AGSSO Component – User Manual

Contents

1 Introduction

2 Installation Steps

2.1 Prerequisites

2.2 Package Content

The software package for the AGSSO software component contains the current release of the CAS Server and a number of required libraries included in the package for the convenience of the user. The main folder and files in the distribution are:

- `conf`: folder containing various configuration files

2.3 Installing the Packages

2.4 AGSSO deployment as a service

2.5 CAS Server Configurations

2.5.1 Configuration File(revise)

3 AGSSO Administration Application

The Administrative application allows the administrator of the node where the CAS Server was installed to set properties that have a direct impact on the functionality of the CAS Server. Successful exposure and advertising of the functionality implemented by the server must follow a sequence of steps.

4 Interfaces of the AGSSO component

4.1 AGSSO Discovery interface

The AGSSO Discovery interface is compound of several Web Service operations that the remote client may call in order to obtain useful information regarding the state of the system, the list of CAS Server that are part of the computational infrastructure, the list of CASs available and their supported functionality, the status of execution for a certain workflow. While tasks are usually planned automatically for execution by AGSSO, there are situations when a user may require a certain task to be executed on a particular machine by a certain CAS. For instance, if the user requires that a certain task is to be executed by a GAP instance, the user will state explicitly this information. In this situation any CAS having its name starting with 'GAP' may be used. If in turn, the user wants a 'GAP 4.0' instance, he must specify so, and all matching CASs can be used. Every CAS installed on a CAS Server may be uniquely identified by its name and the name of the CAS Server it is installed on. So, it the CAS 'GAP 4.0 – instance1' installed on the CAS Server CS1 is targeted, the user has to specify its full identification details. In this situation the AGSSO will make sure that the task is sent to the specified CAS and no other.

The list of operations that allow the remote user to learn details regarding the capabilities of the composition infrastructure are:

- **String getAvailableCASServers()** – returns the list CAS Servers that AGSSO is able to submit tasks to. Any task part of a workflow that must be executed should match the capabilities of at least one CAS installed on a CAS Server that is part of the computational infrastructure.

- **String getInstalledCASs(*String CASServerURL*)** – returns the list of the CASs installed on the CAS Server

- **String getAllSupportedSymbols(*String CASServerURL*)** – returns the list of supported OM Symbols by any CAS installed on the specified CAS Server. The returned message contains the list of symbols supported by the specified CAS Server, and for each symbol, it contains the list of CASs that supports it.

- **String getAllSupportedFunctions(*String CASServerURL*)** – returns the list of functions supported by any CAS installed on the CAS Server with the supplied name. The returned message contains the list of function supported by the specified CAS Server, and for each function, it contains the list of CASs that supports it.

- **String getCASsSupportingFunction(Sting *functionName*, String *package*)** – returns the list of CAS and detail regarding the CAS that implements the function given as a parameters
- **String getCASsSupportingSymbol(Sting *omSymbol*, String *omcd*)** – returns the list of CASs that support the OM Symbols that have the name *omSymbol* and are part of the OM Cd *omcd*, regardless the CAS Server implementing it. The returned message contains information regarding the CAS hosting CAS Server.
- **String getFunctionsMatch(Sting *match*)** – get the list of functions that are implemented by a CAS installed on any CAS Server and that have as a substring of their name or of their package name the string given as parameter. For every matching function, identification information regarding the CAS implementing it is also retrieved: the CAS Server's service URL and the CAS name.
- **String getSymbolsMatch(Sting *match*)** – get the list of OM symbols that are supported by a CAS installed on the CAS Server and that have as a substring of their name or of their Cd name the string given as parameter. For every matching symbol, identification information regarding the CAS implementing it is also retrieved: the CAS Server's service URL and the CAS name.
- **String getSupportedFunctions(*String CASServerURL*, Sting *casName*)** – the list of functions that are implemented by the CAS with the *casName*, installed on the CAS Server identified by *CASServerURL*
- **String getSupportedSymbols(*String CASServerURL* ,String *casName*)** – – returns the list of supported OM Symbols by the CAS having the name *casName*, installed on the CAS Server identified by *CASServerURL*

All operations specified above return results formatted as strings. The results are well formed XML documents that contain the actual relevant information, depending on the operation being invoked. It must also be noted that the results contains information regarding only those CASs, functions, OpenMath Symbols, etc.., that can be used to formulate tasks/execution worflows.

The structure of the *getInstalledCASs* operation's response

The string returned by invoking the operation of the service is an XML document with the following structure:

```
<!ELEMENT casservers (serverurl+) >
<!ELEMENT serverurl (#CDATA)>
```

The structure of the *getAllSupportedSymbols* operation's response

The string returned by invoking the operation of the service is an XML document with the following structure:

```
<!ELEMENT symbolinfos (symbolinfo+) >
<!ELEMENT symbolinfo (casinfo+, symbol)>
<!ELEMENT casinfo (casserver, cas)>
<!ELEMENT casserver (#CDATA)>
<!ELEMENT cas (#CDATA)>
<!ELEMENT symbol (symbolname, symbolcd, symboldescription)>
```

```
<!ELEMENT symbolname (#CDATA)>
<!ELEMENT symbolcd (#CDATA)>
<!ELEMENT symboldescription (#CDATA)>
```

The structure of the *getAllSupportedFunctions* operation's response

The string returned by invoking the operation of the service is an XML document with the following structure:

```
<!ELEMENT functioninfos (functioninfo+) >
<!ELEMENT functioninfo (casinfo+, function)>
<!ELEMENT casinfo (casserver, cas)>
<!ELEMENT casserver (#CDATA)>
<!ELEMENT cas (#CDATA)>
<!ELEMENT function (functionname, functionsignature,
                                Functionpackage, functiondescription)>
<!ELEMENT functionname (#CDATA)>
<!ELEMENT functionsignature (#CDATA)>
<!ELEMENT functionpackage (#CDATA)>
<!ELEMENT functiondescription (#CDATA)>
```

The structure of the *getCASsSupportingFunction* operation's response

The string returned by invoking the operation of the service is an XML document with the following structure:

```
<!ELEMENT cass (cas+) >
<!ELEMENT cas (casserverurl, casname, casdescription, casexample,
                  cascanpause,
                   cpupower, availablecpupower, nrofprocessors,
                    totalram, availableram, availablediskspace)>
<!ELEMENT casserverurl (#CDATA)>
<!ELEMENT casname (#CDATA)>
<!ELEMENT casdescription (#CDATA)>
<!ELEMENT casexample (#CDATA)>
<!ELEMENT cascanpause (#CDATA)>
<!ELEMENT availablecpupower (#CDATA)>
<!ELEMENT nrofprocessors (#CDATA)>
<!ELEMENT totalram (#CDATA)>
<!ELEMENT availableram (#CDATA)>
<!ELEMENT availablediskspace (#CDATA)>
```

The structure of the *getCASsSupportingSymbol* operation's response

The string returned by invoking the operation of the service is an XML document with the following structure:

```
<!ELEMENT cass (cas+) >
<!ELEMENT cas (casserverurl, casname, casdescription, casexample,
                  cascanpause,
                   cpupower, availablecpupower, nrofprocessors,
                    totalram, availableram, availablediskspace)>
<!ELEMENT casserverurl (#CDATA)>
<!ELEMENT casname (#CDATA)>
<!ELEMENT casdescription (#CDATA)>
```

```
<!ELEMENT casexample (#CDATA)>
<!ELEMENT cascanpause (#CDATA)>
<!ELEMENT availablecpupower (#CDATA)>
<!ELEMENT nrofprocessors (#CDATA)>
<!ELEMENT totalram (#CDATA)>
<!ELEMENT availableram (#CDATA)>
<!ELEMENT availablediskspace (#CDATA)>
```

The structure of the *getFunctionsMatch* operation's response

The string returned by invoking the operation of the service is an XML document with the following structure:

```
<!ELEMENT functioninfos (functioninfo+) >
<!ELEMENT functioninfo (casinfo+, function)>
<!ELEMENT casinfo (casserver, cas)>
<!ELEMENT casserver (#CDATA)>
<!ELEMENT cas (#CDATA)>
<!ELEMENT function (functionname, functionsignature,
                                Functionpackage, functiondescription)>
<!ELEMENT functionname (#CDATA)>
<!ELEMENT functionsignature (#CDATA)>
<!ELEMENT functionpackage (#CDATA)>
<!ELEMENT functiondescription (#CDATA)>
```

The structure of the *getSymbolsMatch* operation's response

The string returned by invoking the operation of the service is an XML document with the following structure:

```
<!ELEMENT symbolinfos (symbolinfo+) >
<!ELEMENT symbolinfo (casinfo+, symbol)>
<!ELEMENT casinfo (casserver, cas)>
<!ELEMENT casserver (#CDATA)>
<!ELEMENT cas (#CDATA)>
<!ELEMENT symbol (symbolname, symbolcd, symboldescription)>
<!ELEMENT symbolname (#CDATA)>
<!ELEMENT symbolcd (#CDATA)>
<!ELEMENT symboldescription (#CDATA)>
```

The structure of the *getSupportedFunctions* operation's response

The string returned by invoking the operation of the service is an XML document with the following structure:

```
<!ELEMENT casfunctions (casname, function+) >
<!ELEMENT casname (#CDATA)>
<!ELEMENT function (functionname, functionsignature,
                                functionpackage, functiondescription)>
<!ELEMENT functionname (#CDATA)>
<!ELEMENT functionsignature (#CDATA)>
<!ELEMENT functionpackage (#CDATA)>
<!ELEMENT functiondescription (#CDATA)>
```

The structure of the *getSupportedSymbols*operation's response

The string returned by invoking the operation of the service is an XML document with the following structure:

```
<!ELEMENT cassymbols (casname, symbol+) >
<!ELEMENT casname (#CDATA)>
<!ELEMENT symbol (symbolname, symbolcd, symboldescription)>
<!ELEMENT symbolname (#CDATA)>
<!ELEMENT symbolcd (#CDATA)>
<!ELEMENT symboldescription (#CDATA)>
```

4.1.1 Submitting a New Task and Task Management

SCSCP Call

Remote Function Call

Remote function call is an alternative to standard SCSP call supplied for commodity reasons. This type of invocation may be useful if the target CAS does not implement the SCSCP protocol and does not have built in support for OpenMath. Usually, such CAS instances may be started as a separate process and the required interaction with the process may be carried out through the standard input/output streams. In this situation using the CAS functionality is similar to direct interaction through the command line interface of the CAS. The CAS Server acts as a communication bridge between the remote client and the CASs installed on the local machine.

The correct formulated call must comply with the following DTD structure:

```
<!ELEMENT OMOBJ (OMA) >
<!ELEMENT OMA (OMS, OMSTR,OMSTR,OMSTR*)>
<!ELEMENT OMS (#CDATA)>
<!ATTLIST OMS cd CDATA #FIXED "casall1">
<!ATTLIST OMS name CDATA #FIXED "procedure_call">
<!ELEMENT OMSTR (#CDATA)>
```

An example of such call might be:

```
<OMOBJ>
      <OMA>
            <OMS cd="cascall1" name="procedure_call"/>
            <OMSTR>gcd</OMSTR>
            <OMSTR>default</OMSTR>
            <OMSTR >5</OMSTR >
            <OMSTR >20</OMSTR >
      </OMA>
<OMOBJ>
```

The *OMS* tag and its fixed valued parameters mark that the call is a remote function invocation so the CAS Server is able to treat the call accordingly. The first *OMSTR* tag contains the name of the function that must be called on the target CAS, the second *OMSTR* designates the package where the function is implemented, while the rest of the *OMSTR* tags represent the parameters that must be sent to the function. Based on the given description the function call string is constructed, the correct CAS is started in a new process and the call is sent to the process. For the above described call, the function call is:

```
gcd(5,20)
```

Once the result was computed by the CAS, it is read by the CAS Server and the result is sent back to the original client. Since the CAS Server is not able to semantically understand the result, it provides to the client the result "as is", in whichever format it was returned by the executing CAS. It is the responsibility of the client to parse the result and extract any useful information it may contain. The result provided to the client has the structure:

```
<!ELEMENT OMOBJ (OMSTR|OME) >
<!ELEMENT OMSTR (#CDATA)>
<!ELEMENT OME (#CDATA)>
```

where the *OMSTR* tag contains the result, or in case of an error, the *OME* contains the error message.

4.1.2 Execution Management Interface

In scientific computational field, task execution management may be proven useful due to general characteristics of such tasks: long running time, amount of computational resources required by the tasks, real-time management of computational results. Execution management may be achieved simple through the following operations:

- *PAUSE* – enables the client to pause an already submitted task. The client has submitted a task using an asynchronous call and therefore, it waits for a call-back message from the executing CAS Server with the response. If a pause request is issued, as a first consequence, the call-back containing the response is blocked, regardless of the real execution status of the task at CAS Server level. The CAS Server is not directly executing the task but it submits it further to a CAS. If the execution CAS implements pause/resume mechanisms the actual execution is paused. If not, the execution continues and, when it finishes, the result is stored so it may be provided to the client when a resume operation is called.
- *RESUME* – enables the client to resume the execution of a task that was previously paused. If the underlying CAS was actually able to pause it, the execution is resumed. If the result was obtained meanwhile, the result is sent to the client.
- *CANCEL* – enables the client to cancel a certain task. The CAS Server acknowledgs the received cancel call and tries to stop the execution of the task. Since the task was cancelled at client's will, the CAS Server does not execute a call-back call as it would normally do when a task is finished. It is the client's responsibility to manage this situation at client side.
- *EXPRESS RESULT SETING* – a task execution may require a big amount of computational resources and a long completion time. Especially when the CAS Server is used as an execution unit integrated in a bigger architecture that is able to run computational workflows it may be useful to enable the client to assume a certain result for a task without really computing it. If the result of the task is set by the client, any result obtained by computing the task is discarded. The express result set operation triggers also the call-back through which the result is provided to the waiting client. This behavior is useful especially when the CAS Server's client is a workflow management engine.

The operations on the interface that allow the above mentioned functionality are:

- void pauseTask (String clientIdentifier, String invokeIdentifier) throws CASServerFaultType
- void cancelTask (String clientIdentifier, String invokeIdentifier) throws CASServerFaultType
- void resumeTask (String clientIdentifier, String invokeIdentifier) throws CASServerFaultType
- void setTaskResult (String clientIdentifier, String invokeIdentifier, String result) throws CASServerFaultType

TODO complete explanations

4.2 OpenMath Resolver Interface

For simplicity, clarity and efficiency reasons, an OpenMath document may contain references to sections of the same document or even sections that are part of another document. In fact, the OpenMath object may be defined by reusing objects defined in the same or external document. In order to understand correctly an OpenMath object, a CAS using the described object must be able to parse any referenced object encountered. The resolve operation must implement mechanisms that collect all referenced objects used in the description of the OpenMath object being parsed even if the resolve process must use calls to remote third party storage entities that are able to provide the referenced objects.

One of the functionalities implemented by the CAS Server is ability to act as OpenMath storage repositories. The CAS Server implements two main functionalities. Given a set of OpenMath references targeting OpenMath objects that are stored as XML documents in the CAS Servers file system, the CAS Server is able to extract the list of OpenMath symbols used in the scope of the target OpenMath objects or to extract the targeted objects and store them in separate files for later retrieval by a client. Retrieving the list of OpenMath symbols is important when a client has to decide if a certain CAS Server implements all the functionality that a certain object requires. Storing extracted nodes in separate files allows the client to retrieve the targeted objects at execution time.

During the resolving process a request for the list of symbols or for the actual OpenMath objects may lead, while parsing the targeted objects, to discovery of references to documents hosted on the same machine or even on another machine. In this situation the resolving process must recurrently search for targeted objects. The resolver component of the CAS Server is responsible for contacting third party CAS Servers implementing the resolve specific operations. In this situation a resolver chain is created.

Example. Lets assume that CAS Server CS1 receives a call that contains the references CS1_OMR1 that targets an OpenMath object hosted on a local document and CS1_OMR2 that targets a OpenMath object hosted by CAS Server CS2. We also assume that the object targeted by CS1_OMR2 contains the reference CS2_OMR1 that targets an object hosted by CAS Server CS3. The resolving remote invocation chain is thus:

$$CS1 \rightarrow CS2 \rightarrow CS3$$

The nature of the messages exchanged between the three CAS Servers depends on the nature of the resolving process required. If the list of symbols is required, CS1 will formulate the request to CS2 and will suspend the resolving process until CS2 contacts back the CS1 server with the response to the request. The response contains the list of symbols discovered by CS2 or any other resolver further contacted by CS2, in our case, CS3.

A similar approach is used when the resolve process must obtain the actual OpenMath objects targeted by the references. The difference is that the response contains a list of files that the interested client should download. Using the above assumptions, CS3 finds all the OpenMath Objects that can be found locally and stores them to a temporary file. As a response, it sends to CS2 the name of the file containing the target objects. CS2 will receive the response from CS3 and will send to CS1 the file address received from CS3 and the name of the file where the OpenMath local objects were extracted. As a final step, CS1 will contact all third party servers and request the files containing the required objects.

The resolving process establishes a temporary hierarchy among the CAS Servers participating to a certain resolve process. Establishing the hierarchy CS1 → CS2 → CS3 prevents CS2 to try to resolve references that target objects hosted on CS1. No CAS Server

that has an inferior grade will try to solve references to objects hosted on a higher grade server.

TODO complete with real life examples

4.2.1 Service Interfaces Participating in OpenMath Symbol List Retrieval

Retrieving the list of OpenMath symbols that are used to describe a certain OpenMath object is important if the client needs to determine if a certain CAS is able to understand a call to solve. An OpenMath call may be solved by a CAS Server only if it manages a CAS instance that is able to understand every symbol describing the call.

Obtaining the list of OpenMath symbols describing a call should be achieved by trying to obtain the list of symbols used in the call and every targeted object used within the call. A correct approach is to group by target host the references encountered in the call and send a symbol resolve request for every target host. At CAS Server level, if parsing the referenced objects results in additional discovered references, it is the responsibility of the CAS Server to contact other CAS Servers using the same interface described below.

The interface is compound of two operations:

- *getSymbolList* – used to submit to a CAS Server a request to resolve a list of references that are in the scope of the contacted CAS Server. The references are in the scope of the CAS Server if the host used in the reference URLs matches the host of the CAS Server.

Parameters:

- *requestID* – of type string, represents an identifier of the request, unique both in the scope of the client and the CAS Server.
- *request* – of type string, represents a well formed XML describing the request. The structure of the parameter is detailed in the following sub-section.
- callbackURL – of type string, represents the URL of the service where the resolve response shall be sent.

Returns: nothing

Fault: OMRResolveFaultType(requested, errorMessage)

- *putSymbolList* – used to accept resolve responses from third party CAS Servers to which previously submitted resolve requests to obtain the list of symbols have been sent. This operation must also be implemented by any client regardless it is a CAS Server or a client that needs to resolve a list of references for the contained list of symbols.

Parameters:

- *requestID* – of type string, represents an identifier of the request that was originally sent by the client, unique both in the scope of the client and the CAS Server. It may be used to pair the original request.
- result – of type string, represents a well formed XML describing the response. The structure of the parameter is detailed in the following sub-section.

Returns: nothing

Fault: OMRResolveFaultType(requested, errorMessage)

4.2.2 Messages for Retrieving the List of OpenMath Symbols

In order to obtain the list of symbols, the client must call the correct CAS Server and supply the list of references that must be solved. The contacted CAS Server will contact the client when the submitted references and subsequent references are solved or it may respond with an error message.

The OM Symbol Resolve Request

The request message must be a plain string representing a well formed XML document having the structure described by the following DTD:

```
<!ELEMENT symbolrequest (reference+, skipreference*)>
<!ELEMENT reference (#CDATA)>
<!ELEMENT skipreference (#CDATA)>
```

An example of a valid message is:

```
<symbolrequest>
      <reference>http://host1.com/path/file#id1</reference>
      <skipreference>http://host.com/</skipreference>
</symbolrequest>
```

The above request message specifies that the server should resolve the reference specified by the "`<reference>`" element. All subsequent references that are in the scope of the host specified through the "`<skipreference>`" (i.e. the references that begin with the string `http://host.com`) should not be solved by the resolver.

The OM Symbol Resolve Response

The response message must be a plain string representing a well formed XML document having the structure described by the following DTD:

```
<!ELEMENT symbolmessage ((symbol+, reference*)|error) >
<!ELEMENT symbol (name, cd)>
<!ELEMENT name (#CDATA)>
<!ELEMENT cd (#CDATA)>
<!ELEMENT skipreference (#CDATA)>
<!ELEMENT error (#CDATA)>
```

An example of a valid response message that is sent when the resolve process ended successfully is:

```
<symbolmessage>
      <symbol>
            <name>symbol1</name>
            <cd>cd_of_symbol1</cd>
      </symbol>
      <symbol>
            <name>symbol2</name>
            <cd>cd_of_symbol2</cd>
      </symbol>
      <reference>http://host.com/skipped_ref/document#id</reference>
</symbolmessage>
```

The above response message specifies that the server found the OpenMath symbols (`symbol1, cd_of_symbol1`) and (`symbol2, cd_of_symbol2`) and encountered a reference that was in the scope of a parent resolver "`http://host.com/skipped_ref/document#id`" which the resolver did not attempt to solve.

An example of a valid response message that is sent when the resolve process ended with an error is:

```
<symbolmessage>
      <error>Error message</error>
</symbolmessage>
```

The above response message specifies that the resolve process ended with the error "`Error message`".

4.2.3 Service Interfaces Participating in Target Node Retrieval

Before starting to execute a call described based on OpenMath objects, all the required objects must be available for the executing CAS. Since CASs are not in general able to contact remote hosts to obtain objects that are not hosted on the execution machine, the CAS Server implements such mechanisms.

The interface consists of two operations. One operation allows third party clients to submit object requests while the other one is able to store responses for requests that the current CAS Server previously submitted to other CAS Servers. The signature of the operations and their usage scenario is presented below:

- *fullSolveGetList* – used to submit to a CAS Server a request to resolve a list of references that are in the scope of the contacted CAS Server. The references are in the scope of the CAS Server if the host used in the reference's URLs matches the host of the CAS Server. This operation examines the list of references, finds the targeted objects and stores them in a temporary file that is available for download.

Parameters:

- *requestID* – of type string, represents an identifier of the request, unique both in the scope of the client and the CAS Server.
- *request* – of type string, represents a well formed XML describing the request. The structure of the parameter is detailed in the following sub-section.
- *callbackURL* – of type String, represents the URL of the service implemented at client level waiting for the resolve response.

Returns: nothing

Fault: OMRResolveFaultType(requested, errorMessage)

- *fullSolvePutList* – used to accept resolve responses from third party CAS Servers to which it has previously submitted resolve requests. The request message contains a list of file URLs that may be used to retrieve the actual targeted OpenMath objects.

Parameters:

- *requestID* – of type string, represents an identifier of the request that was originally sent by the client, unique both in the scope of the client and the CAS Server. It may be used to pair the original request.
- *result* – of type string, represents a well formed XML describing the response. The structure of the parameter is detailed in the following sub-section.

Returns: nothing

Fault: OMRResolveFaultType(requested, errorMessage)

4.2.4 Messages to Retrieve the Targeted Nodes

At the moment of the execution, all the objects referenced by an OpenMath reference and used in the initial call document or in a subsequent referenced object must be available locally. The CAS executing the call should not and is not expected to contact remote machines to transfer the requested objects.

The OM Object Resolve Request

The request message must be a plain string representing a well formed XML document having the structure described by the following DTD:

```
<!ELEMENT noderequest (reference +, skipreference*) >
<!ELEMENT reference (value, newid)>
<!ELEMENT value (#CDATA)>
<!ELEMENT newid (#CDATA)>
```

```
<!ELEMENT skipreference (#CDATA)>
```

The string value supplied as a text child for the "<value>" element should contain the reference URL that must be solved. The *newid* value supplied as a text child of the "<newid>" element should be the identifier value that replaces the identifier part of initial reference. In the reference "http://host.com/skipped_ref/document#id" the target identifier "id" is replaced with the new identifier supplied in the request message. This replacement is required in order to avoid identifier collision of identifiers since all objects retrieved by executing the resolver process will be at the end stored in the same OpenMath document. As a consequence, the client request should be formulated to make sure that identifier name collision is highly improbable.

The "<skipreference>" elements should be used to designate the root URLs that identify references that should be skipped by the server. An example of a valid request message is:

```
<noderequest>
<reference>
      <value>http://host1.com/path/file#id1</value>
      <newid>newid1</newid>
</reference>
<skipreference>http://host.com/</skipreference</skipreference >
</noderequest>
```

The above request message requests that "http://host1.com/path/file#id1" should be resolved and the identifier of the target object should be replaced with the "newid1" value. It also instructs the resolver to skip all references that begin with "http://host.com/".

The OM Object Resolve Response

The request message must be a plain string representing a well formed XML document having the structure described by the following DTD:

```
<!ELEMENT noderesponse ((reference*, filename+)|error) >
<!ELEMENT reference (#CDATA)>
<!ELEMENT error (#CDATA)>
```

An example of a success response for a object solver request is presented below:

```
<noderesponse>
      <reference>http://host.com/skipped_ref/document#id</reference>
      <filename>gsiftp://temp_dir/file1</filename>
</noderesponse>
```

The message informs the client that during the resolve process a reference that was in the scope of a higher rank resolver was discovered and was not solved. It also informs the client that targeted objects were saved and may be retrieved by transferring the file at the designated URL. As the DTD shows, the list of skipped references may contain more than one element. Similarly, the list of files containing the target nodes may contain more than one file but only one file is hosted on the current CAS Server. The rest of files were created and are stored on partner CAS Servers that the current CAS Server had to contact in order to fully resolve the references encountered that were in its resolving scope.

If an error occurs during the resolve process, an error message similar with the following one may be sent to the client:

```
<noderesponse>
      <error>An error message.</error>
</noderesponse>
```

4.3 Handling update messages from monitored CAS Servers (revise)

The CAS Server may be integrated into a bigger architecture to which the CAS Server represents an execution node. Automated election of execution nodes must be based on real time information that describes the capabilities and status of the execution nodes. The interface of the CAS Server exposes operations that allow a remote client to find all these details. In order to save communication effort, a better approach is to allow CAS Servers to advertise only those changes that are relevant for a certain discovery registry, usually hosted at the same location with the automated workflow engine.

When using *Local Registry Administrator* application, changes made to CAS Server status are stored into a local database but they are not automatically advertised to the intended registries. **After the administrator operates all the changes in the CAS Server status, he must activate the synchronization to remote registries explicitly.** This operation is achieved by selecting the "Notify" menu.

When the "Notify" menu item is selected, specific messages are constructed and sent to the registries via Web services calls. **The remote registry – the recipient of the update message - must expose an operation that is ready to receive update messages from CAS Servers.**

Based on the current status of the CAS Server and the actual information successfully submitted in a previous synchronization call, the custom message tailored for every registry may contain:

2. If the registry was newly added to the CAS Server and no synchronization messages were exchanged between the CAS Server and the remote registry, the message contains:
   - the list of CASs exposed by CAS Server
   - for every CAS, the capabilities of the machine were a particular CAS is installed
   - for every CAS, complete details about the functions and OpenMath symbols that are supported/implemented by every CAS
3. If the main registry was already successfully informed by existing characteristics of the CAS Server but in the meanwhile several characteristics were updated by the CAS Server's administrator, it contains:
   - full details of the entity that was changed and identification details regarding the related CAS. For instance, if the details regarding a certain function were changed, the update message contains the details regarding the method and the list of the CAS that supports it.

Note. Each entity described in an update message, i.e. CAS, method, OpenMath symbol, etc…, is identified by a unique key that does not change over time. If a CAS named GAP1 is installed on a machine and it is registered through the *Local Registry Administrator* application, its key is advertised to all related discovery registries. If at a later time the administrator of the CAS Server decides to uninstall the GAP1 instance and add a new CAS called GAP2, it should register the adding the new CAS and not by changing the name of the CAS from GAP1 to GAP2 in the *Local Registry Administrator.* With the latter approach taken, the information in the database will be inconsistent and for all tasks that were run on the GAP1 instance the system will indicate that they were run on the GAP2 instance.

4.3.1 Changes update message structure

The update message send by the CAS Server to a certain registry contains only those pieces of information that are relevant for the registry to which the update is sent. It contains details regarding the CASs, symbols and functions that are related to the registry and that were updated since the last successful update.

The interface of the Web service through with the discovery registry accepts update from CAS Servers must have the following signature:

```
updateRegistryInfo(String CASServerURL, String updateMessage)
```

where the *CASServerURL* represents the URL address of the CAS Server sending the update message. The *updateMessage* parameter must be an XML document having the following structure:

```
<!ELEMENT server ((machine*, cas+, method*, symbol,
                         casmethod,cassymbol, symbolcd) >

<!ELEMENT machine(machineid, cpupower, availcpupower,
                        processornr, totalram, availram,
                        availdisk  )>
<!ELEMENT machineid (#CDATA)>
<!ELEMENT cpupower (#CDATA)>
<!ELEMENT availcpupower (#CDATA)>
<!ELEMENT processornr (#CDATA)>
<!ELEMENT totalram (#CDATA)>
<!ELEMENT availram (#CDATA)>
<!ELEMENT availdisk  (#CDATA)>

<!ELEMENT cas(casid, machineref ?, casname, casactive,
                   casdescription, casexample,updated)>
<!ELEMENT casid (#CDATA)>
<!ELEMENT machineref (#CDATA)>
<!ELEMENT casname (#CDATA)>
<!ELEMENT casactive (#CDATA)>
<!ELEMENT casdescription (#CDATA)>
<!ELEMENT casexample (#CDATA)>
<!ELEMENT updated (#CDATA)>


<!ELEMENT method(methodid, methodname, methodpackage,
                   methodsignature, methoddescription )>
<!ELEMENT methodid (#CDATA)>
<!ELEMENT methodname (#CDATA)>
<!ELEMENT methodpackage (#CDATA)>
<!ELEMENT methodsignature (#CDATA)>
<!ELEMENT methoddescription (#CDATA)>

<!ELEMENT symbol(symbolid, symbolname, symbolcdid,
                   symbolcdcdid, symboldescription, updated)>
<!ELEMENT symbolid (#CDATA)>
<!ELEMENT symbolname (#CDATA)>
<!ELEMENT symbolcdid (#CDATA)>
<!ELEMENT symbolcdcdid (#CDATA)>
<!ELEMENT symboldescription (#CDATA)>
<!ELEMENT updated (#CDATA)>
```

```
<!ELEMENT casmethod(casmethodcasid, casmethodmethodid,
                        casmethodactive, updated)>
<!ELEMENT casmethodcasid (#CDATA)>
<!ELEMENT casmethodmethodid (#CDATA)>
<!ELEMENT casmethodactive (#CDATA)>
<!ELEMENT updated (#CDATA)>


<!ELEMENT cassymbol(cassymbolcasid, cassymbolsymbolid,
                        cassymbolactive, updated )>
<!ELEMENT cassymbolcasid (#CDATA)>
<!ELEMENT cassymbolsymbolid (#CDATA)>
<!ELEMENT cassymbolactive (#CDATA)>
<!ELEMENT updated (#CDATA)>



<!ELEMENT symbolcd (symbolcdcdid, symbolcdname)>
<!ELEMENT symbolcdcdid (#CDATA)>
<!ELEMENT symbolcdname (#CDATA)>
```

5 The SWIP package for GAP

The targeted audience of the current system is comprised of CAS users. A set of wrapper functions for GAP, grouped into the SWIP package, allow the GAP user to create their desired workflows for solving specific mathematical problems. The workflow declared indirectly by the user is an abstract workflow description using the functions provided by the GAP SWIP package. The user does not have to specify exact details of the XML abstract language, but only how the logical tasks are composed. Each function is mapped on an XML construct of the abstract language. Each function call helps to build the workflow in an incremental approach. Thus, invoking functions in this package does not result in immediate execution of the invocation, but in adding corresponding activities to the workflow. The resulted workflow is sent to the server only when the description is complete and the user has finished describing the problem.

The package that we describe here for GAP may be easily ported for other for other CASs that support either invoking external scripts or executables, or have support for remote Web service invocations. To give a more clear view of the functionality and how it can be ported to other CASs, we provide sections with technical details that may aid an experienced user that wants to build support for our system.

Tag or tag sequence	Generating CAS Function
<workflow>	SWIP_startWorkflow()
</workflow>	SWIP_endWorkflow()
<sequence>	SWIP_startSequence()
</sequence>	SWIP_endSequence()
<multichoice>	SWIP_startMultiChoice()
</multichoice>	SWIP_endMultiChoice()
<branch>	SWIP_startChoiceBranch(condition)
</branch>	SWIP_endChoiceBranch()

<if>	SWIP_if(condition)
</if>	SWIP_endIf()
<trueBranch>	SWIP_if(condition)
</trueBranch>	SWIP_else()
<falseBranch>	SWIP_else()
</falseBranch>	SWIP_endIf()
<parallel>	SWIP_startParallel()
</parallel>	SWIP_endParallel()
<foreach>	SWIP_startForeach(initValue, endValue)
</foreach>	SWIP_endForeach()
<while>	SWIP_startWhile(condition)
</while>	SWIP_endWhile()
<invoke .../>	SWIP_invoke(CasID, command, varReference)
<variable name=”var1”> $value </variable>	SWIP_declareVariable(varName, varValue)
<casID>	SWIP_invoke(CasID, command, varReference)
<call>	SWIP_invoke(CasID, command, varReference)
<initValue>	SWIP_startForeach(initValue, endValue)
<endValue>	SWIP_startForeach(initValue, endValue)

5.1 SWIP package functions

The list of the functions provided by the SWIP package for this version is presented below. They may be used to describe arbitrary complex workflows if rules for well formed.

Remark1. Variables within the workflow are automatically managed by the current solution and with the direct support if the workflow execution engine. The user may not access directly the values of the variables when the workflow is declared. The only acces that the user has is to the handlers of a variable. The system uses two types of variables:

- variables that are meant to be used in conditional expressions of while, if, etc…This variables may only contain numerical values encoded as strings and not OpenMath encoded objects. These numerical values may be used to express conditions and may be modified by storing result of remote service invocations when they are provided as the third parameter of the *SWIP_invoke* function designed for CASs. This kind of variables must be declared using the *SWIP_declareVariable* function. The user does not provide de actual name of the variable as it is used by the workflow but provides the name of the variable that will contain the handler of the variable.

Declaring a variable will store the handler of the variable into a local variable of type string declared within the CAS. This handler may be further used to specify valid conditions. Valid conditions are expressions that can be evaluated to boolean values by a subset of rules defined by XPath standard. The subset is for the moment limited to composing simple expressions that contain decimal numbers, boolean and comparison operators, grouping parentheses and variable handlers of expressed declared variables. Examples of valid conditional expressions include (though they may not make any sense in thin context):

“5 < 4”

“1 >= $variable_1”

Note. To obtain the second expression within the CAS, the "1>=" string needs to be concatenated with a the value of a local variable that was previously assigned with the handler of a workflow variable. Thus, a call

SWIP_declareVariable(varName,"1")

results in declaring a new workflow variable, assigning that variable the value "1" and storing the handler of the workflow variable using the local variable *varName.* A concatenation between "1>=" and the value of *varName* will have the expected result :

"1 >= $variable_1"

• variables that are meant to be used to link through data flows two or more invoke activities. The input of one invoke may be thus be obtained from the output of one or more results of other invoke activities. These variables contain values encoded in OpenMath and cannot be used in the contexts of conditional expressions for if, while or other similar constructs.

A call to the *SWIP_invoke()* result in declaring a variable that will store the result as an OpenMath object and the handler is returned by the function. A attribution :

```
aVar:= invoke()
```

has the effect of storing a handler of the result obtained by the invoke in *aVar* variable. This handler can be used in a following call to mark that the result is used as input data for another invoke:

```
aVar:= invoke("CASID", call)
```

where *call* is a string obtained by concatenation of several strings and variable handlers. Let *aVar1* and *aVar2* be two local variables that hold handles *$variable_1* and *$variable_2* for to previous executed invokes. To obtain the call of a remote function in the current implementation(i.e. without full support for OpenMath encoding), the call

```
"aFunction($variable_1, $variable_2)"
```

may be obtained by concatenation:

```
"aFunction("+aVar1+","+ aVar2+")";
```

where "+" represents here the concatenation operator.

It must be noted that all variable names that appear within the condition must be preceded by the special character "$". Thus, a variable name "aVariable" will appear within the condition string as "$aVariable".

The *condition* should be a string that can be evaluated to a boolean value. The expression may be formed using numerical values, comparison operators and variable names.

SWIP_startWorkflow

Functionality:

This function is used to mark the beginning of a workflow description. Every workflow should start by calling this function. The function may only be called once to start the workflow description. Multiple redundant calls of this function will result in an error.

The function returns a unique identifier of the current workflow that is later used by the client to get the result of the workflow's execution.

Implementation Remarks:

Implementation of this function must result in starting a new workflow document and adding the <*workflow*> tag to the document.

SWIP_endWorkflow

Functionality:

This function is used to mark the end of a workflow description. Every workflow definition should be marked by calling this function. The function may only be called once in correspondence with a previous *SWIP_startWorkflow()* call. Multiple redundant calls of this function will result in an error.

Implementation Remarks:

Implementation of this function must result in ending the current workflow document by adding the <*/workflow*> end tag to the document.

Example:

GAP code:

```
SWIP_startWorkflow();
...
SWIP_endWorkflow();
```

XML Code:

```
<workflow>
...
</workflow>
```

SWIP_declareVariable

Functionality:

Declaration of a new variable may be achieved using the function call : *SWIP_declareVariable(varName, varValue)*

The *varName* parameter is a local GAP variable that will hold a string representing the handler to the variable. A variable declared with this function is intended to hold only simple data types that can be used in describing XPath condition. Most common is that the variable will have a numerical value. This kind of variables cannot be used as input parameters for *invoke* activities.

Implementation Remarks:

A variable declaration should be marked by adding a <*newvariable*> node with an attribute *name* having an automatically generate value that it is unique amongst all other variables declared so far. The value of the *name* attribute represents the handler of the variable and it will be stored in the *varName* variable. If the user supplies a *varValue*, the <*newvariable*> node should have a child text node with the value *varValue,* otherwise, the "0" text node should be provided as a child.

Example:

GAP code:

```
...
```

```
SWIP_declareVariable(aVar,”101”);
...
```

XML Code:

```
...
<newvariable name="variable_0">101</newvariable>
...
```

SWIP_startSequence

Functionality:

This function is used to mark the start of a new sequence of activities. All activities declared after the call of this function and the call of a corresponding *SWIP_endSequence* function will be executed as a sequence of activities.

Note. The sequence may contain other activities that mark parallel or other execution types.

Implementation Remarks:

This function should add a <*sequence*> start tag to the document that describes the workflow.

SWIP_endSequence

Functionality:

This function is used to mark the end of a sequence of activities.

Implementation Remarks:

A call to this function should result in adding a </*sequence*> tag to the document describing the workflow.

Example:

GAP code:

```
...
SWIP_startSequence();
  ...
SWIP_endSequence ();
        ...
```

XML Code:

```
...
<sequence>
  ...
</sequence>
...
```

SWIP_startMultiChoice

Functionality:

This function is used to mark the container of multichoice brances. The call to this function must have as direct descendants only calls to functions that mark the start/end of a branch. When all the branches were defined, a call to the function that marks the end of the current container must follow.

Implementation Remarks:

This function should add a <*multichoice*> start tag to the document that describes the workflow.

SWIP_endMultiChoice

Functionality:

This function is used to mark the end of a multichoice.

Implementation Remarks:

A call to this function should result in adding a <*/multichoice*> tag to the document describing the workflow.

SWIP_startChoiceBranch

Functionality:

The function *SWIP_startChoiceBranch(condition)* may be used to define a conditional execution within a *multichoice* container. The *SWIP_startChoiceBranch* represents a marker for the start of the branch. See also Remark1 regarding accepted condition format.

Implementation Remarks:

The call of that function should add to the workflow document a new choice branch construct tags. Thus, a new *branch* tag should be added and immediately after, a node containing the conditional expression: <*condition*>*theCondition*<*/condition*>, where the "theCondition" represents the actual condition that the user stated.

Note. Here, variable name refers to the name of the variable as it appears in the workflow document and not the name of the variable container that is used within the CAS to hold the name. For additional information see the section explaining variable types and methods of declaring them.

SWIP_endChoiceBranch

Functionality:

This function is used to mark the end of a multichoice branch.

Implementation Remarks:

A call to this function should result in adding a <*/choice*> tag to the document describing the workflow.

Example:

GAP code:

```
...
SWIP_startMultichoice();
    SWIP_startChoiceBranch( Concatenation("5 > ",aVar));
      ...
    SWIP_startChoiceBranch();
SWIP_endMultichoice ();
...
```

XML Code:

```
...
<choicebranch>
```

```
            <condition>5 &gt; $variable_0</condition>
                    ...
        </ choicebranch >
        ...
```

SWIP_if

Functionality:

The call to of *SWIP_if(condition)* function allows the user to start an *if* construct. In the scope of the *if* construct an *SWIP_trueBranch* or a *SWIP_trueBranch* followed by an *SWIP_elseBranch* may appear. The *condition* must meet the requirements stated by Remark1.

Implementation Remarks:

The call to this function must add to the document an opening tag <*if*> that must be followed by a node <*condition*>*theCondition*</*condition*> that states the condition to be met and a starting tag <*truebranch*>. For additional details see Remark1.

SWIP_endIf

Functionality:

This function is used to mark the end of the *if* construct.

Implementation Remarks:

A call to this function should result in adding a </*truebranch*> closing tag if the *if* construct does not contain an *else* branch, a </*elsebranch*> if the *if* construct contains an *else* branch, and a final </*if*> tag to the document describing the workflow.

SWIP_else

Functionality:

The call to the function *SWIP_else()* result in marking a code section that is executed if the condition specified with the *SWIP_if()* call is not met.

Implementation Remarks:

The call to this function should result in adding a closing </*truebranch*> and an opening <*elsebranch*> tag to the document.

Example:

GAP code:

```
...
SWIP_if(condition);
    ...
SWIP_else();
      ...
SWIP_endIf();
...
```

XML Code:

```
...
<if>
    <condition>5 &gt; $variable_0</condition>
       <truebranch>
```

```
            ...
        </truebranch>
        <elsebranch>
            ...
        </elsebranch>
    </if>
    ...
```

SWIP_startParallel

Functionality:

This function is used to mark that the activities that main activities that are in the scope of the parallel section will be executed in parallel. It must be noted that activities that are in the scope of a *sequence* section that is itself in the scope of a *parallel* section will be executed as a *sequence*.

Implementation Remarks:

A call to this function must add to the document a <*parallel*> starting tag.

SWIP_endParallel

Functionality:

This function is used to mark the end of the *parallel* section.

Implementation Remarks:

A call to this function should add to the document a closing tag for the parallel section : *</parallel>*

Example:

GAP code:

```
...
SWIP_startParallel();
        ...
SWIP_startParallel();
        ...
```

XML Code:

```
...
<parallel>
    ...
</parallel>
...
```

SWIP_startWhile

Functionality:

The function *SWIP_startWhile(condition)* marks the start of a *while* construct. As expected, all activities specified in the scope of the *while* construct are executed repeatedly as long as the *condition* is evaluated to the boolean *true.* The *condition* must be expressed as to meet the requirements stated by Remark1.

Implementation Remarks:

A call to this function should result in adding a <*while*> start tag that must be followed by a node <*condition*>*theCondition*</*condition*> that states the condition to be met. The *condition* must be expressed as to meet the requirements stated by Remark1.

SWIP_endWhile

Functionality:

This function is used to mark the end of the *while* construct.

Implementation Remarks:

A call to this function should result in adding a *</while>* closing tag for the *while* construct.

Example:

GAP code:

```
...
SWIP_startWhile(Concatenation("101 < ",aVar));
        ...
SWIP_startWhile();
...
```

XML Code:

```
...
<while>
    <condition>101 &lt; $variable_1</condition>
    ...
</while>
...
```

SWIP_invoke

Functionality:

Invoking a remote service may be achieved by invoking the function *SWIP_invoke(CasID, command, varReference).*

The *CasID* parameter is the identifier of the CAS that the user wants to use in solving the invoke. Valid IDs may be obtained by querying the system that the workflow will be submitted to. Examples of such IDs: "GAP", "KANT", "MAPLE", etc.

The *command* parameter is actual call of the remote function as it is documented by the system that exposes that function.

The optional *varReference* is the handler provide by a *SWIP_declareVariable* function. As a result, the variable identified by the handler stored in *varReference* will hold the numerical value obtained by invoking the remote *command.* The numerical value is represented using a string encoding of the number.

The function **returns** a string representing a handler to a the variable that holds the OpenMath representation of the result obtained by invoking remotely the *command*. It must be noted thus that a function call that has all three input parameters specified will be used to compute a numerical value. The numerical value is stored within to variables in two different formats(i.e. OpenMath encoding and plain string encoding), each of them suited to be used as an input for another invoke or as a term in a conditional expression respectively.

Note. The main difference between the variable reference returned by this function and the variable reference identified by *varReference,* as stated by Remark1, is that the variable identified by the former contains an OpenMath representation of the result and the later contains only the numerical value. An evaluation process consisting from converting the OpenMath value to a numerical value by extracting the numerical value from the OpenMath object is automatically done at the server side.

Implementation Remarks:

A call to this function should add to the workflow document an <*invoke*> node with all the details of the invocation. Several details must be filled in :

- the *invokeID* attribute of the <*invoke*> node should be assigned a unique value *invoke_N* where *N* represents the order number of the current invoke, 0-indexed

- a child node of the <*invoke*> node named <*variable*>, if the *SWIP_invoke* function was called with the optional argument . The <*variable*>node should contain a text node with the value stored in the *varReference* parameter.

- a child node of the <*invoke*> node named <*casid*>. It will contain a child text node with the ID of the CAS as it is registered in the Main Registry.

- a child node of the <*invoke*> node named <*call*>. It will contain a child text node with the actual call. For the moment OpenMath is not fully supported by CASs so the function call is expected. As the support for OpenMath is increasing, the call will be an OpenMath object encoded in XML and converted to string (i.e. the special characters like "<" must be replaced with their reference, according to the XML specification).

Example:

Assuming that *aVar* holds the handler *$variable_0,* and the localGAP variable *var* is intended to store the handler to the variable containing the result of the invoke, the GAP code:

```
...
var := SWIP_invoke("GAP","Gcd(100,10)",aVar);
...
```

translates into the XML Code:

```
...
<invoke invokeID = "invoke_0">
    <variable>$variable_0</variable>
    <casid>GAP</casid>
    <call>Gcd(100,500)</call>
</invoke>
...
```

SWIP_startForeach

Functionality:

This function is used to implement a variation of the *while* construct. This construct will repeat the activities in the scope of the *SWIP_startForeach(initValue, endValue)* for a number of times determined by the values of *initValue* and *endValue*. The two values may be references to pre-initialized variables that hold numerical values(OpenMath values may not be used) or they may represent string codifications of numerical values.

Implementation Remarks:

This function must add to the workflow document a <*foreach*> start tag and two child nodes, <*initvalue*> and <*endvalue*>. The *initvalue* and the *endvalue* must be copied to the document as text nodes of the <*initvalue*> and <*endvalue*> nodes.

SWIP_endForeach

Functionality:

This function is used to mark the end of the *forEach* construct.

Implementation Remarks:
This function must add to the workflow document a closing <*foreach*> tag.

Example:

Assuming that *localInit* and *localEnd* are GAP variables that contain either numerical values or handlers to variables previously declared the GAP code:

```
...
SWIP_startForeach(localInit, localEnd);
...
SWIP_endForeach();
```

translates to the XML Code:

```
...
<foreach>
    <initvalue>1</initvalue>
    <endvalue>10</endvalue>
    ...
</foreach>
...
```

SWIP_readOutput

Functionality:
The *SWIP_readOutput(workflowID)* function can be used to retrieve the result of the execution identified by the *workflowID.*

Implementation Remarks:
This function must invoke a remote predefined Web service of the system to supply the workflow identifier and to get the result of the computation.

Example:

The GAP code:

```
SWIP_readOutput(SWIP_processHandler);
```

will retrieve the result of the process identified by the `SWIP_processHandler`.

6 Examples

To show how our system can be used to implement scientific workflows from within a CAS we present next a couple of examples.

Note. For simplicity several situations where variables are used in the examples below are shortened. As explained above, if the user declares a variable using the call

```
SWIP_declareVariable(n,"0");
```

the *n* variable represents a local GAP variable that will store a handler to the actual variable. This means that for conditional constructs, the use of the handler stored by *n* in the correct approach is:

```
SWIP_startChoiceBranch(Concatenation(n,"<10");
```

that will result in the condition `"$variable_1<10"` is not obtained by:

```
SWIP_startChoiceBranch("$n<10");
```

Such several situations appear in the code above and are coded in this manner just to offer the user a clear picture of the code.

First, a simple example is used in order to demonstrate how composition over different CASs can be performed. For this we use the following simple formula:

```
gcd(DenominatorRat(Bernoulli(1000)), DenominatorRat(Bernoulli(1200)))
```

The denominator of the two Bernoulli numbers will be computed in parallel using the GAP algebra system, while the gcd will be determined using KANT. The GAP code for this example is presented in the next code snippet:

```
LoadPackage("SWIP");
SWIP_startWorkflow();
    SWIP_startSequence();
        SWIP_startParallel();
            aVar1:=SWIP_invoke("GAP",
                     "DenominatorRat(Bernoulli(1000))");
            aVar2:=SWIP_invoke("GAP",
                     "DenominatorRat(Bernoulli(1200))");
        SWIP_endParallel();
        SWIP_invoke("KANT", "GCD($aVar1,$aVar1:=)");
    SWIP_endSequence();
SWIP_endWorkflow();
SWIP_readOutput(SWIP_processHandler);
```

The next example is the ring workflow. Imagine a ring of services, where each service accepts request from its left neighbour (for example, an integer n), performs an action (for example, n := n + 1) and sends the new value of n to its right neighbour. The test is started with the initial value n = 0 and will be terminated by that service on which the parameter n will reach the prescribed upper bound. By implementing the ring workflow example we illustrate how the system can be used to implement more complicated workflows.

```
LoadPackage("SWIP");
SWIP_startWorkflow();
SWIP_declareVariable(n,"0");
    SWIP_startWhile("$n<10");
        SWIP_startSequence();
             aVar1:=SWIP_invoke("GAP", "Int($n+1)", "$n");
             SWIP_startMultiChoice();
                 SWIP_startChoiceBranch("$n<10");
                     SWIP_invoke("GAP", "Int($n+1)", "$n");
                 SWIP_endChoiceBranch();
             SWIP_endMultiChoice();
        SWIP_endSequence();
     SWIP_endWhile();
SWIP_endWorkflow();

SWIP_readOutput(SWIP_processHandler);
```

Another complete example for the creation of a simple workflow is given in the next code snippet:

```
LoadPackage("SWIP");
SWIP_startWorkflow();
SWIP_declareVariable(i, "0");
SWIP_startSequence();
    SWIP_invoke("GAP", "Int(1000*300)", "");
    SWIP_startWhile("$i<5");
        SWIP_invoke("GAP", "increment($i)",i);
    SWIP_endWhile();
SWIP_endSequence();
SWIP_startForeach("1", "5");
    SWIP_startParallel();
        SWIP_invoke("Kant","GCD(1234,5678)","");
        SWIP_invoke("Maple","gcd(1234,5678)","");
    SWIP_endParallel();
SWIP_endForeach();
SWIP_endWorkflow();
SWIP_readOutput(SWIP_processHandler);
```

The previous GAP code produces the following workflow in the proposed abstract language:

```
<workflow xmlns="http://www.ieat.ro">
<sequence>
    <invoke uniqueID="invoke_1">
        <casID>GAP</casID>
        <call>Int(1000*300)</call>
    </invoke>
    <variable name="variable_1">0</variable>
    <while>
        <condition>$variable_1 &lt; 5</condition>
        <invoke uniqueID="invoke_2">
            <variable>$variable_1</variable>
            <casID>GAP</casID>
            <call>increment($variable_1)</call>
        </invoke>
    </while>
</sequence>
<foreach>
    <initValue>1</initValue>
    <endValue>5</endValue>
    <parallel>
        <invoke uniqueID="invoke_3">
            <casID>Kant</casID>
            <call>GCD(1234,5678)</call>
        </invoke>
        <invoke uniqueID="invoke_4">
            <casID>Maple</casID>
            <call>gcd(1234,5678)</call>
        </invoke>
    </parallel>
</foreach>
</workflow>
```

References

[1] S.Freundt, P.Horn, A.Konovalov, S.Linton, D.Roozemond, Symbolic Computation Software Composability Protocol (SCSCP) specification, Version 1.3, 2009 (http://www.symbolic-computation.org/scscp)

[2] The OpenMath standard, http://www.openmath.org
[3] GAP Computer Algebra System, http://www.gap-system.org/
[4] Oasis Web Services Resource Specification, http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf