A Knowledge-based Framework for Building Web Service Domains[1]

Nick Bassiliades and Ioannis Vlahavas

Department of Informatics, Aristotle University of Thessaloniki, Greece
{nbassili|vlahavas}@csd.auth.gr

Abstract. This paper describes a knowledge-based framework, called SWIM, for building Web Service Domains, which are collections or communities of related Web Services that are mediated and/or aggregated by a single Web Service, called the Mediator Service that functions as a proxy for them. When a requestor sends a message to the Mediator Service our system will select one or more of the Web Services to dispatch the message and will fuse the results returned by the selected services. The selection of Web services and the algorithm for fusing the results is defined by the administrator of the Service Domain using a declarative rule language, called X-DEVICE. SWIM system offers services for registering new Web Services and Service Domains. The main advantage of the SWIM system, compared to similar proposed approaches is that it allows the easy definition of arbitrary service selection strategies using a logic-based language. Furthermore, it goes beyond the mere conditional re-routing of Web Service requests by allowing combination of results of multiple Web Services leading to a simple logic-based form for Web Service composition.

1 Introduction

The Web is becoming more than just a collection of documents; applications and services are coming to the forefront. Web services will play a crucial role in this transformation as they will become the basic components of Web-based applications [15]. A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by internet protocols [10].

The use of the Web services paradigm is expanding rapidly to provide a systematic and extensible framework for application-to-application (A2A) interaction, built on top of existing Web protocols and based on open XML standards. Web services aim to simplify the process of distributed computing by defining a standardized mechanism to describe, locate, and communicate with online software systems. Essentially,

[1] Partially supported by the Greek R&D General Secretariat through a bilateral Greek-Ukrainian project (EPAN-M.4.3, No. 2013555).

each application becomes an accessible Web service component that is described using open standards.

When individual Web Services are limited in their capabilities, they can be composed to create new functionality in the form of Web Processes. Web Service composition is the ability to take existing services (or building blocks) and combine them to form new services [16] and is emerging as a new model for automated interactions among distributed and heterogeneous applications. To truly integrate application components on the Web across organization and platform boundaries merely supporting simple interaction using standard messages and protocols is insufficient [1] and Web services composition languages, such as WSFL [14], XLANG [19] and BPEL4WS [12], are needed to specify the order in which WSDL services and operations [11] are executed.

Web Service Domain is a service composition model where a requestor needs a collection of related services that he/she will use in a non-predefined manner [15]. Properties beyond the signature level of a concrete service are irrelevant to a requestor, i.e. individual ports providing the same service are indistinguishable from a requestor's point of view. A service domain aggregates these services by providing a single service that functions as a proxy for them [18]. When a requestor sends a message to this proxy the environment will select one of the services and dispatch the message to it. Another reason for building Service Domains is to increase system scalability for large Web-based applications [7]. When the number of services to be composed is large and continuously evolving, the most appropriate approach to follow is *divide-and-conquer*; services providing similar capabilities are grouped together, and these groups take over some of the responsibilities of service composition.

Existing approaches for building Service Domains [18] or Service Communities [7] just select a single service for re-routing the requestor message that arrives at the proxy service. In this paper we go beyond this simple aggregation model for Service Domains and we propose the SWIM system, which is a knowledge-based framework for building Web Service Domains that have the capability of delegating a single request to multiple Web Services and fusing the results into a single response message. The selection of Web services and the algorithm for fusing the results is defined by the administrator of the Service Domain using a declarative rule language, called X-DEVICE. SWIM system offers services for registering new Web Services and Service Domains. The main advantage of the SWIM system, compared to similar proposed approaches is that it allows the easy definition of arbitrary service selection strategies using a logic-based language. Furthermore, it goes beyond the mere conditional re-routing of Web Service requests by allowing combination of results of multiple Web Services leading to a simple logic-based form for Web Service composition.

The rest of the paper is organized as follows. Section 2 presents the architecture and main functionality of the SWIM system. Section 2.3 presents an overview of X-DEVICE, a deductive object-oriented XML database system [6] that is used for registering Web Service components and for defining logic-based algorithms for selecting Web Services to delegate the requestor message and for fusing the results. Section 4 gives several examples of how the X-DEVICE rule language is used in SWIM to manage the Service Domains. Finally, Section 5 concludes this work and poses future research directions.

In the rest of the paper we will use the WebDisC system [20] as an example of a Service Domain under the SWIM system. WebDisC is a knowledge-based Web information system for the fusion of syntactically heterogeneous classifiers induced at geographically distributed databases. SWIM is actually a generalization of the features firstly encountered during the development of the WebDisC system. In the paper we show how the WebDisC system could be developed as a Service Domain of the SWIM system.

2 SWIM System

SWIM is a knowledge-based framework for building Service Domains i.e. communities of related Web Services. Each Service Domain consists of one or more Mediator Services. Each Mediator Service either fuses (or aggregates) the results of multiple Web services or just reroutes Web Service requests to the appropriate Web Service(-s). All Web services that are mediated by the same Mediator Service perform the same functionality, although Web Services are not homogeneous, i.e. their signatures (input and output messages) may structurally differ. The system's main functionality includes: i) a declarative rule language for defining Web Service selection strategies and result aggregation algorithms, ii) Web services for creating new mediators and domains and for registering Web Services. The rest of this section describes the architecture, functionality, and methodologies of the system. The architecture of SWIM comprises 5 basic components as depicted in **Fig. 1**: i) Clients, ii) Domain administrators, iii) Web Service administrators, iv) the SWIM server, and v) the SWIM Nodes.

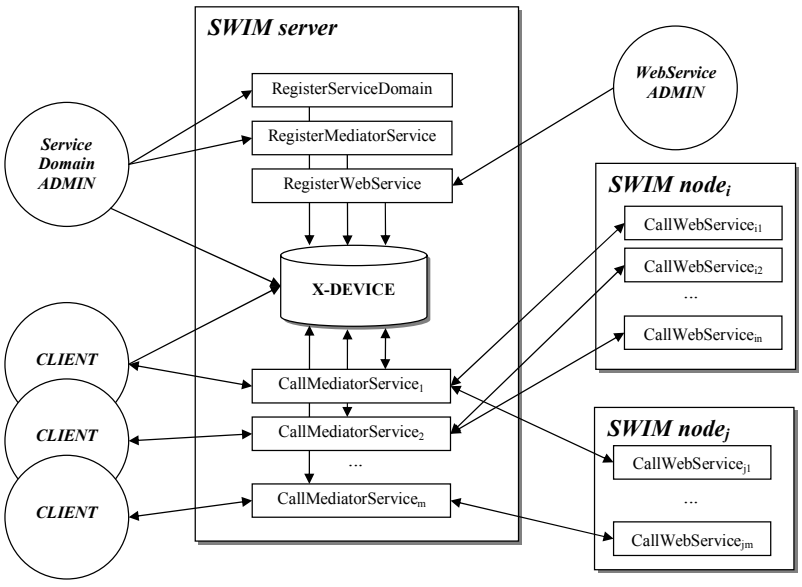

Fig. 1. The architecture of SWIM

2.1 SWIM Nodes

SWIM Nodes are Web sites that host one or more Web services that participate in the Service Domain hosted by SWIM. The WSDL descriptions of all SWIM Node Web services must follow the template (sample) that can be found at [17], along with the common structure of the input and output messages. A sample input message that conforms to this schema is shown in **Fig. 4**.

Web services need to register to the SWIM server through the `RegisterWS` service ([17]). The input message of this service describes the names and types of input and output attributes of the Web service, as well as additional data that characterize the service and help the mediator service at selecting Web Services and processing the results (see sample message in **Fig. 2**).

2.2 SWIM Server

The SWIM Server is the coordinating component of the system. It consists of the X-DEVICE deductive XML database system and the `RegisterServiceDomain`, `RegisterMediatorService`, and `RegisterWS` services for registering the corresponding entities. Furthermore, for each mediator service that has been registered there is a corresponding Web service. WSDL descriptions for the above services as well as for a sample Mediator Service can be found at [17].

X-DEVICE's main purpose is the storage of meta-data regarding the Service Domains, Mediator Services and Web Services that are registered with SWIM. These meta-data include: service names, descriptions, and addresses, names and types of the input and output attributes, plus additional data needed for Web service selection and/or result fusion. The meta-data DTDs define the type of objects that are stored in X-DEVICE for each entity type, according to the XML-to-object mapping scheme of X-DEVICE. Notice that the actual XML Schema data types for `attType` and `wSAddress` elements are `xs:anyType` and `xs:anyURI`, respectively. Sample metadata for Service Domain, Mediator Service and Web Service can be found in [17].

2.3 Clients and Administrators

Clients are applications that exploit the functionality of SWIM by directly using the SWIM server's Mediator Services either to combine results from multiple Web services or to just use Web services without knowing details regarding their name and location. In the latter case Mediator Services just re-route the incoming request to the appropriate Web Service.

Service Domain administrators are users that register entities in the SWIM server. Service Domain administrators first register a Service Domain and then can register one or more mediator services within that Domain. Web service administrators register Web services for a specific Mediator Service within SWIM.

3 X-DEVICE Rule Language

X-DEVICE is an OODB system that stores XML documents by automatically mapping the DTD to an object schema. Furthermore, X-DEVICE employs a powerful rule-based query language for intelligently querying stored Web documents and data and publishing the results. X-DEVICE is an extension of the active object-oriented knowledge base system DEVICE [4]. DEVICE integrates deductive and production rules into an active OODB with event-driven rules [13], on top of Prolog. This is achieved by translating the condition of each declarative rule into a set of complex events that is used as a discrimination network to incrementally match the condition against the database.

The advantages of using a logic-based query language for XML data come from the well-understood mathematical properties and the declarative character of such languages, which both allow the use of advanced optimization techniques, such as magic-sets. Furthermore, X-DEVICE compared to the XQuery [8] functional query language has a more high-level, declarative syntax that allows users to express everything that XQuery can express, in a more compact and comprehensible way, with the powerful addition of general path expressions, which is due to fixpoint recursion and second-order variables.

3.1 XML Object Model

The X-DEVICE system translates DTD definitions into an object database schema that includes classes and attributes, while XML data are translated into objects. Generated classes and objects are stored within the underlying object-oriented database ADAM [13]. The mapping of a DTD element to the object data model depends on the following:

- If an element has `PCDATA` content (without any attributes), it is represented as a string attribute of the class of its parent element node. The name of the attribute is the same as the name of the element.
- If an element has either a) children elements, or b) attributes, then it is represented as a class that is an instance of the `xml_seq` meta-class. The attributes of the class include both the attributes of the element and the children elements. The types of the attributes of the class are determined as follows:
  - Simple character children elements and element attributes correspond to object attributes of string type. Attributes are distinguished from children elements through the `att_lst` meta-attribute.
  - Children elements that are represented as objects correspond to object reference attributes.

The order of children elements is handled outside the standard OODB model by providing a meta-attribute (`elem_ord`) for the class of the element that specifies the correct ordering of the children elements. This meta-attribute is used when (either whole or a part of) the original XML document is reconstructed and returned to the user. The query language also uses it.

Alternation is also handled outside the standard OODB model by creating a new class for each alternation of elements, which is an instance of the `xml_alt` meta-class and it is given a unique system-generated name. The attributes of this class are determined by the elements that participate in the alternation. The structure of an alternation class may seem similar to a normal element class; however the behaviour of alternation objects is different, because they must have a value for exactly one of the attributes specified in the class.

The mapping of the multiple occurrence operators, such as "star" (*), etc, are handled through multi-valued and optional/mandatory attributes of the object data model. The order of children element occurrences is important for XML documents, therefore the multi-valued attributes are implemented as lists and not as sets.

Examples of objects and OODB schemata that are generated using the mapping scheme of X-DEVICE can be found in [21].

3.2 XML Deductive Query Language

X-DEVICE queries are transformed into the basic DEVICE rule language and are executed using the system's basic inference engine. The query results are returned to the user in the form of an XML document. The deductive rule language of X-DEVICE supports generalized path and ordering expressions, which greatly facilitate the querying of recursive, tree-structured XML data and the construction of XML trees as query results. These advanced expressions are implemented using second-order logic syntax (i.e. variables can range over class and attribute names) that have also been used to integrate heterogeneous schemata [6]. These XML-aware constructs are translated through the use of object meta-data into a combination of a) a set of first-order logic deductive rules, and/or b) a set of production rules that their conditions query the meta-classes of the OODB, they instantiate the second-order variables, and they dynamically generate first-order deductive rules.

In this section we mainly focus on the use of the X-DEVICE first-order query language to declaratively query the meta-data of the Web services that are represented as XML documents. More details about DEVICE and X-DEVICE can be found in [5] and [6]. The general algorithms for the translation of the various XML-aware constructs to first-order logic can be found in [6].

In X-DEVICE, deductive rules are composed of condition and conclusion, whereas the condition defines a pattern of objects to be matched over the database and the conclusion is a derived class template that defines the objects that should be in the database when the condition is true. For example, rule **`R2`** (in Section 4.2) defines that an object with attribute `serviceID` with value `WS` and attribute `serviceAddress` with value `WSA` exists in class `selectedWS` if several conditions are satisfied. For example, one of the conditions states that the input SOAP message [9] must contain an auxiliary attribute with name `select` and value `'At least one'`. Furthermore, the URL address `MSA` associated with the incoming SOAP message must coincide with the `mSAddress` attribute of a registered Mediator Service `MS`.

Class `selectedWS` is a derived class, i.e. a class whose instances are derived from deductive rules. Only one derived class template is allowed at the THEN-part (head)

of a deductive rule. However, many rules can exist with the same derived class at the head. The final set of derived objects is a union of the objects derived by all the rules.

The syntax of such a rule language is first-order. Variables can appear in front of class names (e.g. `WS`, `MS`), denoting OIDs of instances of the class, and inside the brackets, denoting attribute values, i.e. object references (`AIV`) and simple values (`MSName`), such as strings, integers, etc. Variables are instantiated through the "`:`" operator when the corresponding attribute is single-valued, and the `э` operator when the corresponding attribute is multi-valued. Conditions can also contain comparisons between attribute values, constants and variables. Negation is also allowed if rules are safe, i.e. variables that appear in the conclusion must also appear at least once inside a non-negated condition.

Path expressions can be composed using dots between the "steps", which are attributes of the interconnected objects, which represent XML document elements. For example, in the second condition of rule **`R2`** the names of the input attributes are retrieved by navigating from the top-level `mediatorService` object-element through an `inputVector` object-element to the `attName` attribute of a `pair` object-element. The innermost attribute should be an attribute of "departing" class, i.e. `inputVector` is an attribute of class `mediatorService`. Moving to the left, attributes belong to classes that represent their predecessor attributes. Notice the right-to-left order of attributes, contrary to the common C-like dot notation, that stress out the functional data model origins of the underlying ADAM OODB. Under this interpretation the chained "dotted" attributes can be seen as function compositions.

A query is executed by submitting the set of stratified rules (or logic program) to the system, which translates them into active rules and activates the basic events to detect changes at base data. Data are forwarded to the rule processor through a discrimination network (much alike in a production system fashion). Rules are executed with fixpoint semantics (semi-naive evaluation), i.e. rule processing terminates when no more new derivations can be made. Derived objects are materialized and are either maintained after the query is over or discarded on user's demand. X-DEVICE also supports production rules, which have at the THEN-part one or more actions expressed in the procedural language of the underlying OODB.

The main advantage of the X-DEVICE system is its extensibility; it allows the easy integration of new rule types as well as transparent extensions and improvements of the rule matching and execution phases. The current system implementation includes deductive rules for maintaining derived and aggregate attributes. Among the optimizations of the rule condition matching is the use of a RETE-like discrimination network, extended with reordering of condition elements, for reducing time complexity and virtual-hybrid memories, for reducing space complexity [4]. Furthermore, set-oriented rule execution can be used for minimizing the number of inference cycles (and time) for large data sets [5].

4 Managing Service Domains with X-DEVICE

In this section we describe in detail how X-DEVICE deductive rules are used to manage the Service Domains of the SWIM system.

4.1 Registration of Service Domains, Mediator Services and Web Services

The initial task that X-DEVICE performs within SWIM is to register the meta-data for all SWIM entities, namely Service Domains, Mediator Services and Web Services. The DTD of the Web Services' meta-data can be found in [17] along with the WSDL descriptions for the `registerServiceDomain`, `registerMediatorService` and `registerWS` services. New SWIM nodes sent in a SOAP message that contains their Web Service's meta-data. A sample SOAP message is shown in **Fig. 2**. The schema of the incoming SOAP message is determined at the input message of the corresponding port type of the WSDL description. Service Domains and Mediator Services are registered by the Service Domain Administrator via similar SOAP messages. Here we will only consider registration of Web services, since registration of other entities is almost identical.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   xmlns:m0="http://startrek.csd.auth.gr/registerWS.xsd">
   <SOAP-ENV:Body>
      <m:RegisterWS xmlns:m="http://startrek.csd.auth.gr/registerWS.wsdl">
         <wSName>Classifier1</wSName>
         <mediatorService>wsDistClassify</mediatorService>
         <wSDesc>A local classifier that uses a Decision Tree</wSDesc>
         <wSAddress>http://startrek.csd.auth.gr/Classifier1</wSAddress>
         <inputAtts>
            <attributePair>
               <m0:attName>income</m0:attName>
               <m0:attType>xs:integer</m0:attType>
            </attributePair>
            <attributePair>
               <m0:attName>loan</m0:attName>
               <m0:attType>xs:string</m0:attType>
            </attributePair>
            <attributePair>
               <m0:attName>card</m0:attName>
               <m0:attType>xs:string</m0:attType>
            </attributePair>
         </inputAtts>
         <outputAtts>
            <attributePair>
               <m0:attName>credit</m0:attName>
               <m0:attType>xs:string</m0:attType>
            </attributePair>
         </outputAtts>
         <additionalData>
            <dataPair>
               <m0:attName>classificationMethod</m0:attName>
               <m0:attVal>Decision Tree</m0:attVal>
            </dataPair>
            <dataPair>
               <m0:attName>acceptsMissingValues</m0:attName>
               <m0:attVal>true</m0:attVal>
            </dataPair>
         </additionalData>
      </m:RegisterWS>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 2. Sample SOAP message for registering a Web service

Input SOAP messages are stored within the X-DEVICE system using the schema for the SOAP message found in the corresponding WSDL description. However, the top-level element node of the input SOAP message is linked to an instance of the `input_soap_message` class, through the OID of the object-element node and its attribute `content`.

The following X-DEVICE rule `R1` iterates over all incoming SOAP messages that register a new Web service and generates a new `webService` object for each one of them.

```
R1
if I@input_soap_message(content:R) and
   R@registerWS(wSName:Name,mediatorService:MS,wSDesc:Desc,
         wSAddress:Address,inputAtts:IA,outputAtts:OA,
         additionalData:AD)
then registeredWS(wSName:Name,mediatorService:MS,wSDesc:Desc,
         wSAddress:Address,inputAtts:IA,outputAtts:OA,
         additionalData:AD)
```

Actually, rule R1 transforms the XML data of SOAP messages (**Fig. 2**) into Web service metadata stored as a set of objects.

4.2 Web Service Selection

One very important task of X-DEVICE is the selection of Web services that are relative to an incoming SOAP request for the Mediator Service. **Fig. 3** shows an example of such a SOAP message from the WebDisC system [20], where Web services are remote classifiers. A classifier can be selected if its output attribute `CAtt` is the same with the output attribute requested by the incoming SOAP message. Furthermore, if the classifier can function when some of its input attributes are missing then the classifier can be selected if it has at least one input attribute `Att` common to the input SOAP message. Rule **`R2`** below performs this selection and creates a derived class `selectedWS` whose instances are the selected Web Services. Notice that the ID of the original input SOAP message is kept for correlation purposes.

```
R2
if I@input_soap_message(content:C) and
   C@mediatorService(url=MSA,auxiliaryInputVector:AIV,
         attName.pair.inputVector:Att) and
   AIV@auxiliaryInputVector(pair=[P1,P2]) and
   P1@pair(attName=classificationAtt,attVal=CAtt) and
   P2@pair(attName=select,attVal='At least one') and
   MS@registeredMS(mSName:MSName,mSAddress=MSA) and
   WS@registeredWS(mediatorService=MSName,wSAddress:WSA,
         attName.attributePair.inputAtts=Att,
         attName.attributePair.outputAtts=CAtt,
         dataPair.additionalData:AMV) and
   AMV@dataPair(attName=acceptsMissingValues,attVal=true)
then selectedWS(request:I,serviceID:WS,serviceAddress:WSA)
```

Furthermore, all the input attribute-value pairs of the input SOAP message that match some of the registered Web services are also kept as instances of the candidate_atts class, using rule **R3**. This is done is order to construct later the SOAP messages to be send to the Web Services.

```
R3
if I@input_soap_message(content:C) and
   C@mediatorService(url=MSA,pair.inputVector:P) and
   P@pair(attName:Att) and
   MS@registeredMS(mSName:MSName,mSAddress=MSA) and
   WS@registeredWS(mediatorService=MSName,
         attName.attributePair.inputAtts=Att)
then candidate_atts(request:I,serviceID:WS,att_val_pair:P)
```

The addresses of the selected Web services are returned to the SWIM server along with the corresponding SOAP messages that should be sent to the corresponding Web services of the SWIM nodes. **Fig. 4** shows such a message.

The result is returned as an XML document and is calculated by rules **R4** to R7 show below. Rule **R4** creates a `webService` object that points to a selected Web Service object. Notice the use of the exclamation mark (`!`) in front of an attribute name to denote a system attribute, i.e. an auxiliary attribute that will not be a part of the query result. Rule R5 creates an `inputVector` object for each selected Web Service and

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:m0="http://startrek.csd.auth.gr/SWIM/mediatorService.xsd">
   <SOAP-ENV:Body>
      <m:MediatorService
            xmlns:m="http://startrek.csd.auth.gr/SWIM/wsDistClassify.wsdl">
         <m0:inputVector>
            <m0:pair>
               <m0:attName>income</m0:attName>
               <m0:attValue>14000</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>loan</m0:attName>
               <m0:attValue>good</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>card</m0:attName>
               <m0:attValue>bad</m0:attValue>
            </m0:pair>
         </m0:inputVector>
         <m0:auxiliaryInputVector>
            <m0:pair>
               <m0:attName>classificationAtt</m0:attName>
               <m0:attValue>credit</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>select</m0:attName>
               <m0:attValue>At least one</m0:attValue>
            </m0:pair>
         </m0:auxiliaryInputVector>
      </m:MediatorService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 3. Sample input SOAP message for a Mediator Service

links it with the corresponding `pair` objects. The `list(P)` construct in the rule conclusion denotes that the attribute `pair` of the derived class `inputVector` is an attribute whose value is calculated by the aggregate function `list`. This function collects all the instantiations of the variable `P` (since many input attributes can exist for each Web service) and stores them under a strict order into the multi-valued attribute `pair`. Notice that the values of the rest of the variables at the rule conclusion define a `GROUP BY` operation. More details about the implementation of aggregate functions in X-DEVICE can be found in [5] and [6].

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:m0="http://startrek.csd.auth.gr/SWIM/webService.xsd">
   <SOAP-ENV:Body>
      <m:WebService
                    xmlns:m="http://startrek.csd.auth.gr/SWIM/wsClassify.wsdl">
         <m0:inputVector>
            <m0:pair>
               <m0:attName>income</m0:attName>
               <m0:attVal>14000</m0:attVal>
            </m0:pair>
            <m0:pair>
               <m0:attName>loan</m0:attName>
               <m0:attVal>good</m0:attVal>
            </m0:pair>
         </m0:inputVector>
      </m:WebService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 4. Sample input SOAP message for a Web Service

Rule `R6` links the `inputVector` object with the corresponding `webService` object through a derived attribute rule, which defines a new attribute `inputVector` for class `webService`. The values for this attribute are derived by this rule. Objects of class `webService` that do not satisfy the condition of this class will have null value for this attribute. More details on derived attribute rules can be found in [5].

Finally, rule `R7` constructs the top-level XML element of the result which is the SOAP message built for each Web Service, augmented with the address of the classifier. The keyword `xml_result` is a directive that indicates to the query processor that the encapsulated derived class (`output_soap_message`) is the answer to the query. This is especially important when the query consists of multiple rules, as in this case.

```
R4
if C@selectedWS(request:R,serviceID:WS)
then webService(!request:R,!serviceID:WS)

R5
if WS@webService(request:R, serviceID:WS1) and
   A@candidate_atts(request=R,serviceID=WS1,att_val_pair:P)
then inputVector(serviceID:WS1,pair:list(P))

R6
if WS@webService(serviceID:WS1) and
```

```
   IV@inputVector(serviceID=WS1)
then WS@webService(inputVector:IV)

R7
if WS@webService(request:R, serviceID:WS1) and
   C@selectedWS(request=R,serviceID=WS1,address:URL)
then xml_result(output_soap_message(!request=R,!address:URL,
         content:WS))
```

4.3 Fusing Web Service Results

After the SOAP messages described in the previous subsection are sent to the selected Web Services of the SWIM nodes, the SWIM server waits for the results to be returned by all of them. X-DEVICE can be used for combining the results from the Web Services and for constructing a single result to be returned by the Mediator Service to the original requester. Again we use an example from the WebDisC system, where the category that an instance belongs (according to the Mediator Service-Classifier) is decided using Majority Voting, i.e. the category is the one that most Web Services-Classifiers decide.

Rules `R8` to `R15` below implement the majority voting algorithm and construct the SOAP message response. The first three rules implement the wait-for-all synchronization construct for the distributed Web Services. More specifically, rule `R8` counts the number of selected Web Services using the `count` aggregation function in order to compare it to the number of Web Services that responded, which are counted by rule `R9`. Rule `R10` performs the comparison and derives an `all_web_services_answered` object that is used by the final rule `R15` to construct the outgoing SOAP message only when all Web Services have responded.

Concerning the construction of the response, rule `R11` counts all distinct attribute-value pairs returned by the Web services using the `count` aggregate function. Rule `R12` finds the maximum vote count for each distinct attribute using the `max` aggregate function. Rule `R13` binds the maximum vote count with the actual most popular attribute value. Rule `R14` constructs the `mediatorService` object of the output SOAP message by tracing back and re-using an `outpuVector` object that contains the selected attribute-value combination. Furthermore, the rule traces the Mediator Service that is connected to this case and returns its URL address as a system attribute `!url`. Finally, rule `R15` constructs the top-level element for the output SOAP message, which is picked up by the SWIM server and is sent to the original requester of the Mediator Service.

```
R8
if SWB@selectedWS(request:R,serviceID:WS)
then web_services_asked(request:R,questions:count(WS))

R9
if I@input_soap_message(content:C) and
   C@webService(url:WSA,request:R) and
   SWB@selectedWS(request=R,serviceID:WS,serviceAddress=WSA)
then web_services_answered(request:R,answers:count(WS))
```

```
R10
if W1@web_services_asked(request:R,questions:N) and
   W2@web_services_answered(request=R,answers=N)
then all_web_services_answered(request:R)

R11
if I@input_soap_message(content:C) and
   C@webService(request:R,pair.outputVector:P) and
   P@pair(attName:Att,attVal:Val)
then returned_value(request:R,attName:Att,attVal:Val,
         votes:count(C))

R12
if RV@returned_value(request:R,attName:Att,votes:V)
then max_votes(request:R,attName:Att,majority:max(V))

R13
if MV@max_votes(request:R,attName:Att,majority:MaxVotes) and
   RV@returned_value(request:R,attName=Att,attVal:Val,
         votes=MaxVotes)
then majority_decision(request:R,attName=Att,attVal:Val)

R14
if MD@majority_decision(request:R,attName=Att,attVal:Val) and
   P@pair(attName:Att,attVal:Val) and
   OV@outputVector(pair=P) and
   WS1@webService(request=R,url:WSA,outputVector=OV) and
   WS@registeredWS(wSAddress=WSA,mediatorService:MSName) and
   MS@registeredMS(mSName=MSName,mSAddress:MSA)
then mediatorService(!request:R,!url:MSA,outputVector:OV)

R15
if A@all_web_services_answered(request:R) and
   MS@mediatorService(request=R,url:URL)
then xml_result(output_soap_message(!request:R,!address:URL,
         content:MS))
```

4.4 Querying Registered Web Services

The Mediator Services of the SWIM server might query X-DEVICE about the stored meta-data of the registered Web services. The following is an example from the WebDisC system that retrieves the output attributes of the registered Web Services and the input attributes that are relevant to each output attribute.

```
R16
if WS@registeredWS(attName.attributePair.outputAtts:CA) and
   not C1@corresponding_atts(outputAtt=CA)
then corresponding_atts(outputAtt:CA)

R17
if C1@corresponding_atts(outputAtt:CA) and
   WS@registeredWS(attName.attributePair.outputAtts=CA,
         attName.attributePair.inputAtt:IA)
then C1@corresponding_atts(inputAtt:set(IA))
```

```
R18
if WS@registeredWS(attName.attributePair.outputAtts:CA)
then all_output_atts(outAtt:set(CA))
```

Rule `R16` creates an instance of `corresponding_atts` class for each distinct output attribute and stores the name of the attribute in the attribute `outputAtt`. Rule `R17` iterates over all distinct output attributes, i.e. all instances of class `corresponding_atts`, and then retrieves all the input attributes of all the Web services that have the same output attribute. These input attributes are stored in the multi-valued attribute `inputAtt`, using the `set` aggregate function. This function is similar to `list`, except that no duplicate values are stored inside the list. Finally, rule `R18` creates a single instance of the class `all_output_atts` that holds a list (set) of all the distinct output attributes.

5 Conclusions and Future Work

This paper has presented the SWIM system, a knowledge-based framework for building Web Service Domains. A Service Domain is a Web Service composition model where a requestor needs a collection of related services that he/she will use in a non-predefined manner and the Service Domain aggregates these services by providing a single service that functions as a proxy for them. When a requestor sends a message to this proxy the environment will select one of the services and dispatch the message to it. Service Domains offer increased scalability for large Web-based applications.

Existing approaches for building Service Domains just select a single service for re-routing the requestor message that arrives at the proxy service. The main advantage of the SWIM system is that it allows the easy definition of arbitrary service selection strategies using a logic-based language. Furthermore, it goes beyond the mere conditional re-routing of Web Service requests by allowing combination of results of multiple Web Services leading to a simple logic-based form for Web Service composition.

In the future we will explore the possibility of extending the framework for composing not only Service Domains but for developing arbitrary Web Service composition models. We also intend to allow for user-defined selection and fusion algorithms and to enrich the system with a user-profiling system. Its purpose will be to keep the history of the user-defined selection and fusion strategies for each different user of SWIM. This way, strategies that have been successfully used in the past by a user can be retrieved and re-used in the future.

Finally, we plan to extend the current system for supporting richer Web Service meta-data expressed in an ontology language like DAML-S [2], utilizing an RDF-aware extension of our own X-DEVICE system [3]. Using domain-specific ontologies will address syntactic and semantic heterogeneity problems that arise from the possibly different data schemata that are used by the distinct Web Services. This is an important future trend in Web information systems development that is driven by the Semantic Web vision.

6 References

[1] Aalst W. van der, "Don't Go with the Flow: Web Services Composition Standards Exposed", *IEEE Intelligent Systems*, Vol. 18, No. 1, pp. 72-76, 2003.

[2] Ankolekar A. et al., "DAML-S: Web Service Description for the Semantic Web", *Proc. Int. Semantic Web Conf.*, LNCS 2342, Springer-Verlag, 2002, pp. 348–363.

[3] Bassiliades N., Vlahavas I., "Capturing RDF Descriptive Semantics in an Object Oriented Knowledge Base System", *12th Int. WWW Conf. (WWW2003)*, Budapest, Hungary.

[4] Bassiliades N., Vlahavas I., "Processing production rules in DEVICE, an active knowledge base system", *Data and Knowledge Engineering*, Vol. 24, No. 2, pp. 117–155, 1997.

[5] Bassiliades N., Vlahavas I., Elmagarmid A.K., "E-DEVICE: An extensible active knowledge base system with multiple rule type support", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, No. 5, pp. 824-844, 2000.

[6] Bassiliades N., Vlahavas I., Sampson D., "Using logic for querying XML data", *Web-Powered Databases*, D. Taniar, W. Rahayu (Eds.), pp. 1-35, Idea Publishing, 2003.

[7] Benatallah B., Dumas M., Maamar Z., "Definition and Execution of Composite Web Services: The SELF-SERV Project", *Bulletin of IEEE TC on Data Engineering*, Vol. 25, No. 4, pp.47-52, 2002.

[8] Boag S., Chamberlin D., Fernandez M.F., Florescu D., Robie J., Simeon J., "XQuery 1.0: An XML query language", November 2002. http://www.w3.org/TR/xquery/

[9] Box D., Ehnebuske D., Kakivaya G., Layman A., Mendelsohn N., Nielsen H.F., Thatte S., Winer D., "Simple Object Access Protocol (SOAP) version 1.1", May 2000. http://www.w3.org/TR/SOAP/

[10] Champion M., Ferris C., Newcomer E., Orchard D., "Web services architecture", November 2002. http://www.w3.org/TR/ws-arch/

[11] Chinnici R., Gudgin M., Moreau J., Weerawarana S., "Web Services Description Language (WSDL) version 1.2", Working Draft, July 2002. http://www.w3.org/TR/wsdl12/

[12] Curbera F. et al., "Business Process Execution Language for Web Services (v. 1.0)", IBM, July 2002. www-106.ibm.com/developerworks/webservices/library/ws-bpel

[13] Diaz O., Jaime A., "EXACT: An extensible approach to active object-oriented databases", *VLDB Journal*, Vol. 6, No. 4, pp. 282–295, 1997.

[14] Leymann F., "Web Services Flow Language (WSFL 1.0)", IBM, May 2001. www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[15] Leymann F., "Web Services: Distributed Applications without Limits - An Outline", *Proc. Database Systems for Business, Technology and Web (BTW 2003)*, Weikum G., Schöning H., Rahm E., (Eds.), GI-Edition - Lecture Notes in Informatics (LNI), P-26, Bonner Köllen Verlag, 2003.

[16] Piccinelli G., "Service Provision and Composition in Virtual Business Communities", Tech. Report, HP, 1999. www.hplhp.com/techreports/1999/HPL-1999-84.html

[17] SWIM. http://lpis.csd.auth.gr/systems/swim.html

[18] Tan Y.-S., Topol B., Vellanki V., J. Xing, "Implementing service Grids with the service domain toolkit", IBM Corporation, 2002.

[19] Thatte S., "XLANG:Web Services for Business Process Design", Microsoft, Redmond, Wash., 2001. www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[20] Tsoumakas G., Bassiliades N., Vlahavas I., "A Knowledge-based Web Information System for the Federation of Distributed Classifiers", to appear at *Web Information Systems*, D. Taniar, W. Rahayu (eds.), Idea Publishing, 2004.

[21] X-DEVICE. http://lpis.csd.auth.gr/systems/x-device.html.