SCRIPT ENGINE USE GUIDE

May 2016

Version 5

indra

1 INDEX

2 INTRODUCTION

2.1 Requirements

Before Reading this guide, we suggest you to read **SOFIA2-Concepts SOFIA2.doc**

2.2 Goals and scope of the current document

The current document describes how to use SOFIA2's script engine, which allows the platform users to create their own scripts to be executed either when an ontology instance arrives or after a specified time interval.

2.3 Concepts

- **Script types:** The platform supports 2 script types:
  - **Event-reacting Scripts:** Allows to execute code when an ontology instance arrives.
  - **Temporized Scripts:** Allows to execute code after a time interval. This interval can be specified with a cron.
- **Scripting language:** The scripts are defined in Groovy language (**http://groovy.codehaus.org**).
- **Available APIs for the scripts (Operations):** The scripts can access a set of APIs which allow for the execution of actions such as:
  - Send an e-mail.
  - Invoke a URL.
  - Insert an Ontology instance.
  - Query for Ontology instances.
  - Delete an Ontology instance.

  Users with privileges can create their own APIs.
- **Security in scripts:** The scripts are executed in a secure environment, thus guaranteeing that an error in one script cannot affect the other scripts. Besides, their invokation has a time-out paramter. The scripts can only invoke a set of operations.

3 SCRIPT DEVELOPMENT

3.1 Groovy

The scripts are developed in Groovy language, which can be programmed using a a non-typed, easy syntax.

Groovy is compiled in a Java execution time and is executed in the JVM. From it, all the Java classes and libraries can be accessed.

You can find more information on Groovy at **http://groovy.codehaus.org**.

3.2 API Operations

The **Operations** allow us to define the APIs that the script can access. They are the commands that our Groovy Scripts can execute.

Only the Administrator users can define operations.

This is an example of API:

```
import com.indra.jee.arq.spring.sofia2.script.api.Api;

public class ApiGroovy {

	public class ScriptException extends Throwable{

		public ScriptException(Exception e){
			super(e);
		}

		public ScriptException(Exception e, String message){
			super(message, e);
		}

		public ScriptException(String message){
			super(message);
		}
	}

	private Api api;

	public ApiGroovy(){
		api = new Api();
	}

	public String insert(String ontologyName, String ontology) throws ScriptException{
		try{
			return api.insert(ontologyName, ontology);
		}catch(Exception e){
			throw new ScriptException(e);
		}
	}

	public String rolback(String ontologyName, String id) throws ScriptException{
		try{
			return api.rolback(ontologyName, id);
		}catch(Exception e){
			throw new ScriptException(e);
		}
	}
```

This API is distributed with the SOFIA2 platform:

- It does not have any package, because the Scripts cannot import Classes. Our operations must be hosted in the base package.
- It performs a Class's import. Operations do not have restrictions and, by using Groovy, integration with Java is 100%

```
import com.indra.jee.arq.spring.sofia2.script.api.Api;
```

- It can define classes and subclasses, and it must define methods exposing the operations we want to use.

```
public class ApiGroovy {
	public class ScriptException extends Throwable{
		public ScriptException(Exception e){
			super(e);
		}
```

3.2.1 ApiGroovy

The `ApiGroovy` operation is a facade to the Java API class offering those methos:

- **Insert** inserts an ontology instance in the real time data base.
- **Rolback** deletes an ontology from the real time data base.
- **getAtribute** provides the value of one attribute from one ontology.
- **sendMail** allows to send an e-mail.

```
import com.indra.jee.arq.spring.sofia2.script.api.Api;
public class ApiGroovy {
	public class ScriptException extends Throwable{
		public ScriptException(Exception e){
			super(e);
		}
		public ScriptException(Exception e, String message){
			super(message, e);
		}
		public ScriptException(String message){
			super(message);
		}
	}
	private Api api;
	public ApiGroovy(){
		api = new Api();
	}
	public String insert(String ontologyName, String ontology) throws
ScriptException{
		try{
			return api.insert(ontologyName, ontology);
		}catch(Exception e){
			throw new ScriptException(e);
		}
	}
	public String rolback(String ontologyName, String id) throws
ScriptException{
		try{
```

```
                return api.rolback(ontologyName, id);
            }catch(Exception e){
                throw new ScriptException(e);
            }
      }
      public String getAtribute(String ontology, String atribute) throws
ScriptException{
            try{
                return api.getAtribute(ontology, atribute);
            }catch(Exception e){
                throw new ScriptException(e);
            }

      }
      public void sendMail(String to, String subject, String msg) throws
ScriptException{
            try{
                api.sendMail(to, subject, msg);
            }catch(Exception e){
                throw new ScriptException(e);
            }
      }
}
```

3.3 Platform's Variables

In execution time, the Scripting engine provides a number of data so that the scripts can use those:

`ontology`	Data on the inserted or updated ontology
`ontologyName`	Name of the inserted or updated ontology
`ontologyId`	List with the identifier of the instance of the inserted ontology or the updated ontologies
`typeMessage`	Type of operation that has triggered the INSERT/UPDATE script
`sessionKey`	sessionKey that made the insert or update

These variables can be accessed by event-reacting scripts, but not by temporized scripts.

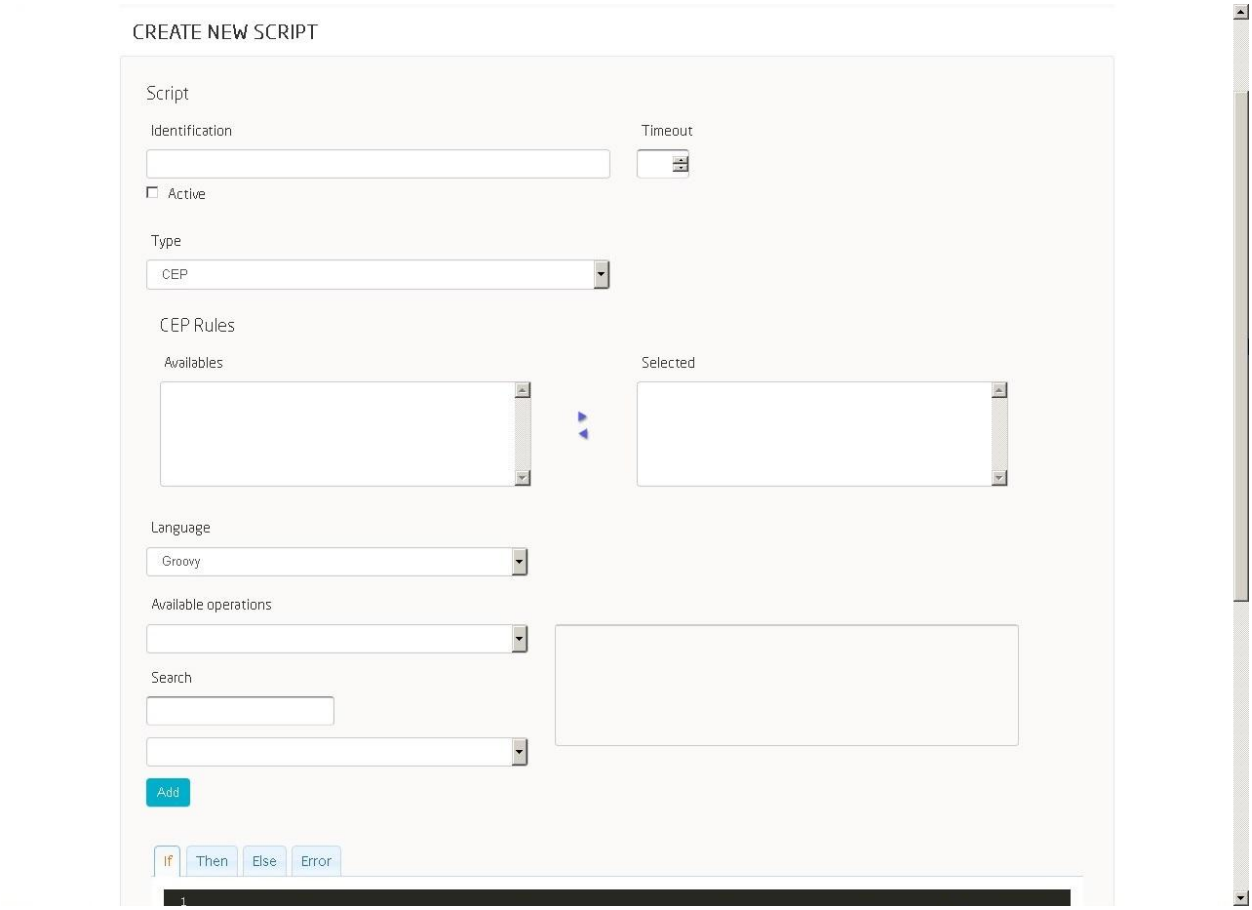
`scriptName`	Name of the script that has been launched.

This variable can be accessed by any script.

`Error`	Generated exceptions

This variable only has value in scripts from the Error block.

3.4 Creation of a Script



When creating a script, we must provide this information:

- **Identification**: Unique value to identify the script.
- **Ontology:** Ontology that will trigger the script's execution.
- **Active**: Field to activate and deactivate the script.
- **Timer:** Specifies the time interval before launching the script (in a temporized script).
- **Timeout:** Maximum time interval that the script is allowed

Depending on the input parameters, a Script becomes a:

- **Event-reacting Script:** if Timer is selected.
- **Temporized Script:** if Ontology is selected.

We will now see an example of a script that is triggered on the insertion or update of the ontology **SensorHumedad**.

- The **If block**  , mandatory in scripts associated to ontologies, defines a condition that returns a true or false value and that will trigger either the **then** block or the **else** block.

```
api = new ApiGroovy();
texto = api.getAtribute(ontology, "SensorHumedad.identificador");
if (texto=="ST-TA114"){
    return true;
}else{
    return false;
}
```

In the previous example, the first thing we do is initializing one of the previously defined operations.

```
api = new ApiGroovy();
```

Then we use a variable to store the value of the attribute **SensorHumedad.identificador** (identifier) of the inserted ontology.

```
texto = api.getAtribute(ontology, "SensorHumedad.identificador");
```

We could also have previously evaluated whether the operation is an insert or an update.

```
if (typeMessage=="INSERT"){
```

Lastly, once we have the values on which we want to make the if block's evaluation, we execute it.

```
if (texto=="ST-TA114"){
    return true;
}else{
    return false;
}
```

Depending on the previous block's evaluation, we will execute the block Then or the block else if there is (It is optional in temporized scripts)

- The **then block** is executed when the **If**'s condition is **true**

```
api = new ApiGroovy();
try{
    borrado = api.rolback(ontologyName, ontologyId[0]);
    creado = api.insert(ontologyName, ontology);
    texto = api.getAtribute(ontology, "SensorHumedad.identificador");
    c = new systemprinter();
    c.print("Borrado id "+borrado);
    c.print("Creado id "+creado);
    c.print("Valor del Atributo SensorHumedad.identificador "+texto);
    c.print("Operacion Lanzada "+typeMessage);
    api.sendMail("sofia2oncloud@gmail.com", "CASO THEN", ontology);
}catch(ScriptException e){
    api.sendMail("sofia2oncloud@gmail.com", "ERROR SCRIPT", e);
}

```

  - `borrado = api.rolback(ontologyName, ontologyId[0]);` We delete the ontology instance that has been inserted.
  - `creado = api.insert(ontologyName, ontology);` We insert again the ontology that we had inserted through the SSAP message.
  - `c = new systemprinter();` We make a new operation library available.
  - We handle the exceptions that can be generated so that, if an error occurs, we send an e-mail with the subject "ERROR SCRIPT" and the generated exception.

```
}catch(ScriptException e){
    api.sendMail("sofia2oncloud@gmail.com", "ERROR SCRIPT", e);
}
```

- The **ELSE block** is optional and is executed when the condition in the If block is false, if any operation has been defined for that.

```
api = new ApiGroovy();
api.sendMail("sofia2oncloud@gmail.com", "CASO ELSE", ontology);
```

  In this case, it sends an e-mail with the subject "CASO ELSE", including the ontology that has triggered that execution.

- The **ERROR block** is optional and is executed when an uncontrolled error occurs in any of the previous cases.

  In this example, it sends an e-mail explaining that an error has occurred and the specific error.

```
api = new ApiGroovy();
api.sendMail("sofia2oncloud@gmail.com", "ERROR SCRIPT", error);
```

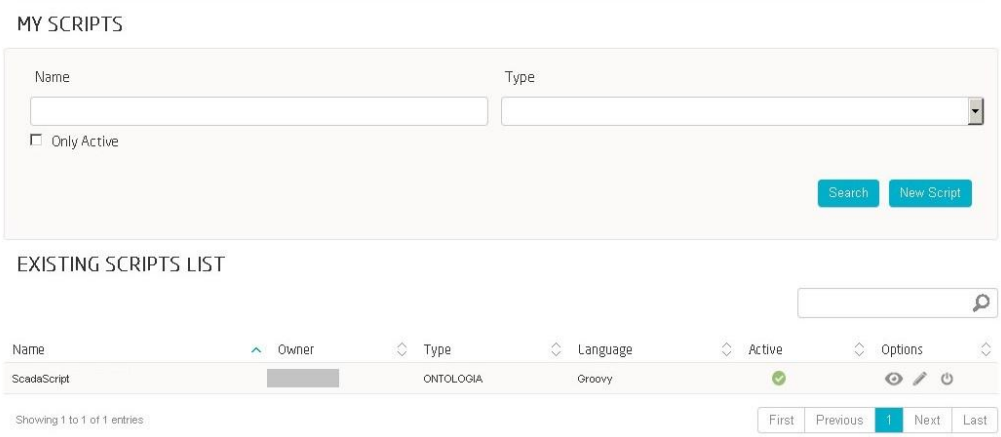
4 UI TO GENERATE PERSONALIZED SCRIPTS

SOFIA2 allows for the creation of scripts and APIs (operations) from the Web Console and from its REST API.

Depending on the user's role, we can:

- Manage APIs: Administrator user.
- Manage scripts: Administrator and collaborator users.

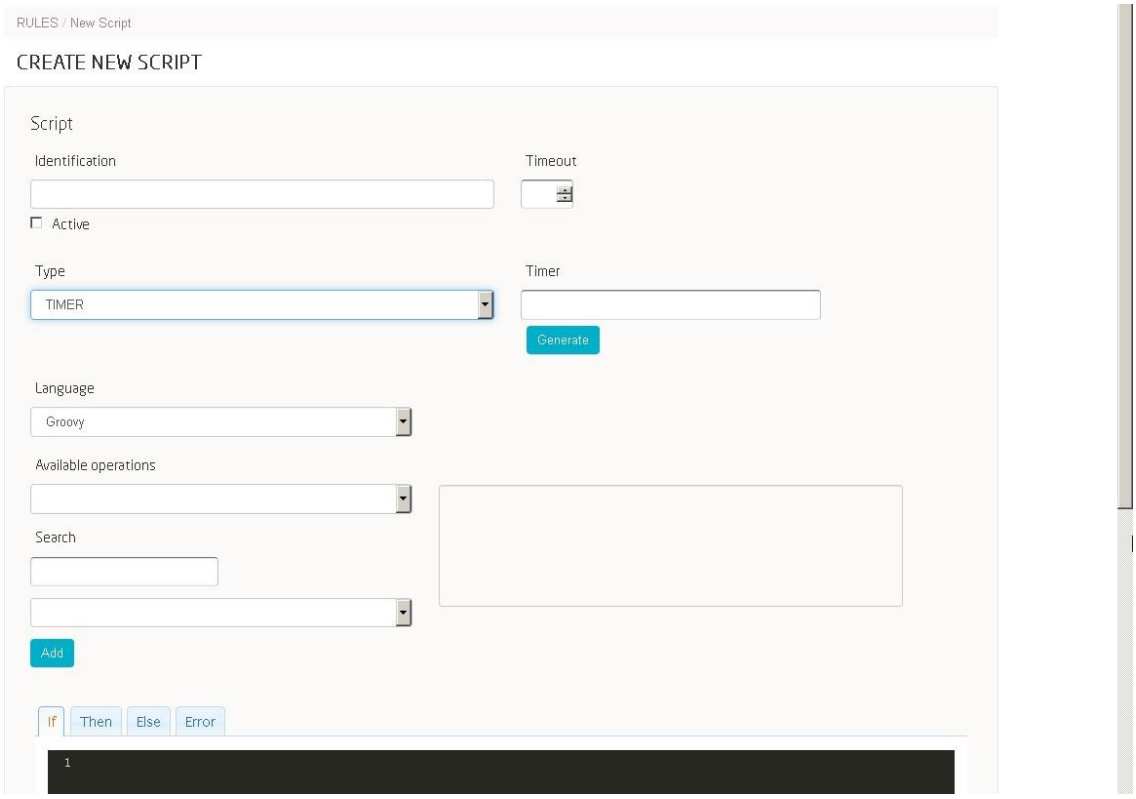
4.1 Script List



Depending on the user's role, we can access to certain sections.

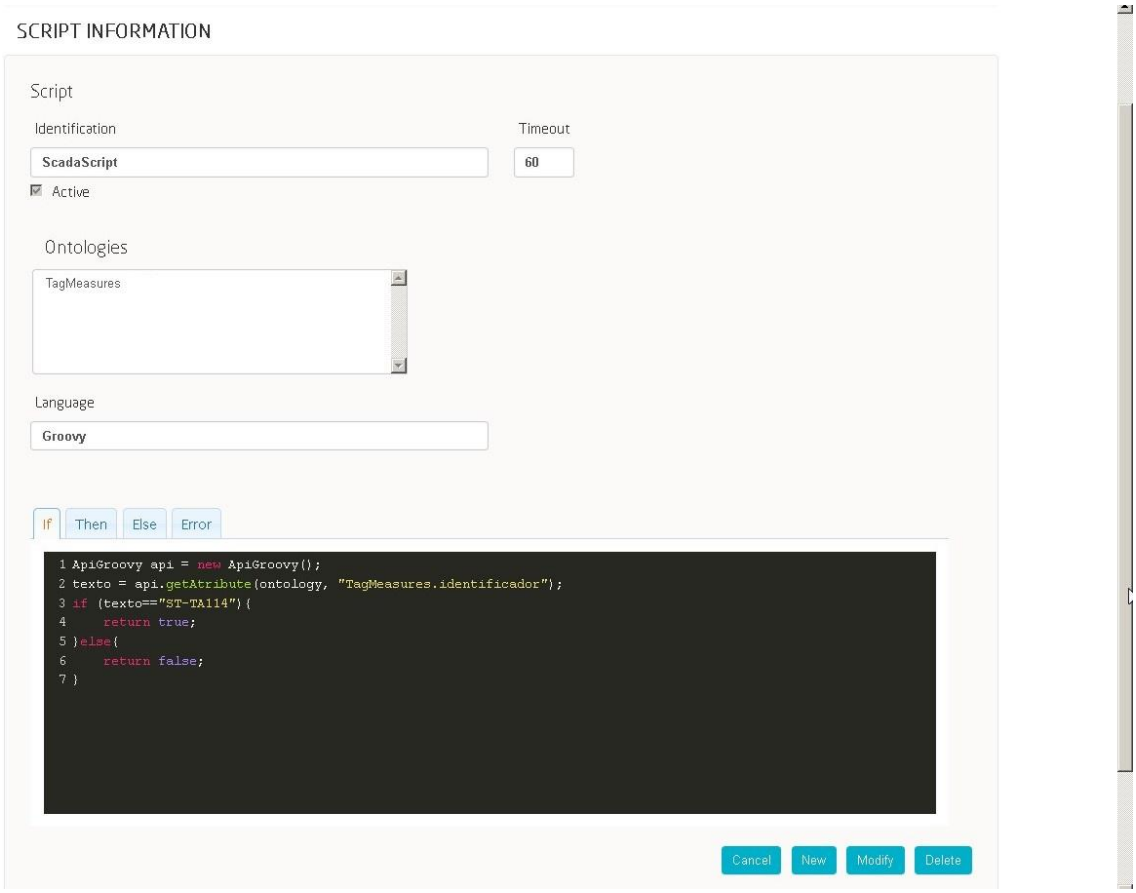
- **Temporized scripts** can only be created by Administrators.
- **Scripts associated to an ontology:**
  - An Administrator can manage these completely.
  - A Collaborator can only manage her own scripts, and these can only use ontologies on which that user has permissions.
- **Scripts associated to a parent ontology:**
  - An Administrator can manage these completely.
  - A Collaborator can only manage her own scripts, and these can only use ontologies on which that user has permissions.
- **Cep-type scripts** can be created by users with the roles Administrator and Collaborator.

4.2 Temporized Scripts



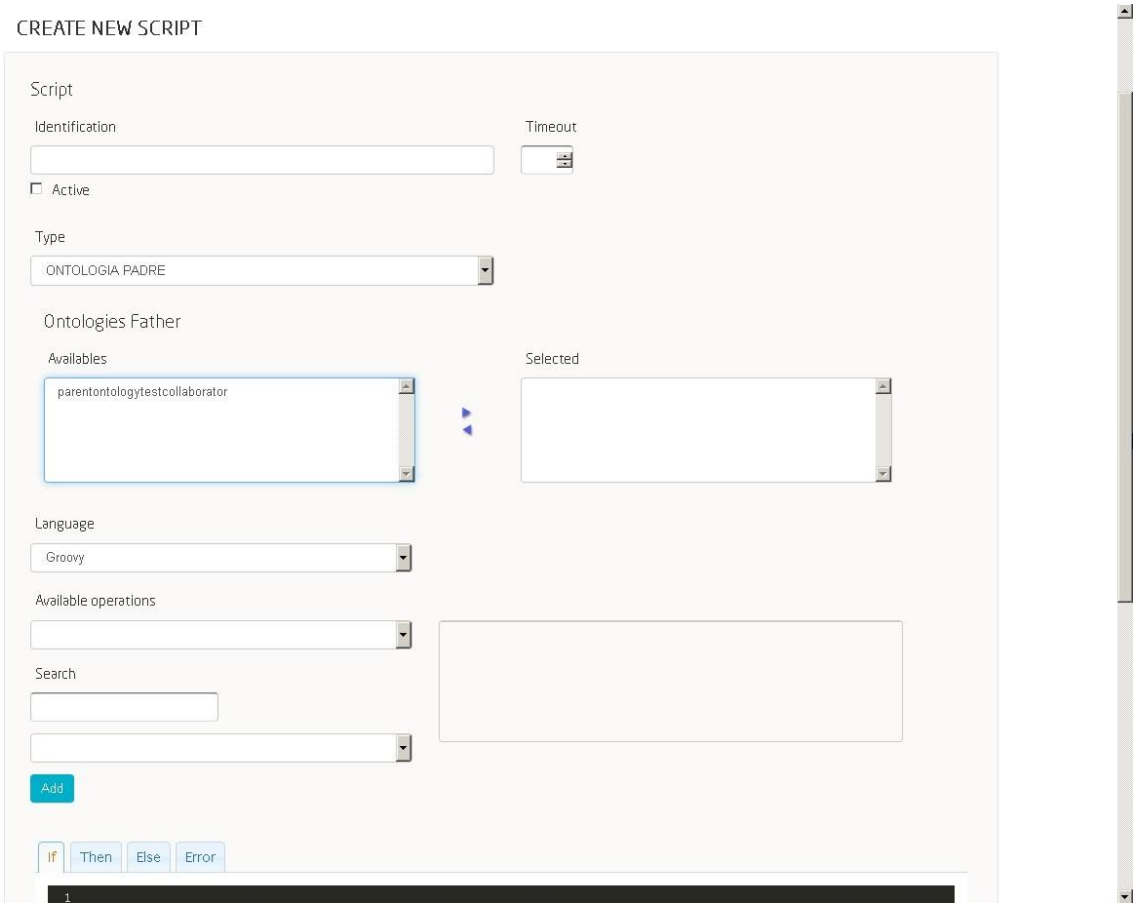
- Temporized scripts are not associated to any Ontology.
- They define the temporization type through final elements. This temporization ultimately becomes a CRON type temporization.
- The **Then** block is mandatorily defined in this type of scripts.
- The blocks **If, Else** and **Error** can be optionally defined.
- If the **If** block is defined, then every time the script is launched, depending on the value specified in the attribute **Timer,** the condition is evaluated and, depending on the value it gives, the **Then** or **Else** block will be executed. If the condition does not exist, then the **Then** block will always be executed.

4.3 Scripts associated to ontologies



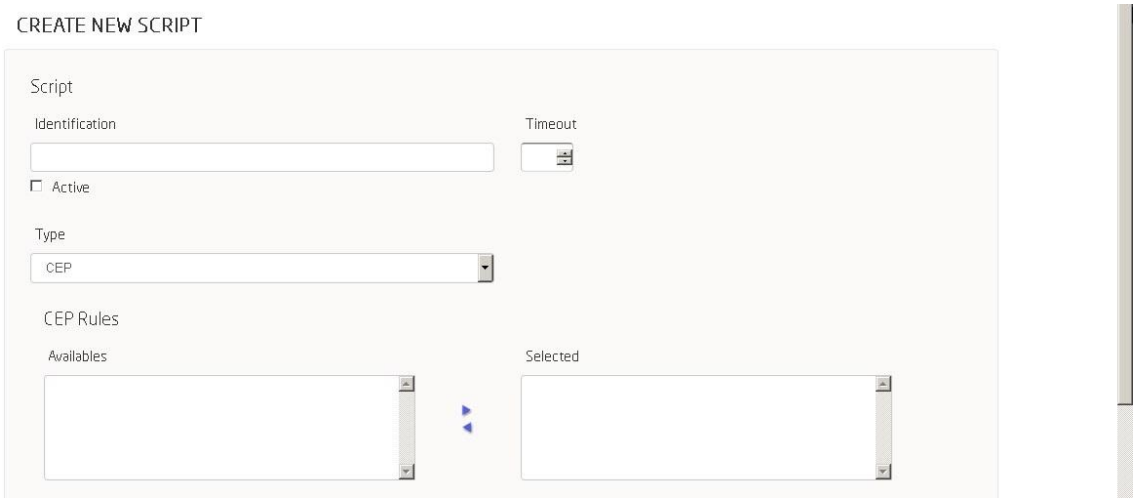
- Scripts associated to an Ontology are triggered whenever there is an insert or an update on the associated Ontology.
- The blocks **If** and **Then** are mandatorily defined in this type of script.
- The blocks **Else** and **Error** can be optionally defined.
- Whenever an insert or an update of an Ontology is made, the scripts associated to this Ontology are launched. The first step is executing the **If** block, which must return a true or false value. If the condition is fulfilled, the Then block is executed; if the condition is not fulfilled and the Else block is defined, then the Else block is executed.

4.4 Scripts associated to parent ontologies



- Scripts associated to a Parent Ontology are triggered whenever there is an insert or an update on the associated Ontology.
- The blocks **If** and **Then** are mandatorily defined in this type of script.
- The blocks **Else** and **Error** can be optionally defined.
- Whenever an insert or an update of a Parent Ontology is made, the scripts associated to this Ontology are launched. The first step is executing the **If** block, which must return a true or false value. If the condition is fulfilled, the Then block is executed; if the condition is not fulfilled and the Else block is defined, then the Else block is executed.

4.5 Cep-type scripts



- The main characteristic of the Cep-type Scripts is that they are executed every time that the event specified by the script's creator takes place.
- The blocks **If** and **Then** are mandatorily defined in this type of script.
- The blocks **Else** and **Error** can be optionally defined.
- Every time the script is executed, the condition specified in the **If** block is evaluated and then the control is returned to the script's container, which must return a true or false value. If the condition is fulfilled, the Then block is executed; if the condition is not fulfilled and the Else block is defined, then the Else block is executed.

5 SECURITY IN SCRIPT EXECUTION

The script module in the SOFIA2 platform has two security measures to prevent a script's execution to become a risk to the system's integrity.

5.1 Active Security

Based on the Java Policy (more information on this specification in the links **http://docs.oracle.com/javase/6/docs/technotes/guides/security/permissions.html** and

http://docs.oracle.com/javase/6/docs/technotes/guides/security/PolicyFiles.html).

It allows us to define the operations that the JVM will support. The JVM will be in charge of preventing the execution of any code not fulfilling the security restrictions.

This security level applies to both the scripts and the defined operations.

The definition of a TimeOut through the platform ensures that a script cannot be in execution beyond a maximum allowed time. This will cause the thread on which it is launched to be removed when the maximum execution period expires.

5.2 Passive Security

Based on a syntactic analyzer, it prevents us from using forbidden structures when defining a script.

The scripts have the following restrictions:

- Defining Classes.
- Defining Methods.
- Importing Classes.
- Making use of System.
- Making use of the following classes:

  BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter, ByteArrayInputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, Console, DataInputStream, DataOutputStream, File, FileDescriptor, FileInputStream, FileOutputStream, FilePermission, FileReader, FileWriter, FilterInputStream, FilterOutputStream, FilterReader, FilterWriter, InputStream, InputStreamReader, LineNumberInputStream, LineNumberReader, ObjectInputStream, ObjectInputStream.GetField, ObjectOutputStream, ObjectOutputStream.PutField, ObjectStreamClass, ObjectStreamField, OutputStream,

OutputStreamWriter, PipedInputStream, PipedOutputStream, PipedReader, PipedWriter, PrintStream, PrintWriter, PushbackInputStream, PushbackReader, RandomAccessFile, Reader, SequenceInputStream, SerializablePermission, StreamTokenizer, StringBufferInputStream, StringReader, StringWriter, Writer, Class, ClassLoader, Compiler, InheritableThreadLocal, Process, ProcessBuilder,ProcessBuilder.Redirect, Runtime, RuntimePermission, SecurityManager, StackTraceElement, Thread, ThreadGroup, ThreadLocal, Authenticator, CacheRequest, CacheResponse, ContentHandler, CookieHandler, CookieManager, DatagramPacket, DatagramSocket, DatagramSocketImpl, HttpCookie, HttpURLConnection, IDN, Inet4Address, Inet6Address, InetAddress, InetSocketAddress, InterfaceAddress, JarURLConnection, MulticastSocket, NetPermission, NetworkInterface, PasswordAuthentication, Proxy, ProxySelector, ResponseCache, SecureCacheResponse, ServerSocket, Socket, SocketAddress, SocketImpl, SocketPermission, StandardSocketOptions, URI, URL, URLClassLoader, URLConnection, URLDecoder, URLEncoder, URLStreamHandler.

To perform any kind of action outside the control blocks, we must invoke the defined operations that do not have these restrictions.

When creating a script through the UI, the system validates that the scripts does not fail to fulfil these validations and thus can be stored in the database for its execution.