ON ODE_EVENT (INSTANCE_ID);
CREATE INDEX I_OD_VENT_PROCESS ON ODE_EVENT (PROCESS_ID);
CREATE INDEX I_OD_MSSG_MESSAGEEXCHANGE ON ODE_MESSAGE (MESSAGE_EXCHANGE_ID);
CREATE INDEX I_D_MSHNG_CORRELATOR ON ODE_MESSAGE_EXCHANGE (CORR_ID);
CREATE INDEX I_D_MSHNG_PARTNERLINK ON ODE_MESSAGE_EXCHANGE (PARTNER_LINK_ID);
CREATE INDEX I_D_MSHNG_PROCESS ON ODE_MESSAGE_EXCHANGE (PROCESS_ID);
CREATE INDEX I_D_MSHNG_PROCESSINST ON ODE_MESSAGE_EXCHANGE (PROCESS_INSTANCE_ID);
CREATE INDEX I_D_MSHNG_REQUEST ON ODE_MESSAGE_EXCHANGE (REQUEST_MESSAGE_ID);
CREATE INDEX I_D_MSHNG_RESPONSE ON ODE_MESSAGE_EXCHANGE (RESPONSE_MESSAGE_ID);
CREATE INDEX I_D_MS_RT_CORRELATOR ON ODE_MESSAGE_ROUTE (CORR_ID);
CREATE INDEX I_D_MS_RT_PROCESSINST ON ODE_MESSAGE_ROUTE (PROCESS_INSTANCE_ID);
CREATE INDEX I_D_MXPRP_MEX ON ODE_MEX_PROP (MEX_ID);
CREATE INDEX I_D_PRLNK_SCOPE ON ODE_PARTNER_LINK (SCOPE_ID);
CREATE INDEX I_D_PRTNC_FAULT ON ODE_PROCESS_INSTANCE (FAULT_ID);
CREATE INDEX I_D_PRTNC_INSTANTIATINGCORRELATOR ON ODE_PROCESS_INSTANCE
(INSTANTIATING_CORRELATOR_ID);
CREATE INDEX I_D_PRTNC_PROCESS ON ODE_PROCESS_INSTANCE (PROCESS_ID);
CREATE INDEX I_D_PRTNC_ROOTSCOPE ON ODE_PROCESS_INSTANCE (ROOT_SCOPE_ID);
CREATE INDEX I_OD_SCOP_PARENTSCOPE ON ODE_SCOPE (PARENT_SCOPE_ID);
CREATE INDEX I_OD_SCOP_PROCESSINSTANCE ON ODE_SCOPE (PROCESS_INSTANCE_ID);
CREATE INDEX I_D_XM_DT_SCOPE ON ODE_XML_DATA (SCOPE_ID);
CREATE INDEX I_D_XMPRP_XMLDATA ON ODE_XML_DATA_PROP (XML_DATA_ID);
CREATE INDEX I_STR_CSS_DU ON STORE_PROCESS (DU);
CREATE INDEX I_STR_PRP_ELEMENT ON STORE_PROC_TO_PROP (element_id);
CREATE INDEX I_STR_PRP_PROCESSCONFDAOIMPL_PID ON STORE_PROC_TO_PROP (ProcessConfDaoImpl_PID);

# IMPORTED FROM OPENSYMPHONY QUARTZ, SEE NOTICE FILE FOR DETAILS
#
# In your Quartz properties file, you'll need to set
# org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.StdJDBCDelegate
#
# By: Ron Cordell - roncordell
#  I didn't see this anywhere, so I thought I'd post it here. This is the script from Quartz
#  to create the tables in a MySQL database, modified to use INNODB instead of MYISAM.

DROP TABLE IF EXISTS QRTZ_JOB_LISTENERS;
DROP TABLE IF EXISTS QRTZ_TRIGGER_LISTENERS;
DROP TABLE IF EXISTS QRTZ_FIRED_TRIGGERS;
```

```
DROP TABLE IF EXISTS QRTZ_PAUSED_TRIGGER_GRPS;
DROP TABLE IF EXISTS QRTZ_SCHEDULER_STATE;
DROP TABLE IF EXISTS QRTZ_LOCKS;
DROP TABLE IF EXISTS QRTZ_SIMPLE_TRIGGERS;
DROP TABLE IF EXISTS QRTZ_CRON_TRIGGERS;
DROP TABLE IF EXISTS QRTZ_BLOB_TRIGGERS;
DROP TABLE IF EXISTS QRTZ_TRIGGERS;
DROP TABLE IF EXISTS QRTZ_JOB_DETAILS;
DROP TABLE IF EXISTS QRTZ_CALENDARS;
CREATE TABLE QRTZ_JOB_DETAILS(
JOB_NAME VARCHAR(80) NOT NULL,
JOB_GROUP VARCHAR(80) NOT NULL,
DESCRIPTION VARCHAR(120) NULL,
JOB_CLASS_NAME VARCHAR(128) NOT NULL,
IS_DURABLE VARCHAR(1) NOT NULL,
IS_VOLATILE VARCHAR(1) NOT NULL,
IS_STATEFUL VARCHAR(1) NOT NULL,
REQUESTS_RECOVERY VARCHAR(1) NOT NULL,
JOB_DATA BLOB NULL,
PRIMARY KEY (JOB_NAME,JOB_GROUP))
TYPE=InnoDB;

CREATE TABLE QRTZ_JOB_LISTENERS (
JOB_NAME VARCHAR(80) NOT NULL,
JOB_GROUP VARCHAR(80) NOT NULL,
JOB_LISTENER VARCHAR(80) NOT NULL,
PRIMARY KEY (JOB_NAME,JOB_GROUP,JOB_LISTENER),
INDEX (JOB_NAME, JOB_GROUP),
FOREIGN KEY (JOB_NAME,JOB_GROUP)
REFERENCES QRTZ_JOB_DETAILS(JOB_NAME,JOB_GROUP))
TYPE=InnoDB;

CREATE TABLE QRTZ_TRIGGERS (
TRIGGER_NAME VARCHAR(80) NOT NULL,
TRIGGER_GROUP VARCHAR(80) NOT NULL,
JOB_NAME VARCHAR(80) NOT NULL,
JOB_GROUP VARCHAR(80) NOT NULL,
IS_VOLATILE VARCHAR(1) NOT NULL,
DESCRIPTION VARCHAR(120) NULL,
NEXT_FIRE_TIME BIGINT(13) NULL,
PREV_FIRE_TIME BIGINT(13) NULL,
TRIGGER_STATE VARCHAR(16) NOT NULL,
TRIGGER_TYPE VARCHAR(8) NOT NULL,
START_TIME BIGINT(13) NOT NULL,
END_TIME BIGINT(13) NULL,
CALENDAR_NAME VARCHAR(80) NULL,
MISFIRE_INSTR SMALLINT(2) NULL,
JOB_DATA BLOB NULL,
PRIMARY KEY (TRIGGER_NAME,TRIGGER_GROUP),
INDEX (JOB_NAME, JOB_GROUP),
FOREIGN KEY (JOB_NAME,JOB_GROUP)
REFERENCES QRTZ_JOB_DETAILS(JOB_NAME,JOB_GROUP))
TYPE=InnoDB;

CREATE TABLE QRTZ_SIMPLE_TRIGGERS (
TRIGGER_NAME VARCHAR(80) NOT NULL,
TRIGGER_GROUP VARCHAR(80) NOT NULL,
REPEAT_COUNT BIGINT(7) NOT NULL,
REPEAT_INTERVAL BIGINT(12) NOT NULL,
TIMES_TRIGGERED BIGINT(7) NOT NULL,
PRIMARY KEY (TRIGGER_NAME,TRIGGER_GROUP),
INDEX (TRIGGER_NAME, TRIGGER_GROUP),
FOREIGN KEY (TRIGGER_NAME,TRIGGER_GROUP)
REFERENCES QRTZ_TRIGGERS(TRIGGER_NAME,TRIGGER_GROUP))
TYPE=InnoDB;

CREATE TABLE QRTZ_CRON_TRIGGERS (
TRIGGER_NAME VARCHAR(80) NOT NULL,
TRIGGER_GROUP VARCHAR(80) NOT NULL,
CRON_EXPRESSION VARCHAR(80) NOT NULL,
TIME_ZONE_ID VARCHAR(80),
PRIMARY KEY (TRIGGER_NAME,TRIGGER_GROUP),
INDEX (TRIGGER_NAME, TRIGGER_GROUP),
FOREIGN KEY (TRIGGER_NAME,TRIGGER_GROUP)
```

```
REFERENCES QRTZ_TRIGGERS(TRIGGER_NAME,TRIGGER_GROUP))
TYPE=InnoDB;

CREATE TABLE QRTZ_BLOB_TRIGGERS (
TRIGGER_NAME VARCHAR(80) NOT NULL,
TRIGGER_GROUP VARCHAR(80) NOT NULL,
BLOB_DATA BLOB NULL,
PRIMARY KEY (TRIGGER_NAME,TRIGGER_GROUP),
INDEX (TRIGGER_NAME, TRIGGER_GROUP),
FOREIGN KEY (TRIGGER_NAME,TRIGGER_GROUP)
REFERENCES QRTZ_TRIGGERS(TRIGGER_NAME,TRIGGER_GROUP))
TYPE=InnoDB;

CREATE TABLE QRTZ_TRIGGER_LISTENERS (
TRIGGER_NAME VARCHAR(80) NOT NULL,
TRIGGER_GROUP VARCHAR(80) NOT NULL,
TRIGGER_LISTENER VARCHAR(80) NOT NULL,
PRIMARY KEY (TRIGGER_NAME,TRIGGER_GROUP,TRIGGER_LISTENER),
INDEX (TRIGGER_NAME, TRIGGER_GROUP),
FOREIGN KEY (TRIGGER_NAME,TRIGGER_GROUP)
REFERENCES QRTZ_TRIGGERS(TRIGGER_NAME,TRIGGER_GROUP))
TYPE=InnoDB;

CREATE TABLE QRTZ_CALENDARS (
CALENDAR_NAME VARCHAR(80) NOT NULL,
CALENDAR BLOB NOT NULL,
PRIMARY KEY (CALENDAR_NAME))
TYPE=InnoDB;

CREATE TABLE QRTZ_PAUSED_TRIGGER_GRPS (
TRIGGER_GROUP VARCHAR(80) NOT NULL,
PRIMARY KEY (TRIGGER_GROUP))
TYPE=InnoDB;

CREATE TABLE QRTZ_FIRED_TRIGGERS (
ENTRY_ID VARCHAR(95) NOT NULL,
TRIGGER_NAME VARCHAR(80) NOT NULL,
TRIGGER_GROUP VARCHAR(80) NOT NULL,
IS_VOLATILE VARCHAR(1) NOT NULL,
INSTANCE_NAME VARCHAR(80) NOT NULL,
FIRED_TIME BIGINT(13) NOT NULL,
STATE VARCHAR(16) NOT NULL,
JOB_NAME VARCHAR(80) NULL,
JOB_GROUP VARCHAR(80) NULL,
IS_STATEFUL VARCHAR(1) NULL,
REQUESTS_RECOVERY VARCHAR(1) NULL,
PRIMARY KEY (ENTRY_ID))
TYPE=InnoDB;

CREATE TABLE QRTZ_SCHEDULER_STATE (
INSTANCE_NAME VARCHAR(80) NOT NULL,
LAST_CHECKIN_TIME BIGINT(13) NOT NULL,
CHECKIN_INTERVAL BIGINT(13) NOT NULL,
RECOVERER VARCHAR(80) NULL,
PRIMARY KEY (INSTANCE_NAME))
TYPE=InnoDB;

CREATE TABLE QRTZ_LOCKS (
LOCK_NAME VARCHAR(40) NOT NULL,
PRIMARY KEY (LOCK_NAME))
TYPE=InnoDB;

INSERT INTO QRTZ_LOCKS values('TRIGGER_ACCESS');
INSERT INTO QRTZ_LOCKS values('JOB_ACCESS');
INSERT INTO QRTZ_LOCKS values('CALENDAR_ACCESS');
INSERT INTO QRTZ_LOCKS values('STATE_ACCESS');
INSERT INTO QRTZ_LOCKS values('MISFIRE_ACCESS');

# Apache ODE - SimpleScheduler Database Schema
# MySQL scripts by Maciej Szefler.
DROP TABLE IF EXISTS ODE_JOB;

CREATE TABLE ODE_JOB (
  jobid CHAR(64)  NOT NULL DEFAULT '',
```

```
  ts BIGINT  NOT NULL DEFAULT 0,
  nodeid char(64)  NULL,
  scheduled int  NOT NULL DEFAULT 0,
  transacted int  NOT NULL DEFAULT 0,
  details blob(4096)  NULL,
  PRIMARY KEY(jobid),
  INDEX IDX_ODE_JOB_TS(ts),
  INDEX IDX_ODE_JOB_NODEID(nodeid)
)
TYPE=InnoDB;

COMMIT;
```