Contract N° IST-034824

Workpackage 2
Automatic Code Generation from Models

Deliverable 2.6
Prototype of web-based User Interface with interfaces to existing SBVR tools

Project funded by the European Community under the "Information Society Technology" Programme

Contract number: IST-034824
Project acronym: OPAALS
Title: Open Philosophies for Associative Autopoietic Digital Ecosystems

Deliverable N°: D2.6
Due date: May 2009
Delivery date: June 2009

Short description:
This report together with the corresponding code deliverable documents the integration efforts achieved within all WP2 components into the umbrella project Sepiax-Web. Additionally the interface intergation between Sepiax-Web and Guigoh is described and the possible extention points are introduced.

Author: SUAS (R. Eder, T. Kurz, T. Heistracher), T6-Eco (A. Filieri and A. Margarito)
Partners contributed: SUAS, T6-Eco
Made available to: OPAALS Consortium and European Commission

Versioning		
Version	**Date**	**Author, Organisation**
0.1	10/6/2009 (first submission)	SUAS, T6-Eco
0.2	18/6/2009 (review of UniKassel)	SUAS, T6-Eco

Quality check
1st internal reviewer: Oxana Lapteva (UniKassel)
2nd internal reviewer: Tadinada Venkata Prabhakar (IITK)
3rd internal reviewer: na

Dependencies:

Work Packages	WP2
Achievements*	Integration of components developed within WP2 and integration of Sepiax-Web-Editor into Guigoh.
Partners	SUAS, T6-Eco
Domains	Mainly computer science and engineering topics are coverd in this "code plus report" deliverable.
Targets	This deliverable targets the computer science partners of the OPAALS consortium and parties interested in the reuse of prototypical components developed within WP2.
Publications*	none
PhD Students*	none
Outstanding features*	Integration of many components and applications into one code trunk with appropriate build environemnt. Preparation for the connetion to the OPAALS infrastructure by using standadrized interfaces. Optionally the integration of pending semantic search infrastructure can be considered.
Disciplinary domains of authors*	R. Eder (Information Technologies, Software and Systems Engineering) T. Kurz (Information Technologies, Software and Systems Engineering, Interpersonal Communication) T. Heistracher (Information Technologies, Software and Systems Engineering, Biophysical Modelling) A. Filieri (Computer Science and Business Modelling Domain) A. Margarito (Computer Science and Business Modelling Domain)

The information marked with an asterisk (*) is provided in order to address Recommendation n. 4 from the Year 2 review report.

Contents

Executive summary

This report gives an overview about the code delivered as part of the Sepiax initiative, combining all prototypical implementation efforts performed within WP 2. Sepiax-Web is a full Ajax-based web application enabling users to access all related functionality from within one platform. The modularized design and the existing components of this application are described in order to facilitate integration of 3rd party components as well. Where the description of the components is not given in that very detail, references to the related project deliverables are given. In addition to that, the build process is described in order to understand how own custom-made packages can be built and deployed. This report ends with the more detailed description of the integration of Sepiax and Guigoh as part of a SUAS – IPTI researcher exchange and the description of the integration of the workflow generation component developed by partner T6-Eco.

1 Introduction

Deliverable 2.6 is a code deliverable including detailed descriptions of the conceptual background reported within the document at hand. This report summarises the integration of the prototypical components developed within Workpackage 2 of the OPAALS Project. As an umbrella project the *Sepiax* initiative was introduced in Deliverable 2.2 [1] and presented at the Year 2 Review in Brussels (see `www.sepiax.org`). The source code of the components developed individually are combined there as the Sepiax-Web-Editor. This Editor is the main focus of this report and the related code deliverable.

Chapter 2 provides an overview of Sepiax-Web. The integrated components are there described individually to give the reader the possibility to understand the source code published via Subversion (SVN). Important data structures are outlined and leveraged key-libraries are described at that point. This part concludes with an introduction to the build process, which allows readers to download and build their own distribution files to be run in a Servlet Container. Also technical limitation concerning the tested JDK and Servlet Engines is provided.

Following that, Chapter 3 focuses on the work achieved during a researcher exchange of SUAS researchers at IPTI: the integration between Guigoh and the Sepiax-Web-Editor. This part describes a use case, developed to let users leverage a sub-part of the Sepiax-Web-Editor functionality. As Guigoh and Sepiax-Web are both Ajax-based web applications, the advantage to let them operate separately in a loosely coupled manner is described at that point. Both applications stay separate but are integrated by leveraging existing interfaces or creating new interfaces.

Finally the integration of the Automatic Workflow Generation component is described in Chapter 4. As the WP 2 deliverables already cover the components themselves in full detail, this chapter focuses on the integration aspects only.

Sepiax-Web is a prototypical approach bringing together work achieved within WP2 of OPAALS. This project is the first open-source Ajax-based web editors based

on the OMG standard *SBVR* [2]. One of the key features is the presented approach to generate XPDL out of SBVR vocabulary add rules and directly deploy the result in a workflow engine. The user should have the possibility to manage and search SBVR statements and reuse this data with the integrated components. The benefit of Sepiax-Web is that researchers within related domains can work with an already integrated set of software components with clearly defined and described interfaces. The success of this approach is shown with the described integration between Sepiax-Web and Guigoh described in this report. This documents, together with previous WP2 deliverables, should act as an introduction into the reuse and extension of Sepiax-Web and its components. The described application is now ready to be presented to early adopters, researchers and engineers interested in the work performed within this work package and SBVR in general.

Basic knowledge of Deliverables 2.2 [1], 2.4 [3], 2.4 [4] and 2.5 [5] is needed for understanding the individual parts of this effort.

2 Design and Architecture of Sepiax-Web

As many modern web applications, Sepiax-Web is built using Ajax technique, leveraging the *Google Web Toolkit (GWT)* framework. Therefore, Sepiax-Web is separated into two components: (i) the Sepiax-Web or running as a JavaScript application in the user's web browser and (ii) the Sepiax-Web services providing large parts of functionality. The services are hard to implement in JavaScript and need more resources available in the web browser's sandbox.

This chapter explains the architecture of Sepiax-Web and starts with a discussion of GWT in order to understand the separation of the individual components. After that some basic building blocks are explained that are necessary to understand extension points of Sepiax-Web. Finally the integrated components are outlined.

2.1 Google Web Toolkit Framework

Sepiax-Web is built using the *Google Web Toolkit (GWT)* Ajax Framework [6]. The initial prototype was developed using Version 1.4 and was migrated to Version 1.5 in the beginning of 2009 after Google released this version end of 2008. Since April 2009 Version 1.6 of GWT is released and it is planned to migrate to this new version before end of 2009.

GWT was chosen out of many other Ajax Frameworks due to its Java to JavaScript cross-compiling feature. This means that developers of Ajax Applications do not need to write any JavaScript code. All code is written in Java. While the code running on the server side is compiled to Java Bytecode, the parts of the code running on the Web Browser are compiled to JavaScript. The parts of the code needed on both parts are consequently compiled to both target platforms. These two worlds: (i) the server-world and (ii) the browser-world are glued together by using the Web

service paradigm. GWT offers multiple ways to call server side code from within the client application. The easiest way as it hides most coding from the developers is the usage of GWT Remote Procedure Calls (RPC) described in Section 2.1.3.

The following sections describe the package structure required for the cross-compiling feature. After that widgets are explained, as the user interface is built upon them. Finally the RPC mechanism is explained in detail in the section about GWT that is concluded with an explanation on how to deploy and run GWT applications in development and production environments.

2.1.1 GWT Package Structure

As mentioned before different source files are compiled to different target platforms. Therefore GWT requires a certain package structure. Such a package structure could be:

1. `at.somepackage`
2. `at.somepackage.client`
3. `at.somepackage.server`

This kind of structure provides not only the necessary information to the compiler, but also helps the developer to distinguish between the different class types. In this example the class files contained in the first package would be compiled to Java Bytecode *and* JavaScript code. Classes in the second package would only be compiled to JavaScript. Finally the classes in the third package would only be compiled to Java Bytecode, respectively.

The source code is compiled to JavaScript by GWT's own compiler. Therefore, when developing classes that need to be compiled to JavaScript, certain rules have to be followed, as not all language constructs of Java can be compiled to JavaScript by GWT's compiler. Additionally not all Java classes and libraries are available in the JavaScript. Details on which language constructs and which Java classes can be used can be found in the JRE Emulation Reference of the GWT.

2.1.2 GWT Widgets

Every GWT application needs an entry point. This entry point is a class that contains the initial main user interface. Such an user interface consists of one or

more widgets. Such a widget in GWT is an element of the graphical user interface and can be a button, a text area or a composition of other widgets.

GWT offers a broad palette of different widgets similar to other user interface toolkits like Java Swing or the Standard Widget Toolkit (SWT). The Sepiax-Web editor consists of a number of different widget compositions, for example the SbvrEntryComposite - to create and edit Entry objects. Later on in this document, the created Sepiax-Web editor selected widgets are explained individually.

2.1.3 GWT Remote Procedure Calls

There are different methods of RPCs possible within the GWT framework. With Sepiax-Web two of them were used: For the Guigoh-Integration HTTP-Requests applying JavaScript Object Notation (JSON) were used, as this is data exchange format is already supported by Guigoh. This approach is aligned with the Representational State Transfer (REST) approach and will be described in more detail in Section 3.2. All other components were integrated using the *native* GWT-approach. With these two RPC mechanisms, all supported functionality could be integrated.

The native approach of the GWT for creating asynchronous RPC, is to create two interfaces for one service endpoint. One interface specifying the exact method calls that can be performed. This interface itself has to extend the GWT *RemoteService* interface. Such an interface can be seen in Listing 2.1. This created interface consequently has to be implemented by the service implementation class which itself has to be inherited by an abstract *RemoteServiceServlet*. As the name implies the Java servlet mechanism is used to call the published remote procedures.

Yet there is nothing included that indicates this service being asynchronous. For that GWT requires a second interface to be created using the same name as the originally created interface but adding the keyword *Async* to the name as can be seen in Listing 2.2. Note that there is no return value anymore but an additional parameter - an *AsyncCallback* instance. When calling the service within JavaScript code, such an AsyncCallback instance has to be passed to the call. This is needed because of Ajax, a service call is non blocking but asynchronous and the return value is later on passed to the application by calling the *onSuccess* or *onFailure* methods of the AsyncCallback object.

Listing 2.1: Interface of the WordNet Service

```
public interface WordnetService extends RemoteService {
```

```
    // Lookup word in the WordNet dictionary
    public String getWordnetSummary(String word) throws ServiceException;

}
```

Listing 2.2: Asynchronous interface of the WordNet service

```
public interface WordnetServiceAsync {

    public void getWordnetSummary(String word, AsyncCallback callback);

}
```

Complex components that cannot be compiled to JavaScript have to reside on the server-side and are integrated by such procedure calls or service calls. As many components of Sepiax-Web are very complex as well, most of them are running server side. These components are explained in more detail in Section 2.3.

Details and manuals of GWT can be found on the GWT homepage [6].

2.1.4 Running GWT Applications

There are two different methods to run GWT applications. Firstly the hosted mode, which is used for debugging purposes. Secondly the *web mode*, which is the production way of running applications.

For debugging applications in hosted mode, a GWT shell has to be launched. This GWT shell needs the compiled classes, the Java source code and all needed libraries in its class-path. Additionally the entry point for the application is needed as a command line parameter. Using all this information, the GWT shell compiles the necessary JavaScript files and launches a bundled Mozilla engine to run the GWT application. This approach has a number of advantages at development time.

Firstly, the JavaScript code is immediately regenerated, if the Java source is changed. This is not done in the *web mode*. Secondly, debugging works seamless with modern IDEs, as the GWT shell integrates very well. For example breakpoints can be set in the IDE and the GWT shell automatically tracks them in the JavaScript code.

One pitfall of this method is, that it is necessary to have the Java source code in the classpath as well, as it is needed by the GWT compiler for creating the JavaScript code. This is untypical, as for running Java applications no source code is needed anymore.

The second method for running GWT applications is the web mode. This is the way to run the final application in a productive environment. In order to do that,

the JavaScript code has to be compiled manually using the GWT compiler. This can be achieved by running a GWT compiler from the command line, or better by using a build tool such as Ant.

The output of the GWT compiler are the JavaScript files, that, together with the static web content, can be simply put onto a web server, where a web browser can load and run the GWT application from. However, if there are Web services involved (e.g. GWT RPC services), a service container is needed for hosting the back-end services of the GWT application. The best way to bundle the JavaScript files and the back-end services together, is using the build tool and let it generate a Web Archive that can be deployed on a standard J2EE Container such as Apache Tomcat. In the case of Sepiax-Web, an Ant-Buildfile is provided that supports building the application with one command, which is explained later on in this deliverable.

2.2 Basic Building Blocks of Sepiax-Web and Extension Points

The following Section describes the data model of Sepiax-Web. The important part is depicted in Figure 2.1, where the fundamental classes *Entry*, *EntryReceiver* and *EntryReceiverFactory* are shown. The reason for these classes being so important is that every data entity managed in the Sepiax-Web editor is of the type *Entry*. If components are added to Sepiax-Web, they are either services providing functionality, or sinks *receiving Entries*. Consequently, these sinks have to implement the *EntryReceiver* interface and have to be registered at the *EntryReceiverFactory*.

The services providing functionality are built with the previously described GWT service method. Sepiax-Web services are described in Section 2.3. How sinks are added using the *EntryReceiver* interface is described in 2.2.2.

2.2.1 Entry

The entry-class serves as a container for the data stored in the repository. Three attributes were added for the Guigoh integration described in Chapter 3: *uri*, *authorUri* and *keywords*. The *Universally Unique Identifier (UUID)* attribute contains the unique key and is automatically generated by the repository for an item according to RFC 4122. Besides collecting the individual SBVR expressions, different

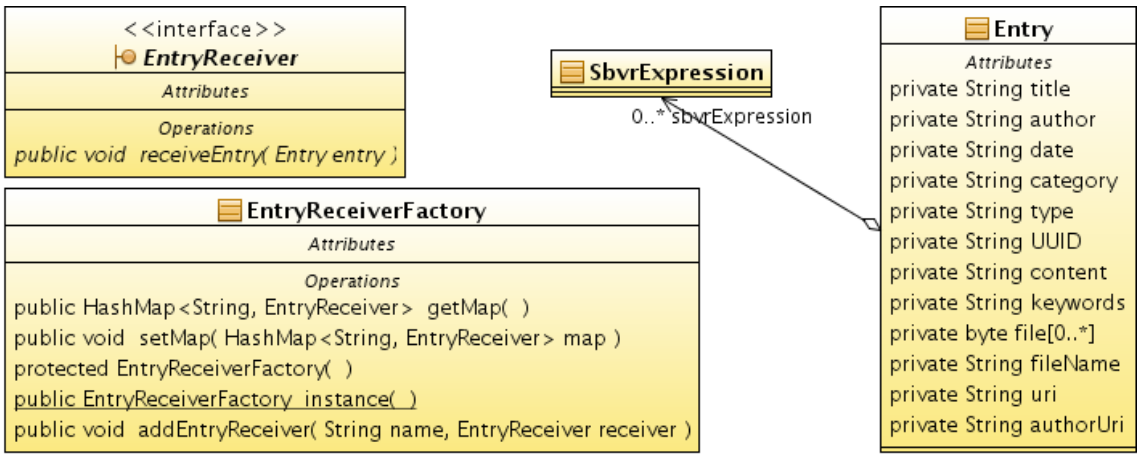

Figure 2.1: Sepiax-Web Entry Model

meta data fields are included such as title, author, date of last change, category and type. The attributes file and fileName are unused at the moment but should allow the attachment of corresponding documents to individual entries.

As a consequence for using unstructured storage of data in the repository, additional fields can be easily included by adapting the *Entry* class and modifying the mapping of the attribute to the stored notes in the repository service, which is described later on in this deliverable.

2.2.2 EntryReceiver

One important way to extend Sepiax-Web is to create a sink for *Entries* by implementing the *EntryReceiver* interface shown in Listing 2.3. This new *EntryReceiver* then has to be registered with a name by invoking the *addEntryReceiver* method of the *EntryReceiverFactory* class. After that, all Entries can be sent to this new sink by selecting the registered name.

At the current stage the following sinks are implemented:

- An entry can be opened in the editor to change the SBVR statements or the meta data of the entries.
- Entries can be submitted to the XPDL generator.
- Entries can be sent to the Grails generator.

It is easy to extend the functionality of Sepiax-Web in this manner with functionality that takes individual entries as input. As not all functionality can be implemented as extensions, the following sections describe the included components, that are based on the GWT service infrastructure.

Listing 2.3: EntryReceiver interface

```
public interface EntryReceiver {

    // Implement this method to Receive Entries
    public void receiveEntry(Entry entry);

}
```

2.3 Components of Sepiax-Web

Sepiax-Web consists of a number of different components, all integrated as GWT services.

2.3.1 Natural Language Parser

The SBVR parsing component of Sepiax-Web is based on the probabilistic parser provided by *The Stanford Natural Language Processing Group* [7]. Syntax trees, generated by this parser, are analyzed by an algorithm created by partner UniKassel described in Chapter 2 of Deliverable 2.2 [1]. The output of this algorithm is SBVR Vocabulary, that, together with the complementary SBVR rules, serves as input for either the Grails generator (Deliverable 2.1 [8] and 2.2 [1]), the XPDL generator (Deliverable 2.2 [1] ,2.3 [3] and 2.4 [4]) or any other in future integrated component.

2.3.2 Repository Implementation

As the distributed repository is still pending, the Apache Jackrabbit repository [9] was used as a data store. Apache Jackrabbit serves as the reference implementation of the Java Content Repository (JCR) API, also known as JSR 170 [10]. By using a content repository following industry standards, other repository implementations can be used if necessary.

The features of Jackrabbit used for Sepiax-Web are the hierarchical storage of unstructured data described in this section, and the inbuilt Lucene search engine described in the next section.

Upon startup, the `initService` method is invoked, which triggers the connection to the locally created JackRabbit repository. This repository is created in the working directory. Which means that upon deployment, e.g. on a Tomcat, a vanilla repository is created. This repository stays in place until it is deleted manually, even if new versions of the Sepiax-Web application are deployed. Because of the unstructured data store, fields can be added without modifying the repository's data definition.

Important as well is that Sepiax-Web stores its data in an own namespace in the repository, which means that it can coexist with other applications using the same repository as well. This will be important when Sepiax-Web can be migrated to the P2P repository in near future. If no namespace support will be provided, the repository service has to adapted in that way.

As already mentioned in Section 2.2.1, the primary key of entries stored in the repository is the UUID, which is automatically assigned by the JackRabbit engine. As can be seen in Listing 2.4, there are three operations currently supported by the repository service:

1. `storeData` - Stores the entry in the repository. If the submitted entry has no UUID, a new node is created. Otherwise the existing node is replaced with the new Data.

2. `deleteEntry` - Deletes the entry from the Repository by using the UUID.

3. `searchData` - Searches the repository using the inbuilt Lucene search engine.

The following section explains the integrated Lucene search engine of the Jackrabbit repository.

Listing 2.4: Interface of the repository service

```
public interface RepoService extends RemoteService {

    // Initialise the Service Runtime
    public void initService();

    // Store an Entry into the Repository
    public String storeData(Entry entryToAdd)
        throws RepoOperationException;

    // search Entries in Repository
    public Entry[] searchData(String title, String author, String date,
        String category, String type, String content)
        throws RepoOperationException;
```

```
    // delete Entry from Repository
    public void deleteEntry(Entry entry)
        throws RepoOperationException;
}
```

2.3.3 Search in the Repository

Jackrabbit provides an embedded *Lucene search engine*. Apache Lucene [11] is a high-performance text search engine written in Java and used in many other open source projects.

Jackrabbit supports different Query Languages such as SQL or XPath. Queries are then translated into Abstract Query Trees (AQT) which then are used to perform searches in the previously generated Lucene Index. Every time an entry is stored or updated in the Jackrabbit repository, the Lucene Index is updated as well. This enables Sepiax-Web to provide fast search capabilities. It is possible to search for meta data as well as for SBVR code within the search fields.

The search service is integrated in Sepiax-Web as part of the repository service described in the previous section as it is embedded in the leveraged Jackrabbit repository implementation.

2.3.4 Coding Assistance using WordNet

The *WordNet* service is integrated into Sepiax-Web to help users formulate their SBVR expressions.

WordNet is a large lexical database of English currently available in Version 3.0 at the homepage of the Cognitive Science Laboratory of Princeton University [12]. For accessing the WordNet database from within Java the Java API for WordNet Searching (JAWS) is used. The JAWS liberary is developed by Brett Spell at the Southern Methodist University [13].

Individual words shown in the SBVR widgets can be selected. If a user clicks on such a word, the WordNet service is invoked in the background. When a result is received by the Sepiax-Web editor, a small dialog widget pops up above the clicked word and shows helpful data about that word.

As this is a prototypical application not all data available in the WordNet database is displayed to the user. The interface for the service is provided in Listing 2.5. A sample screenshot of such a word explanation is depicted in Figure 2.2. In this

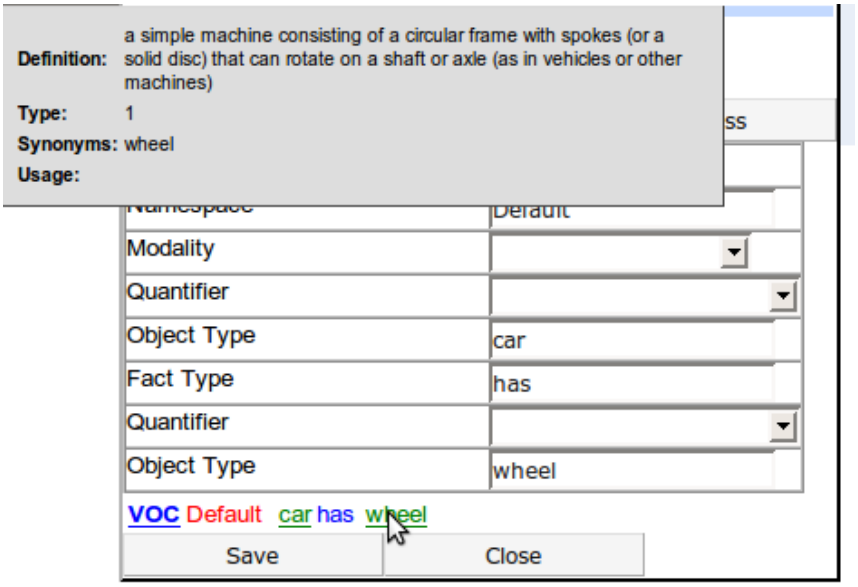

Figure 2.2: Sepiax-Web WordNet Expample

figure a dialog can be seen, providing further information of the selected word. This dialog can contain all data provided by the WordNet dictionary. The *definition*-field provides the exact definition of the selected word. The *type*-field provides the internal WordNet type of that word, which can be ignored at that point. Below that, available synonyms are shown to the user. If available, additional usage samples are provided in the last field of the dialog.

Listing 2.5: Interface of the WordNet service

```
public interface WordnetService extends RemoteService  {

    // Initialise the Service Runtime
    public Boolean initService();

    // Lookup word in the WordNet dictionary
    public String getWordnetSummary(String word) throws ServiceException;

}
```

2.3.5 Grails Generator Service

The Grails generation service consists of two different service calls: (i) the Grails Transformation service itself and (ii) the Grails Deployment service. The interfaces of these service calls can be seen in Listing 2.6

The Grails Transformation service takes the SBVR vocabulary as input and transforms it to the Grails application. This transformation is described in detail in the deliverables 2.1 [8] and 2.2 [1]. The next step is to bundle the created files together with all necessary Grails files to a web archive (WAR) that can be deployed on a Tomcat servlet container. This WAR file is the result value of the first service, which then serves as the input value of the second service.

The Grails Deployment service takes the WAR file as input, invokes another service and deploys it on a Tomcat servlet container. When the deployment is successfully finished, the URL to the generated and deployed Grails application is returned by the service to the editor. This enables the Sepiax-Web editor to finally redirect the web browser to the newly created application.

Listing 2.6: Interface of the Grails generation service

```
public interface GrailsService extends RemoteService {

        public Boolean initService();

        public String tranformGrails2Sbvr(ServentSettings serventSettings, String sbvr)
                throws ServiceException;

        public String deploy(ServentSettings serventSettings, String grailsFileName,
                String appName)
                throws ServiceException;
}
```

2.3.6 Workflow Generation using XPDL

This service integrates the workflow generation capabilities described in the deliverables 2.2 [1], 2.3 [3], 2.4 [4] and 2.5 [5]. The service takes the SBVR vocabulary and SBVR rules as input and returns the generated workflow as XPDL which can be used with a workflow engine. The interface can be seen in Listing 2.7. The integration of this component is described in detail in Chapter 4.

Listing 2.7: Interface of the XPDL generation service

```
public interface XPDLService extends RemoteService  {

    // Initialise the Service Runtime
    public Boolean initService();

    // Transform the input vocabulary and input rules to XPDL Workflow
    public String tranformSbvr2Xpdl(String sbvrVocabular, String sbvrRules)
        throws ServiceException;
}
```

This section provided an overview about the different integrated components in Sepiax-Web from an interface point of view.

2.4 Sepiax-Web User Interface and Widgets

As the previous sections focused on the back-end services, this section focuses on the user interface. Basically the user interface of Sepiax-Web consists of a number of individual widgets created for Sepiax-Web. These widgets then are aligned to expose the desired functionality to the user.

Currently there are two different user interfaces composed using these widgets: (i) the original Sepiax-Web interface described in this section together with the individual widgets and (ii) the meta-data editor interface created for the integration into Guigoh, which will be explained in detail in Chapter 3.

Figure 2.3 shows an example screen shot of the Sepiax-Web editor. Here the application is running in GWT hosted mode, indicated by the GWT logo on the upper right corner of the web browser. Additionally to the common web browser controls, there is a *Compile/Browse* button, which triggers the compilation of all source code to JavaScript, which then is opened with the system's default web browser. This can be used to compare the behavior of both environments, but is not usable for generating code for production environments. The latter should be done with build tools similar to the described approach in Section 2.5.

On the interface depicted in Figure 2.3, there are two different tab panels visible. The upper tab-panel contains the major components integrated into the Sepiax-Web editor:

1. The *SBVR editor* to organize, manage and edit the individual entries-collections of SBVR vocabulary.

2. The *XPDL generator* to generate workflows out of the stored entries.

3. The *Grails generator*, to generate and run Grails applications using the stored entries.

4. The *Servent setup* panel is getting obsolete, as the last services are being migrated away from the Servent.

The currently selected tab is used to manage and edit entries in the repository.

2.4.1 SBVR Editor

In the case of Figure 2.3, a filter in the title for all entries matching *Car* is active. As can be seen there are two different entries matching this filter condition in the example. Left to the individual entries there is a list of actions that can be performed. By clicking on the details-link, the statements are shown below in this screen. By clicking on the delete-link, the entry is removed from the repository. The use-link in the middle opens the dialog containing all the entry-sinks or EntryReceivers for the individual statements, as already explained in Section 2.2.2.

In this case there is the possibility to *send* the selected entry to the editor, the Grails generator and the XPDL generator. Other components can be integrated at that point very easily. The last button in this dialog simply closes it without any effect. Right to these links, the title of the entry and the previously mentioned UUID can be seen.

Figure 2.4 shows the SBVR statement editor widget used to enter and modify SBVR statements. In this example the entry of a SBVR vocabulary statement is shown. The first editable field in this dialog is the *Namespace* field which can be used to add statements with different workspaces within an entry.

The next text field is used to enter a natural language expression. By clicking on the *Process* button the natural language parser service described in Section 2.3.1 is executed. By using the parser the editor automatically fills in the form fields below. In the *SBVR Type* combo box, the corresponding SBVR type is selected (vocabulary, rule or unknown). If the parsing is unsuccessful, or parts of the sentence are not interpreted correctly, the values can be changed any time and adapted to the user's needs. As mentioned before, such an editor can include user assistence functionality. As an example the WordNet lookup service is included as can be seen in Figure 2.2.

2.4.2 Grails Generator

Entries can be sent to the Grails generator tab. This widget only consists of an area where the current entry is shown and a button for starting the generation. Details about the generation were already discussed in Section 2.3.5.

2.4.3 XPDL Generator

The XPDL generator widget is integrated via the entry-sink mechanism as well. Entries sent to this widget are split into SBVR vocabulary and SBVR rules, as the

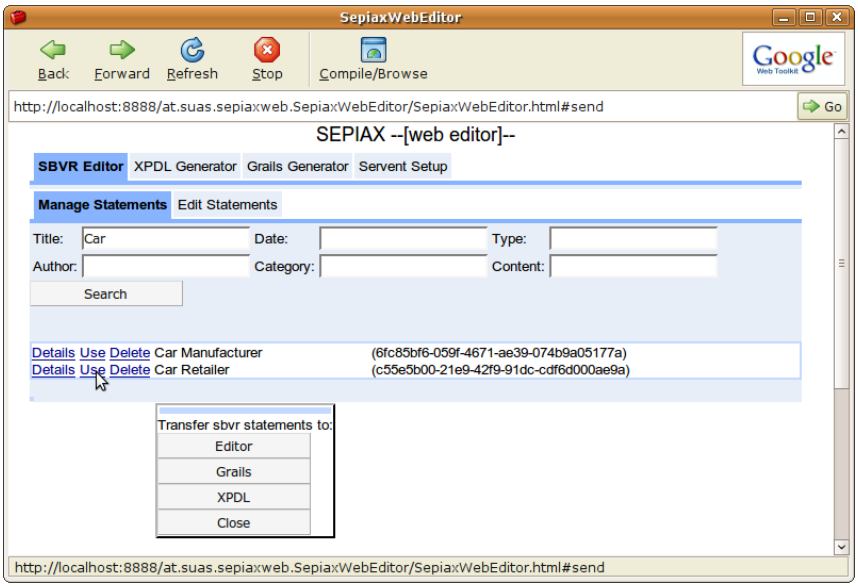

Figure 2.3: Sepiax-Web statement Manager

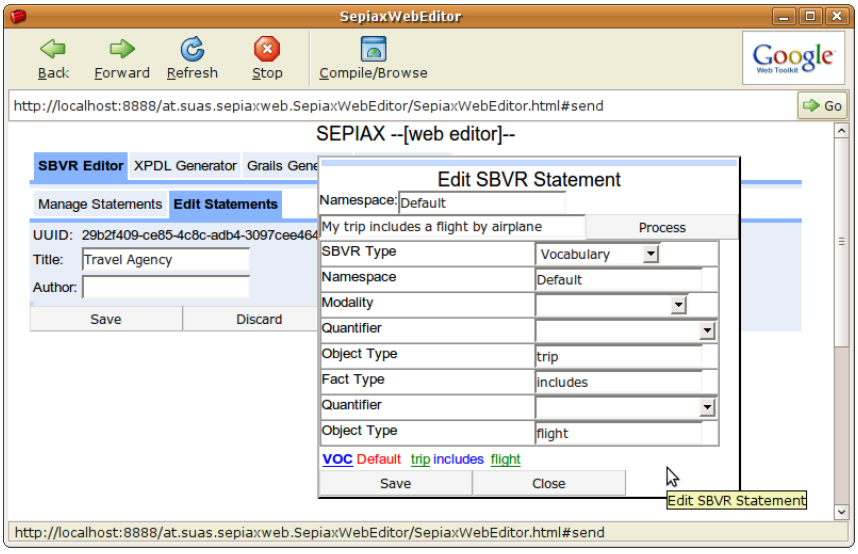

Figure 2.4: Sepiax-Web statement editor

workflow generator needs these input data separately. Additionally the rules cannot be fully entered using the Sepiax-Web editor yet. Therefore, the rules have to be entered manually. The workflow generation process is started using the *Generate XPDL* button. The integration of the workflow component is described in more detail in Chapter 4.

2.5 Sepiax-Web Build and Deployment

With the integration of Sepiax-Web into Guigoh, the build process was migrated to Apache Ant [14]. Because of GWT, the build process is different from other Java applications: The first step is compiling the server-side code. However, the second step is more complicated, as the GWT compiler has to be invoked. In this step the client-side Java code is compiled to JavaScript code. The final step is to create a package containing:

- Compiled server-side class files.
- Static web content including the GWT compiled JavaScript classes.
- All dependencies, which means all used libraries as .jar files.

The output of the build process is a WAR file that can be deployed on any servlet container[1]. As mentioned before, this build process was introduced for the Guigoh integration which will be described further in Chapter 3. As the GWT compiler needs the entry point as a parameter, the Guigoh-Version of Sepiax-Web is explicitly chosen in the build-file. This can be easily changed by altering the arguments of the `compile-gwt-clientcode` build target.

[1] Tested with Java 6 and Apache Tomcat Version 6

3 Integration of Sepiax-Web into Guigoh

Guigoh, the Open Knowledge Space (OKS) implementation of IPTI, is a social networking and collaboration platform based on Ajax technology. The work described in this chapter was done during the research exchange from SUAS in February 2009. In the following the integration of Sepiax-Web and Guigoh is described. To describe the user view point, an integration Use-Case is given in Section 3.1. The technical integration is described in the following Section 3.2.

3.1 Guigoh Integration Use-Case

One functionality of Guigoh is to create, edit and share documents within the community. The search capabilities at the current stage, however, are limited. In order to provide an alternative way of indexing of stored documents, the integration of Sepiax-Web was initiated. Both, the Guigoh application and the Sepiax-Web application are installed individually in order to minimize the needed integration effort. For each document, the user has the opportunity to add it to the Sepiax-Web index by clicking on a link in Guigoh. With this step the use case is initiated.

As the needs of the Sepiax-Web editor are different from the needs of Guigoh users, a special user interface for this integration was created by assembling the available widgets in a different way. This new form, titled as *Knowledge Resource editor* is depicted in Figure 3.1.

As already mentioned before, a new dedicated Sepiax-Web interface for the Guigoh integration was developed. When opened, the interface depicted in Figure 3.1 is presented to the user. As can be seen in the screenshot, the only parameter, this editor needs is the unique *document id*. This document id is then used to retrieve more information about the document by using the interface already mentioned in

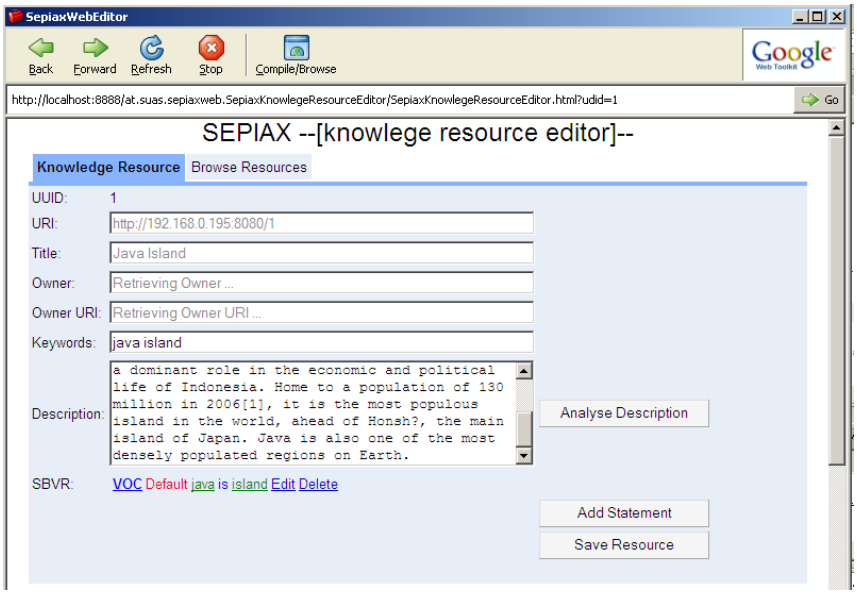

Figure 3.1: Sepiax Knowledge Resource editor

2.1.3.

When the interface startup is successful, all fields (except for the SBVR area) are already filled with data provided by Guigoh. Keywords, description and SBVR data can be altered. All other fields are read only as it is information under Guigoh's authority.

Next to the description area is the *Analyse Description* button. By pressing this button the sentences in the description area are split into individual chunks and analyzed by the Stanford parser described in Section 2.3.1. The parsed statements then are shown to the user in the SBVR area below of the description area.

The SBVR area is the SBVR editor widget explained in Section 2.4.1. The results of the description analysis can be edited at that point. Sentences that could not be parsed are marked as erroneous. The user can either open the statement and try to correct it (if possible) or delete the statement if necessary. Additionally the user can add new statements by using the corresponding *Add statement* button.

Finally the *Save* button triggers storing the new entry in the repository. As already explained in Section 2.3.3, the Lucene index will be updated at the same time. This index is consequently used to lookup the document IDs stored in the repository.

As can be seen in Figure 3.1, the Knowledge Resource editor has an own tab

to search for documents in the index. As described in the Sections 2.3.2 and 2.3.3, there is a preliminary solution for the repository and the search engine. Currently the Lucene full-text search engine is used to lookup documents. This means that there is no native SBVR support in the query engine.

3.2 Technical Integration

Both applications, Guigoh and Sepiax-Web, need a standard servlet container to run. And because of being web-based, the integration can be easily achieved by using referencing mechanisms.

As explained before, the Sepiax Knowledge Resource editor is started out of the Guigoh document browser. The only parameter sent with this request is the document ID of the document to tag. By using this document ID, the editor retrieves all necessary meta data from Guigoh by using Guigoh's REST interface. The following REST calls are used:

- **guigohRestGetDocumentInfo** this call takes the document ID as a parameter and returns the documents meta data: owner ID, title, description and tags.
- **guigohRestGetProfileInfo** using the owner ID, returned by the previous call, this REST call provides additional information about the owner of the document.
- **guigohOpenDocument** opens finally the document and is used to redirect the user to the suggested documents.

The order of the calls can be seen in the Sequence Diagram depicted in Figure 3.2.

The REST interface is protected by an authentication token. This guarantees that only users logged-in into Guigoh can use the REST interface and consequently the integrated Knowledge Resource editor. This prevents users from accessing documents that they are not allowed to access.

All the REST calls are wrapped into a GWT service. The service interface can be seen in Listing 3.1. The parameter *at* in the service calls is the previously mentioned authentication token needed for the REST call. The classes *GuigohUserProfile* and *GuigohDocumentInfo* are wrapper classes for returning the information. The Guigoh

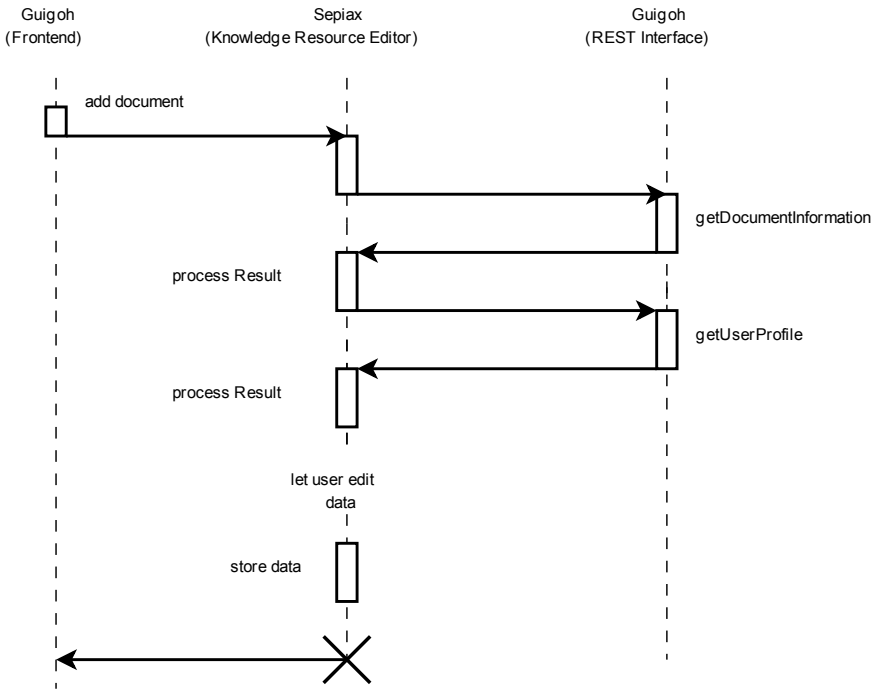

Figure 3.2: Guigoh REST Calls

REST interface serializes the response as JSON, which can easily be parsed by using openly available libraries available at the official homepage [15].

Listing 3.1: Guigoh Interface

```
public interface GuigohAdapter extends RemoteService {

        // Initialize the Service Runtime
  public Boolean initService();

        // Retrieve the user profile from Guigoh
  public GuigohUserProfile getProfile(String uuid, String at)
        throws ServiceException;

        // Retrieve the document information from Guigoh
  public GuigohDocumentInfo getDocumentInfo(String udid, String at)
                throws ServiceException;
}
```

Summarizing it can be said that due to the nature of web applications, the integration was rather straightforward. The Knowledge Resource editor from Sepiax-Web is simply started by a linking mechanism. All other necessary calls from the editor to Guigoh are done using the REST interface provided by Guigoh. The

simplicity of this method together with authentication by using the cookie-based authentication token, offers a seamless integration of two different Ajax web applications.

4 Integration of Worklfow Generation and Execution into Sepiax-Web

As part of Sepiax-Web, the prototype documented in WP 2 Deliverables 2.2 [1], 2.3 [3], 2.4 [4] and 2.5 [5] was integrated into Sepiax-Web. The patterns of the workflow transformations were enriched and further described in [3], where it was also foreseen to integrate this extended version into Sepiax-Web with appropriate modifications to address the shortcomings of the workflows engine in supporting XPDL Version 2.0.

In the D2.3 we propose the generation of XPDL Version 2.0, but engines do not yet support this version [1]. Therefore XPDL Version 1.0 files will be generated in order to be able to execute the workflows. However, not all new patterns introduced in D2.3 can be represented in XPDL Version 1.

For executing generated workflows in XPDL, the Nova Bonita workflow engine is used [4]. The workflow runtime is part of the server side of the Sepiax-Web editor. The integration is achieved, by adding the Nova Bonita Runtime to the Sepiax-Web project. Additionally, a database is needed for persisting the workflow data. Per default an embedded HSQL-Database [16] is used, which is an open-source, pure Java SQL Database. However, for the integration a dedicated MySQL 5.0 server is used, which is accessed by the Nova Bonita Runtime in the Sepiax-Web editor as well as the Nova Bonita console. Using this approach, the XPDL data can be generated and then submitted to the workflow engine, all within the Sepiax-Web editor, while still using Nova Bonita's own approach to manage the deployed workflows.

To administer and use a workflow, the Nova Bonita console can be used, which is a web application accessing the Nova Bonita database as well. The usage of this console is further described in [5]. It is not intended to execute the workflow from within Sepiax-Web, in order to not duplicate functionality, that is already present

[1] `http://wiki.bonita.ow2.org/xwiki/bin/view/Main/Roadmap`

in the Nova Bonita console.

Integrated into Sepiax-Web is the generation of XPDL out of SBVR and the deployment of the generated XPDL workflows. The following section outlines an example use-case, which can be used within Sepiax-Web editor.

4.1 Workflow Reconfiguration Example

This example models the planning of a trip by airplane and the online tickets purchasing process by means of an intermediary agent. The BPMN description of the initial process is depicted on the left hand side of Figure 4.1. The corresponding SBVR statements can be seen in Listing 4.1 and Listing 4.2.

Listing 4.1: Vocabulary

```
traveller
agent
airline
trip
ticket-order
order
ticket
trip-solution
seat
credit-card
flight
confirmation
itinerary
eTicket
traveller plans trip
traveller submits order
traveller receives itinerary
traveller receives eTicket
agent receives order
agent searches trip-solution
agent orders ticket
agent receives confirmation
agent issues itinerary
airline receives ticket-order
airline reserves seat
airline charges credit-card
airline confirms flight
airline issues eTicket
```

Listing 4.2: RuleSet

```
after the traveller plans the trip then the traveller submits the order.
after the traveller submits the order then the agent receives the order.
after the agent receives the order then the agent searches the trip-solution.
after the agent searches the trip-solution then the agent orders the ticket.
```

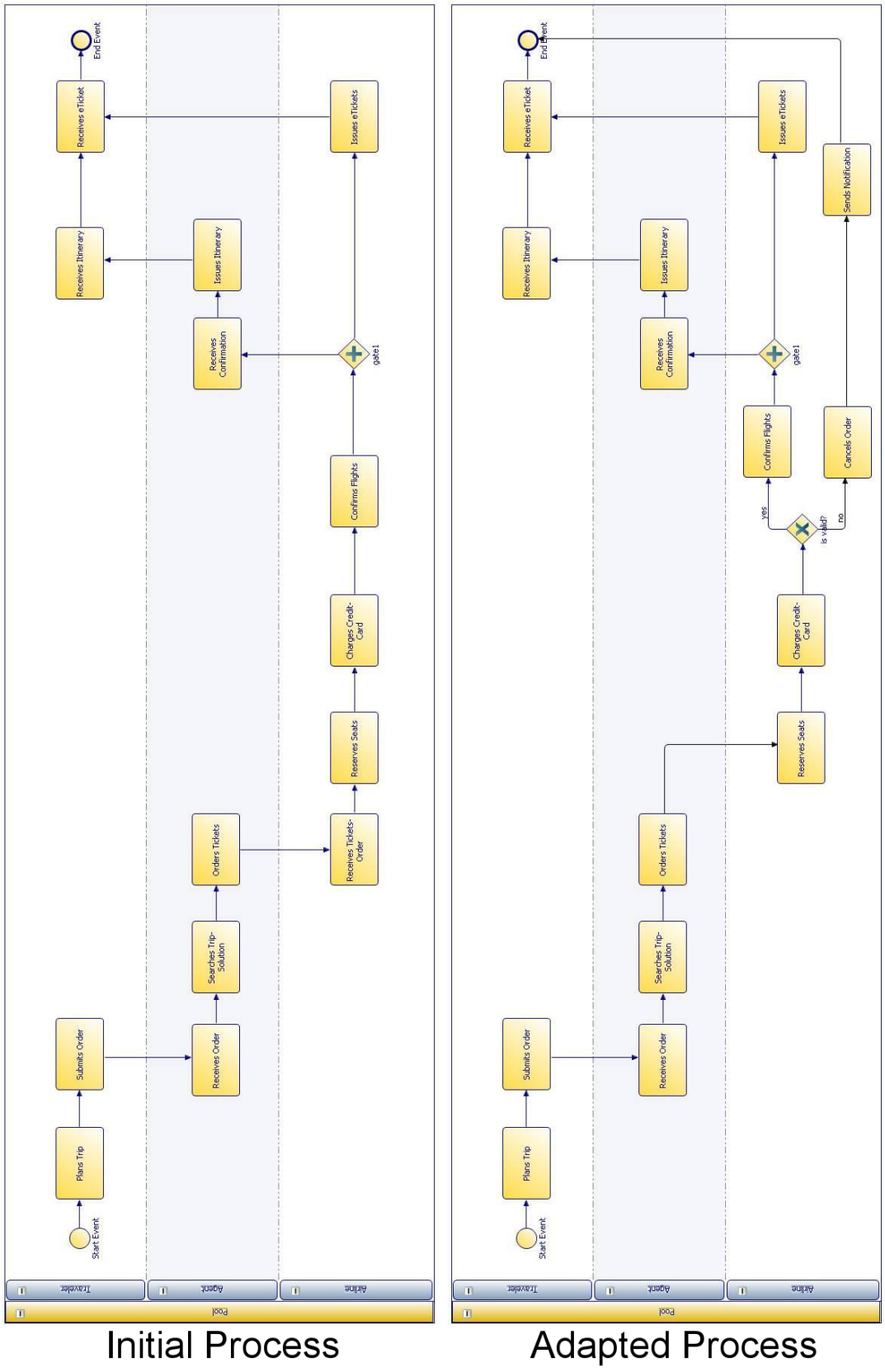

Figure 4.1: BPMN Representation

```
after the agent orders the ticket then the airline receives the ticket-order.
after the airline receives the ticket-order then the airline reserves the seat.
after the airline reserves the seat then the airline charges the credit-card.
after the airline charges the credit-card then the airline confirms the flight.
after the airline confirms the flight then the agent receives the confirmation
  and the airline issues the eTicket.
after the agent receives the confirmation then the agent issues the itinerary.
after the agent issues the itinerary then the traveller receives the itinerary.
after the traveller receives the itinerary then the traveller receives the eTicket.
after the airline issues the eTicket then the traveller receive the eTicket.
```

In a real-world scenario this workflow would stay life for a while in a workflow engine. During that period an undefined number of individual workflows would be executed following the deployed workflow template. After a while the process administrator decides to change the process model in order to obtain the situation described in the BPMN representation on the right hand side of Figure 4.1. The same process is represented in SBVR as shown in Listing 4.3 and Listing 4.4.

Listing 4.3: Vocabulary

```
traveller
agent
airline
trip
ticket
order
trip-solution
seat
credit-card
flight
confirmation
notification
itinerary
eTicket
credit-card is valid
airline sends notification
traveller plans trip
traveller submits order
traveller receives itinerary
traveller receives eTicket
agent receives order
agent searches trip-solution
agent orders ticket
agent receives confirmation
agent issues itinerary
airline receives order
airline reserves seat
airline charges credit-card
airline confirms flight
airline cancels order
airline issues eTicket
```

Listing 4.4: RuleSet

```
after the traveller plans the trip then the traveller submit the order.
after the traveller submit the order then the agent receives the order.
after the agent receives the order then the agent searches the trip-solution.
after the agent searches the trip-solution then the agent orders the ticket.
after the agent orders the ticket then the airline reserves the seat.
after the airline reserves the seat then the airline charges the credit-card.
after the airline confirms the flight then the agent receives the confirmation
  and the airline issues the eTicket.
after the agent receives the confirmation then the agent issues the itinerary.
after the agent issues the itinerary then the traveller receives the itinerary.
after the traveller receives the itinerary then the traveller receives the eTicket.
after the airline issues the eTicket then the traveller receive the eTicket.
after the airline charges the credit-card if the credit-card is valid then
  the airline confirms the flight.
after the airline charges the credit-card if the credit-card is not valid then
  the airline cancels the order.
after the airline cancels the order then the airline sends the notification.
```

Here there are several changes in respect to the previous version:

Activity *Receives ticket-order* was deleted.

Activity *Cancels order* was added in parallel with **Activity** *Confirms flight.*

Activity S*ends notification* was added.

These changes are obtained with these modifications in the SBVR representation:

Term *Ticket-order* Was deleted from vocabulary.

Term *Notification* Was added to the vocabulary.

Fact-type *Airline cancels order* was added to the vocabulary.

Fact-type *Airline sends notification* was added to the vocabulary.

Fact-type *Airlins receives ticket-order* was deleted from the vocabulary.

Rule *after the agent orders the ticket then the airline receives the ticket-order* was deleted from the previous ruleset.

Rule *after the airline receives the ticket-order then the airline reserves the seat.* was deleted from the previous ruleset.

Rule *after the agent orders the ticket then the airline reserves the seat.* was added to the ruleset.

Rule *after the airline charges the credit-card then the airline confirms the flight.* was deleted from the ruleset.

Rule *after the airline charges the credit-card if the credit-card is valid then the airline confirms the flight.* was added to the ruleset.

Rule *after the airline charges the credit-card if the credit-card is not valid then the airline cancels the order* was added to the ruleset.

Rule *after the airline cancels the order then the airline sends the notification.* was added to the ruleset.

In Figure 4.2, the input dialog for the SBVR vocabulary and rules can be seen. The name and the version of the process can be entered in this user interface, which will be included in the generated XPDL. By using the *Generate XPDL* button, the SBVR-to-workflow service is invoked and the resulting XPDL data can be checked on the *XPDL Output* Screen, which is depicted in Figure 4.3.

After that the XPDL file is packaged and deployed at the configured workflow engine by using the *Deploy* button. If there is no process with the entered name, the new process is simply deployed on the workflow engine. If already a process with that name exists, two different scenarios are possible:

(1) The process can be deployed using a new version number, which has no effect on existing workflow instances and therefore has no further implications. (2) If, however, the version number of the existing workflow is the same as the new version, a compatibility check is performed, whether the running workflow instances can be migrated or not (see WP2 deliverables). Figure 4.4 shows an example output of that check.

At the moment it is possible to undeploy the old process and to deploy the new one, by clicking the *Migrate* button, or to keep alive the old process and discard changes, by clicking the *Cancel* button. As soon as the used workflow engine will support on-the-fly reconfiguration of process models, instances for which the admissibility for the reconfiguration is permitted will be migrated without any loss of data to the new process model and the old will be kept running according to the old model.

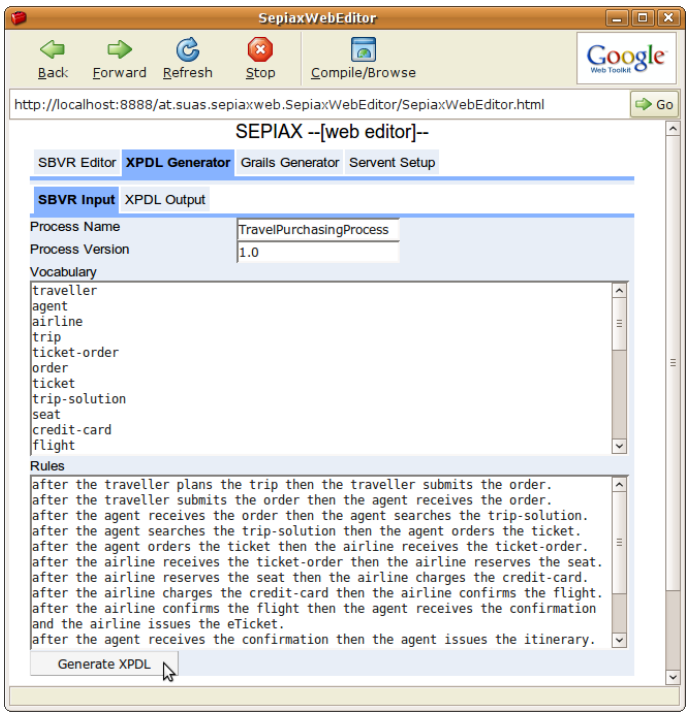

Figure 4.2: SBVR to XPDL Input

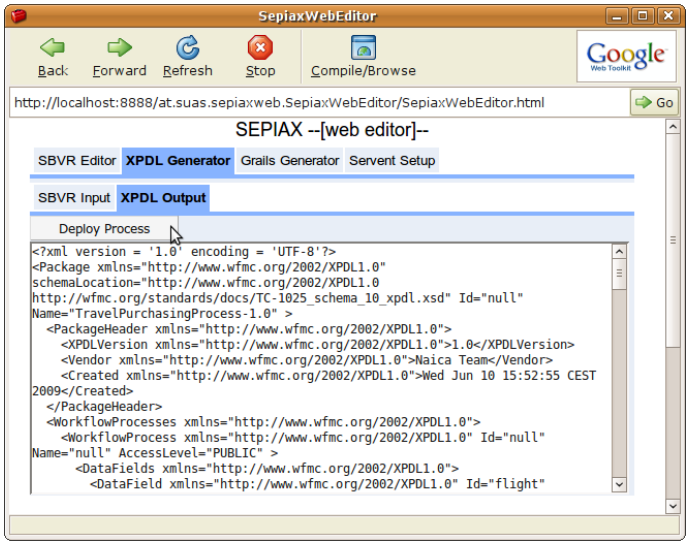

Figure 4.3: SBVR to XPDL Input

Reconfiguration analysis results upon Process Instances

Instance Name	Admissibility
ETicket-1.0-ETicket-1.0$1	Yes
ETicket-1.0-ETicket-1.0$2	Yes
ETicket-1.0-ETicket-1.0$3	Yes

Figure 4.4: Example Compatibility-Check

5 Conclusion

This report describes the integration of the prototypical parts developed in Workpackage 2 into one integrated Ajax application. This application started as a web based frontend to enter SBVR data, and evolved to the melting-pot of all components individually prototyped by different partners within the consortium. These components were roughly described, linking to the dedicated deliverables where more fine grained information can be found.

Additionally, the design of the Sepiax-Web editor was outlined and an introduction into the leveraged Ajax framework, GWT, was given. This report also introduces the extension mechanisms, in order to provide a quick-start for other partners to integrate own components. Furthermore, the build process is described, enabling the reader to not only download the existing source code onto an own development environment, but also easily build an own package to be deployed on a servlet container.

By describing the Guigoh integration, not only the output of a researcher exchange between SUAS and IPTI was described, but also how Sepiax-Web can be integrated with other web applications, without sharing the same code base. This is achieved by the natural web approach: by interlinking web pages and invoking REST-based web services.

The final chapter introduces the current state of the integration of the SBVR to Workflow generation service provided by partner T6-Eco. This integration is outlined by providing a use-case that enables the reader to understand the purpose and use of this integrated component.

Sepiax-Web is now ready to be downloaded by early adopters, developers and researchers interested in the work performed within WP2 of the OPAALS research project. It can be used to try the individual components step-by-step by using an integrated application with clearly defined interfaces, ready to be used for further integration of related components in this research area. Therefore, it provides a con-

venient starting point for SBVR- and workflow-related research in the community.

All code and more information regarding the build process can be found on the homepage of the Sepiax initiative: `http://www.sepiax.org`. Further activities concerning SBVR will continue in workpackage 3.

Bibliography

[1] R. Eder, T. Kurz, T. J. Heistracher, V. Bayon, A. Filieri, M. Russo, Prabhakar TV, H. Peukert, A. Marinos, and S. Hendryx. D2.2 - Automatic code structure and workflow generation from natural language models . OPAALS Project, April 2008.

[2] OMG. *Semantics of Business Vocabulary and Business Rules Specification (SBVR)*, March 2006. Interim Convenience Document, `http://www.omg.org/docs/dtc/06-03-02.pdf`. Last accessed on 04/05/2007.

[3] A. Filieri, A. Margarito, T. Kurz, and R. Eder. D2.3 - Extended vocabulary and rule set for an existing scenario. OPAALS Project, November 2008.

[4] A. Filieri and A. Margarito. D2.4 - Prototypical implementation of dynamic workflow reconfiguration. OPAALS Project, March 2009.

[5] A. Filieri, A. Margarito, T. Kurz, and R. Eder. D2.5 - Extended automated code/workflow generation example for one real-world application case. OPAALS Project, June 2009.

[6] Google Inc. Google Web Toolkit, 2006. `http://code.google.com/webtoolkit/`, Last accessed 01/05/2009.

[7] The Stanford Natural Language Processing Group. The Stanford Parser, 2002. `http://nlp.stanford.edu/software/lex-parser.shtml`, Last accessed 01/05/2009.

[8] R. Eder, T. Kurz, T. J. Heistracher, V. Bayon, M. Russo, and A. Filieri. D2.1 - Design of Software Generation Prototype. OPAALS Project, October 2007.

[9] The Apache Software Foundation. Apache Jackrabbit. `http://jackrabbit.apache.org/`, Last accessed 01/05/2009.

[10] Day Management AG. JSR 170: Content Repository for Java technology API, 2006. http://jcp.org/en/jsr/detail?id=170, Last accessed 01/05/2009.

[11] The Apache Software Foundation. Apache Lucene. http://lucene.apache.org, Last accessed 01/05/2009.

[12] Princeton University. WordNet, 2006. http://wordnet.princeton.edu/, Last accessed 01/05/2009.

[13] Brett Spell. Java API for WordNet Searching (JAWS), 2008. http://lyle.smu.edu/~tspell/jaws/, Last accessed 01/05/2009.

[14] The Apache Software Foundation. Apache Ant. http://ant.apache.org/, Last accessed 01/05/2009.

[15] JavaScript Object Notation (JSON), 2006. http://www.json.org/, Last accessed 01/05/2009.

[16] Hypersonic SQL Group. Hypersonic SQL Database (HSQLDB). http://www.hsqldb.org/, Last accessed 01/05/2009.