Composing Web-based Mathematical Services

Alexandru Cârstea, Georgiana Macariu, Marc Frîncu, Dana Petcu
Institute e-Austria Timisoara
B-dul Vasile Parvan 4, 300223 Timisoara, Romania
science@ieat.ro

Abstract

While description, discovery, communication or binding of Web services, as native capabilities of service-oriented architectures, have been intensively studied in the last decade and are currently standardized, the issue of service composition has not yet been satisfactorily solved. We propose a solution and an implementation that allows the static composition of Web and Grid services based workflows with dynamic bindings. It address a particular field, that of mathematical services. A semi-dynamic approach is also proposed and discussed.

1. Introduction

The main goal of Grid architectures is to offer a computing infrastructure that makes computing details transparent to the user. Moreover, several fields of symbolic and algebraic computing may benefit by an aggregated computing power. Multiprecision integer arithmetic for example is one of the most important fields in symbolic and algebraic computation that requires significant computing power. Also implementations of polynomial arithmetic and methods used to solve sparse systems of linear equations could benefit by additional computing power. Exposing CAS functionality as Web or Grid services offers the opportunity to use remotely available computing power. In order to solve large computational problems, several symbolic computation tools, like the Computer Algebra Systems (CAS), can benefit from such architectures if they are able to communicate with each other using a communication standard. Taking into consideration the particularities and highly specialized nature of the problems they address, one should combine different CAS instances to solve atomic subcomponents of a larger problem.

The aim of the recent project MONET [1] and its successor GENSS [2] is to solve mathematical problems through service discovery and composition by means of intelligent brokerage. The discovery process in MONET is a probabilistic one and finding a service does not guarantee the quality of the service. MathBroker [3], another MONET successor, focuses on adding semantic information to existing standards for Web services in order to find appropriate services to solve a given problem. Similar ideas were followed also in GENSS project which uses the mathematical Web services developed in MONET project. Both approaches use dynamic composition of Web and Grid services that are discovered using semantic languages such as MSDL [4] and MPDL [5] to query modified UDDI registries.

Using Web services to expose CAS functionality is not the only way of supporting remote access to mathematical services. JavaMath [6] offers an API to access mathematical services while the MathWeb-SB [7] project provides several interfaces to access such services. These exposing methods represent a ground basis for composing services in workflows.

Our model starts from the work described in [8] of exposing CAS functionality as Web and Grid services in order to achieve more complex usage scenarios. The standard interface of such services makes the composition process using WS-BPEL 2.0 [9] language relatively easy. The client requests described using OpenMath [10] are deployed using a software component written using Java SDK [11], into an ActiveBPEL engine [12]. The invocation of the services is achieved through dynamic binding of a static or a semi-dynamic workflow. Dynamic binding process is achieved by replacing some CAS identifiers used to describe the workflow with the actual URL addresses of the services that expose the CAS as a Web service.

After a short overview of Web service composition in Section 2, the specific approach for mathematical services is discussed in Section 3. Details about the composition are given in Section 4, while a concrete example is provided in Section 5. Issues regarding semi-dynamic composition are depicted in Section 6. Finally some conclusions are drawn in Section 7.
2. Web service composition background

A service oriented architecture (SOA) relays on loosely coupled software components exposed as Web services. In order to invoke a Web service the client must be aware of the interface of the service. WSDL [13] provides a standard mean to describe a Web service interface but does not offer any additional information about the semantics or the quality of service provided. Finding the method in a service interface that is appropriate for solving a specific problem can be achieved by interrogating UDDI registries.

Most of the time, complicated problems can be described using algorithms that combine multiple steps. Workflows identify, generally speaking, solutions that are obtained by combining results through invoking black box software components. The Web service composition considers a similar approach using functionalities offered by Web services to solve a problem that can be described as a workflow.

The interest manifested by researchers and software professionals for Web service composition led to the emergence of several standards. The first standards that appeared, XLANG [14] and WSFL, enabled only static composition of Web services. The first one relies on structured activities, whereas the second one permits the creation of workflows by linking activities. As it is demonstrated in [14], the XLANG language is more restrictive than WSFL. The WSFL was superseded by the BPEL4WS V1.1 standard language. The latest standard, an enhanced version of BPEL4WS V1.1 is the WS-BPEL 2.0 standard. Currently, efforts are undertaken by IBM to create a hybrid language BPELJ [15] that will allow the usage of Java programming language inside BPEL workflows.

The static composition in workflows is achieved at design time by specifying all the details of the composition elements, i.e. the services and the binding details for these services. The workflow can be described in this case as a directed graph where nodes represent services and edges represent interaction between services. A special type of static composition, namely the workflows with dynamic bindings, offers the ability to specify the actual addresses of the Web services at runtime.

The infrastructure created by the public available Web and Grid services is highly dynamic: some of the services are occasionally available, other are vanishing, new and potentially better services become available, or for some of them the interface changes. In this situation static composition approach fails to offer a robust solution. The alternative is to use dynamic workflows, generated at runtime using latest information available on the services to be composed. In this case special mechanisms and techniques to identify and invoke the right Web services are required. An overview of such methods is given in [15].

An in-between approach, considered in this paper as a solution for composing complicated workflows, can be used when several characteristics of the Web services involved in the composition are known. Our system does not have any “a priori” knowledge about the addresses of the Web services needed by the composition workflow thus the binding is dynamic. What we do know, is the structure of the operations exposed by Web services. As explained in Section 6, this detail makes workflow generation easier and entitles us to consider this approach as a semi-dynamic composition.

3. High level architecture view

The solution provided by this paper for service composition involves external components such as Web services exposing mathematical functionality. This section offers a high level image of the architecture and the components involved in Web service composition. It also highlights the role of the composing components within the architecture.

Web services can be composed to serve for complex scenarios. Solving a complex problem can be usually achieved by combining results of smaller problems in which the original problem can be decomposed.

The work presented in this paper focuses on composing Web services that expose CAS functionality. As it is already demonstrated in [1]-[3], [8], the functionality offered by CASs can be exposed as Web and Grid services. Thus, smaller problems, occasionally referred in this paper as tasks, can be computed separately by several services, as parts of an execution workflow.

The proposed system achieves the execution of mathematical based workflows with the aid of several software components. At the server side level, the main components needed to carry out the workflow execution and to manage related issues include a client manager component, an engine needed to execute the workflow, and several servers that expose CAS functionality as Web services (Figure 1). A quick overview of the architecture is presented further on.

As a first step we have implemented a functionality to execute simple workflows that follows a given pattern. Several components of the server-side architecture collaborate to obtain the result for the workflow execution.
At this moment our system allows to solve simple scenario problems that have a predefined structure. Our model involves interaction between a client CAS software component and a server side system by invoking a Web service. Using the functionality provided by a CAS the user is able to invoke a standard structure workflow and later, to retrieve the result by invoking the same Web service.

A typical scenario implies that the users specify within a CAS the interdependent tasks that compose the workflow. To execute it, the workflow is submitted to a service located at a previously known URL address. An assumption is that the user is indicating the CAS system for every task. This requirement will be overcome in a later version of our system.

Several steps are needed to transform the workflow described within the CAS to a format that complies with a standard orchestration language. The workflow that results at the client side is not complete because the client is not, and should not be aware of the URL addresses of the services that will compute the sub-tasks. As a result, the client component sends an incomplete workflow to the client manager component.

One of the most important responsibilities of the client manager component is to obtain the addresses of the CAS servers that will execute the task by consulting the main registry and to supply this information to the execution engine. For every CAS server there is a corresponding local registry that contains information about the CAS systems exposed and the functionalities supported by them. The main registry duplicates information contained into local registries that are spread on different sites of the distributed infrastructure and it is used to find appropriate CAS servers to solve the atomic tasks of the workflow.

Another responsibility of the client manager component is to send back to the client a workflow identifier that will be used later by the client to retrieve the result of the whole computation.

The management of the workflow execution must be carried out by a workflow engine located at server-side. The client management component is the one that invokes the operations provided by server machines that expose CAS functionality as Web services.

It must be emphasized that all client-server interactions as well as servers to server component interactions are encoded in XML format. The description of mathematical expressions uses the standard OpenMath.

4. Composition details

The BPEL language offers means to describe workflows and to allow execution of complicated workflows. BPEL provides also a number of constructs to control the interactions between the workflow manager and the Web services used in the composition. We aim to enable CAS users to leverage the abilities of the BPEL language when interacting with Web and Grid services. This section presents in detail the components involved in Web service composition and associated implementation issues.

4.1. Creation of workflows

Decomposing problems and translating the components into computational workflows allows the execution of atomic tasks or chunks of tasks on remote machines. These tasks can use computing power offered by servers and, depending on the nature of the problem to solve, part of them may be computed in parallel. This is a basic assumption for the functionality of the system that we propose. We expect that following this path the computation results can be obtained faster than without decomposition.

First of all, to enable workflow specification, the CAS system must provide means for linking atomic tasks together. For a greater degree of freedom, the system can let the user to specify for every task computational constrains such as the CAS systems to compute it, the amount of memory needed, etc. An alternative to the pointing out the CAS system to run the computation is to implement system tools and techniques that discover the appropriate CAS system for solving a problem. It must be noted though, that the discovery process is not well refined yet and occasionally it does not provide the best results. For this reason we consider at this moment to let the user choose the appropriate CAS for its problem.

In order to avoid the overwhelming of the connection between the client and the server with intermediate results, the workflow composed at client level is submitted to and treated by a management system. Submitting the workflow requires that the user specifies the URL of the management system.
With the provided functionality, the system user is able to specify the CAS system to be used within each computation. Each CAS has a unique identifier associated with it. To indicate that a certain CAS should be invoked, the user must provide its unique identifier. The identifiers for the CASs exposed by the system can be obtained by interrogating a provided Web service.

The simplest user scenario will involve finding out which are the CASs exposed, creating a workflow, submitting the workflow to be executed, and then retrieving the result. This general scenario is expressed in the following pseudo-code:

```plaintext
ListOfCAS = getCASList(URL);
ListOfFunction = CAS_getFunctionList(URL,CAS_ID);
CAS_execute(CAS_ID,functionName, parameter1, ..., parameterN);
```

where each of the `parameter1` through `parameterN` can be as well a `CAS_execute(...)` call.

The above presented scenario is translated, by the client side component, in an appropriate request in XML language. Individual tasks that the user wants to compute must be encoded by the client component using OpenMath objects. The standard format of the objects that must be created by the client side component is presented below. In this encoded call the `procedure` of the CAS identified by `cas_id` is invoked with the arguments `Arg1...ArgN`:

```xml
<OMOBJ>
  <OMA>
    <OMS cd="casall1" name="procedure_call"/>
    <OMSTR>cas_id</OMSTR>
    <OMSTR>procedure</OMSTR>
    <OMOBJ>Arg1</OMOBJ>
    ...
    <OMOBJ>ArgN</OMOBJ>
  </OMA>
</OMOBJ>
```

Our system offers a list of CASs that are exposed, and for these CASs, the list of available functions. All details regarding the functionality of the system is available by querying a Web service provided within the system.

### 4.2. Client management at the server level

As mentioned above, the user must be able to submit execution workflows from within a CAS and to retrieve the results of the computation. The nature of the complex problems solved, that is, long time running jobs, imposes that the results are obtained using an asynchronous communication model. When the user submits a job, a unique identifier is assigned to the workflow. This identifier will be used at a later time to obtain the results.

Workflow invocation requires that some details are specified like the ones referring to the binding information of the Web services that need to be invoked. Our system specifies two types of registries that are used to hold binding information: local registries for every CAS server and a main registry that centralizes information from the local registries. Information used to populate the main registry is obtained by querying local registries. Selection of the adequate Web service to solve a certain task is done using simple criteria. First of all, the selected Web service must expose the functionality needed to solve the task. An important detail is the number of already running jobs assigned to a certain server and in particular to a certain CAS. In the case of some CASs (e.g. Maple), if an instance is already started, an additionally second one can not be started on the same computing unit.

Greater the number of running tasks on a server, higher is the possibility that the server will not be able to handle requests efficiently. This paper does not focus on selection techniques but the design of the system offers the ability to add other policies at a later time.

The information retrieved from the Main Registry is used to create endpoint reference documents that are needed by the execution engine. More details about the structure of the binding information are provided in the next subsection.

### 4.3. BPEL process and Execution Engine

The Execution Engine is responsible of calling the services and combines the results of the computation. The results of the computation are stored into a database.

Parallel processing of symbolical computations using OpenMath specifications of the problem to be solved can be achieved using Web Services that expose CAS functionality. For composing such services we have defined a workflow using BPEL. This workflow is presented in Figure 2.

The definition of any workflow starts by identifying the Web services that need to be orchestrated. The relationship between the BPEL process and one of its interaction partner CAS Web Services (CAS-WSs) is a two-way dependency at service level: a CAS-WS is both a service provider and a client of the BPEL process.
This situation appears due to the fact that the interaction between the process and the CAS-WS is based on one-way operations: the process first asks the CAS-WS to perform a computation and, later, the CAS-WS invokes the process to send it the result. Although the configuration of the interactions between the two partners is defined by means of WSDL port types, the actual CAS-WS is dynamically determined within the process. WS-BPEL provides the mechanisms to ensure the above mentioned functionality at runtime via assignment of endpoint references to the partnerLink which defines the relationship between the two partners. The BPEL process receives as part of its input message the endpoint references of the partner CAS-WSs. The general structure of the endpoint reference, as specified by the WS-Addressing specification [17], is presented below:

```xml
<wsa:EndpointReference xmlns:s=NAMESPACE
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing">
  <wsa:Address>
    ADDRESS_URI
  </wsa:Address>
  <wsa:ServiceName PortName="PORT_NAME">
    SERVICE_NAME
  </wsa:ServiceName>
</wsa:EndpointReference>
```

Here, NAMESPACE is the target namespace from the WSDL file defining the interface of the CAS-WS, ADDRESS_URI designates the URI of the service as specified in the <service> section of the WSDL file for the service and PORT_NAME and SERVICE_NAME are the service port name, and, respectively, the service name, as specified in the same place.

Since the BPEL process is in fact a Web service, it will be invoked by sending a SOAP message to the BPEL engine. As a result a new instance of the process will be created. The input message consists of:

- a client ID: this will be used to correlate the new process instance with the client requesting the services provided by the process (i.e. perform a complex computation using various CAS-WSs);
- an array of OpenMath objects encoded as specified in [18]; these will be sent as part of the request message to the CAS-WSs;
- corresponding to each OpenMath object in the above mentioned array, an endpoint reference is sent as part of the input message; this endpoint reference is used to determine the actual CAS-WS used for performing the computation.
As we have already mentioned, the interaction between the process and the CAS-WSs is based on asynchronous operations. This is because the operations performed by the CAS-WS could be rather long running and we do not want the workflow execution to be blocked by waiting for a response. Each of the OpenMath objects received as input can be processed in parallel and consequently we have used a parallel `forEach` BPEL activity for invoking the CAS-WSs. The service operation invoked as part of the interaction with the process requests as input:

- an OpenMath object specifying the CAS and its function to call;
- the URL of the process needed in order by the CAS-WS to return the result of its computations;
- a client identifier and an index (i.e. the index of the `forEach` iteration corresponding to this service invocation); these represent correlation data used to keep a track of the group of messages that belong together in this particular partner interaction.

After the service finishes the requested computation, it calls back the BPEL process to send it the result. The service will send not only the actual result but also the correlation data mentioned necessary for identifying the correct instance of the BPEL process that initially invoked the service.

After collecting all results from the CAS-WSs invoked in parallel, a final call to a CAS-WS must be carried out. This will compute the final result using as arguments the gathered results. After the final result is computed, it is stored in a database, from where the client can retrieve it later, using its client identifier. It is possible that one or more of the parallel computations result in an error. If this is the case, the final computation will not be carried out anymore and the error message will be stored in the database instead.

5. Example

In order to demonstrate the functionality of the system that we have implemented we present a simple example that involves the following formula:

\[ \text{gcd}(\text{Bernoulli}(1000), \text{Bernoulli}(1200)). \]

By this example we intend to show how the composition over multiple CASs can be performed. Therefore we first make a call that involves greatest common divisor (gcd) calculation in an instance of the KANT system. The two arguments of the KANT’s gcd function are the Bernoulli numbers of index 1000, respectively 1200, which are computed in parallel using two separate instances of the GAP system. The OpenMath call required to compute the above formula is presented below:

```xml
<OMOBJ>
  <OMA>
    <OMS cd="cascall" name="procedure_call"/>
    <OMSTR>KANT</OMSTR>
    <OMSTR>Gcd</OMSTR>
    <OMSTR>
      <OMS cd="cascall" name="procedure_call"/>
      <OMSTR>GAP</OMSTR>
      <OMSTR>DenominatorRat</OMSTR>
      <OMSTR>Bernoulli(1000)</OMSTR>
    </OMA>
    <OMA>
      <OMS cd="cascall" name="procedure_call"/>
      <OMSTR>GAP</OMSTR>
      <OMSTR>DenominatorRat</OMSTR>
      <OMSTR>Bernoulli(1200)</OMSTR>
    </OMA>
  </OMA>
</OMOBJ>
```

The above OpenMath code could be invoked from any CAS or client component that is able to construct and to issue such a call.

For verification and visualization purposes we have built a portal that enables the submission of the call. Figure 3 presents its Web interface.
Figure 4. Result fetching

As a result of the request, the client is assigned a unique identifier that can be later used to retrieve the results. Figure 4 shows how the client is able to provide the unique ID to the system and the result that it gets.

6. Semi-dynamic workflows

The previous sections presented a system that offers a flexible and easy to use support for describing and executing basic symbolic computing workflows. Due to the static nature of the involved workflows, certain limitations occur. For more complicated scenarios that cannot be modeled using the workflow pattern described above, dynamic workflow must be considered. Generating dynamic workflows offers to the end user the freedom to imagine arbitrary complex execution patterns.

Classic dynamic workflows are generated at runtime as follows. For a certain problem that a user wants to solve, such a system would have to execute several steps. The first step is the discovery of the services that will carry out the execution. The second step is to create a workflow by linking together the required services. This second step may involve complex processing required by data type conversions: if the output of a certain service is the input for another service, data conversions from output data type to input data type must be performed. These conversions must be handled at discovery time because there is no previous information regarding the interface of these services.

For several reasons, enabling our system to handle complicated scenarios that involve dynamic workflows is an easier task than the one dealing with classic dynamic workflows. The first reason is that the problems induced by data type conversion are eliminated since our CAS servers expose functionality in a standard format: the use of OpenMath objects to specify input and output of CAS operations offers the certainty of interoperability between services. The second reason is offered by the structure of services that must be invoked to compute a workflow. With dynamic workflows, the number of services that must be invoked to obtain a certain result might vary from one execution to another of the same workflow due to availability issues. Our system will call the same type of services for several invocation of the same workflow. These reasons entitle us to say that the architecture we propose targets semi-dynamic workflows.

In order to support dynamic capabilities the system must suffer several modifications. The structure and functionality of the CAS server components does not need to be changed in any way. Some changes are still required within the client component and the client management component.

When more complicated workflows than the ones presented in previous section must be supported by the system, the user that specifies the workflow must also dispose of additional system functionalities. One scenario is that in which the user is able to delimit the code representing a workflow that maps in every respect the capabilities offered BPELWS 2.0 language. An instantiation for such a scenario is presented below:

```plaintext
ListOfCAS=getCASList(URL);
ListOfFunction= CAS_getFunctionList(URL,CAS_ID);
startComposingSequence();
    Out1=CAS_execute(CAS_ID,functionName1,parameters)
    Out2=CAS_execute(CAS_ID,functionName2,Out1);
...
endComposingSequence();
executeComposingSequence();
```

Specifying a workflow is achieved by linking tasks together; for complex workflows, intermediate variables specify the relations between tasks. In the hypothetic example presented above, the linkage of the two tasks is marked by the appearance of the \textit{Out1} variable as output of the first task and as input of the second task.

In order to support even more complicated workflows, the client side components must be enriched with workflow generating capabilities. The code written by the user within the CAS system must be translated into a BPEL language workflow. That imposes code analysis in order to map code written CAS language to BPEL language. As an example, workflows often contain chunks of tasks that can be executed in parallel. In order to detect such parallel
For instance, the ActiveBPEL engine refreshes the list induced by the time required to deploy the workflow. Flows at run time involves supplementary latency that are encountered more often. Developing workflows from within the CAS should indicate those workflows carry out all the steps required by the execution engine. The deployment step is needed. The client manager must already deployed into the BPEL execution engine, a minor change. Since the BPEL workflow is not dynamic generated workflows this component requires to the client manager component. In order to support workflow document, the workflow can be forwarded to the client manager workflows every 10 seconds. When usage patterns are identified, the best solution is to create static workflows that are deployed on the server side. While this is the best solution for previous known interaction patterns, by using only those patterns, the user does not have the ability to change them in any way – in this respect, they may be considered, from the user point of view, as plain Web services.

7. Conclusions

Static workflow patterns with dynamic links to existing Web and Grid mathematical services were implemented, tested, and reported in this paper. These patterns are taking into consideration the specific requirements of the mathematical services. It is also shown that this approach is not sufficient for arbitrary complex workflows and one should be also considering a dynamic workflow solution. Due to the structure of the mathematical services that are composed a compromise can be done by using semi-dynamic workflows. This approach will be further explored and implemented in the near future.

Acknowledgments. This research is partially supported by European Union Framework 6 grant RII3-CT-2005-026133 SCIENCE: Symbolic Computing Infrastructure in Europe.

References