Web Service Composition Using a Deductive XML Rule Language*

Nick Bassiliades[1], Dimosthenis Anagnostopoulos[2] and Ioannis Vlahavas[1]

[1]Department of Informatics, Aristotle University of Thessaloniki, Greece

{nbassili|vlahavas}@csd.auth.gr

[2]Department of Geography, Harokopio University, Athens, Greece

dimosthe@hua.gr

Abstract

This paper describes a knowledge-based Web Service composition system, called SWIM, which is based on the Service Domain model. Service Domains are communities of related Web Services that are mediated by a single Web Service, called the Mediator Service, which functions as a proxy for them. When a requestor sends a message to the Mediator Service one or more of the related Web Services are selected to dispatch the message and the results returned are aggregated to a single answer to the requestor. Mediator Services can be further composed to more complex Mediator Services that combine several selection and aggregation algorithms among many heterogeneous web services. The system utilizes the X-DEVICE deductive XML rule language for defining complex algorithms for selecting registered web services, combining the results, and synchronizing the workflow of information among the combined web services in a declarative way. In the paper, we demonstrate the flexibility and expressibility of our approach for composing Web Services using several e-business examples, covering most of the workflow patterns found in a comprehensive workflow management system [2].

Keywords: Web Service Composition, Service Domain, Deductive Object Oriented Database, XML Rule Language

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 [21]. 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 [15].

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, each application

* This work is partially supported by the Greek R&D General Secretariat through a bilateral Greek-Ukrainian project (EPAN-M.4.3, No. 2013555).

becomes an accessible Web service component that is described using open standards. The basic architecture of Web services includes technologies capable of:

- *Exchanging messages.* The standard protocol for communication among Web services is the *Simple Object Access Protocol* (SOAP) [14], a simple and lightweight XML-based mechanism for creating structured data packages that can be exchanged between network applications.
- *Describing Web services*. The standard language for formally describing Web services is *Web Services Description Language* (WSDL). WSDL [16] is an XML document format for describing Web services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented (RPC) messages.
- *Publishing and discovering Web service descriptions*. The most popular means of publishing service descriptions available to Web services is through a UDDI registry [11]. When publishing a Web service description to a UDDI registry, complete business context and taxonomies are essential if the service is to be found by its potential consumers.

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 [22] 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 [20], XLANG [25] and BPEL4WS [17], are needed to specify the order in which WSDL services and operations 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 [21]. 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 [24]. 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 [12]. 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 [24] or Service Communities [12] 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 a knowledge-based system, called SWIM, for building Web Service Domains that have the capability of delegating a single request to multiple Web Services and aggregating the results into a single response message. The selection of Web services and the algorithm for aggregating the results is defined by the administrator of the Service Domain, or occasionally by the client, using a declarative rule language, called X-DEVICE. The SWIM system also offers services for registering new Web Services, Service Domains and proxies, which we call Mediator Services.

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. Finally, the SWIM system goes even beyond the composition model of Service Domains allowing for composing complex Mediator Services, leading to a full powered Web Service composition system. This is due to the fact that the deductive rule

language allows the definition of arbitrary workflow and synchronization models among Web/Mediator Services, based on the data-driven nature of the underlying rule inference engine. Compared to other Web Service composition frameworks, logic offers declarativeness, flexibility and expressibility. Furthermore, in the paper it is shown how SWIM supports most of the workflow patterns that have been identified to cover the entire spectrum of workflow functionality [2].

The rest of the paper is organized as follows. Section 2 briefly overviews the related work in the field of web service composition. Section 3 presents the architecture and main functionality of the SWIM system. Section 4 presents an overview of X-DEVICE, a deductive object-oriented XML database system [9] that is used for defining the behavior of the composite Web Services through logic-based algorithms. Section 5 presents in detail how X-DEVICE deductive rules are used in the SWIM system to compose simple Mediator Services by selecting Web Services and combining their results, while Section 6 presents how complex Mediator Services are composed by combining simple Mediator Services. Finally, Section 7 concludes this work and poses future research directions.

2. Related Work

Recently, several Web services composition languages and standards have been proposed, such as BPEL4WS [17], WSFL [20], XLANG [25], WSCI [4], and BPML [5]. These languages are also known as Web services flow languages, Web services execution languages, Web services orchestration languages, and Web-enabled workflow languages.

BPEL4WS (Business Process Execution Language for Web Services) builds on IBM's WSFL (Web Services Flow Language) and Microsoft's XLANG (Web Services for Business Process Design) and combines accordingly the features of a block structured language inherited from XLANG with those for directed graphs originating from WSFL. The language is intended to support the modeling of executable and abstract processes. An abstract process is a business protocol, specifying the message exchange behavior between different parties without revealing the internal behavior for anyone of them. An executable process specifies the execution order between a number of activities constituting the process, the partners involved, the messages exchanged, and the fault and exception handling.

Processes in BPEL4WS are flow-charts, where each element in the process is either a primitive or a structured activity. Structured activities include sequence, conditional routing, looping, conditions based on timing or external triggers, parallel routing, and grouping activities. Structured activities can be nested and combined in arbitrary ways. Within activities executed in parallel the execution order can further be controlled by the usage of links, which allows the definition of directed graphs. The graphs too can be nested but must be acyclic.

BPML (Business Process Modeling language) is a standard developed and promoted by the Business Process Management Initiative. The main ingredients of BPML are: activities, processes, contexts, properties, and signals. Activities are components performing specific functions. There are two types of activities: simple activities and complex activities. A process is a complex activity which can be invoked by other processes. Contexts define an environment, e.g. properties, processes, signals, etc., for the execution of related activities and can be used to exchange information and coordinate execution. The complex activities offered by BPML include parallel execution, choice among multiple alternatives based on the occurrence of an event, iteration, sequence, conditional execution, and loops.

The Web Service Choreography Interface (WSCI) is an XML-based interface description language that describes the flow of messages exchanged by a Web Service participating in choreographed interactions with other services. WSCI describes the dynamic interface of the Web Service participating in a given message

exchange by means of reusing the operations defined for a static interface, working in conjunction with WSDL. WSCI describes the observable behavior of a Web Service. This is expressed in terms of temporal and logical dependencies among the exchanged messages, featuring sequencing rules, correlation, exception handling, and transactions. WSCI also describes the collective message exchange among interacting Web Services, thus providing a global, message-oriented view of the interactions. WSCI has substantial overlaps with BPML.

The main advantage of the SWIM system, compared to the above Web Service composition languages is the use of a logic-based rule language which offers declarativeness, flexibility and expressibility. The deductive rule language we use allows the definition of arbitrary service selection, result combination, workflow and synchronization models among Web/Mediator Services, based on the data-driven nature of the underlying rule inference engine. A comparison of Web Service composition languages according to their control-flow abilities [1] has shown that none of the standards directly support all workflow patterns that typically cover the entire spectrum of workflow functionality [2]. In Table 1 we summarize how our system supports these workflow patterns. With the exception of the external cancellation of an already started Web Service (and related patterns), the SWIM system can support all patterns. Table 1 either discusses the SWIM support for each pattern or points out to the appropriate section in the paper for further discussion.

The answer of the Semantic Web community to issues related to Web Services is called DAML-S [3]. DAML-S is a Web service markup language that supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in unambiguous, computer-interpretable form. DAML-S markup of Web services will facilitate the automation of Web service tasks including automated Web service discovery, execution, interoperation, composition and execution monitoring. Concerning the composition of Web Services DAML-S supports ordered and unordered sequencing, synchronized and unsynchronized parallel routing, choices, loops and decisions. Furthermore, dataflow can be defined in terms of relating input and output parameter bindings of composite and component processes. However workflow and dataflow patterns can only be statically defined. Using a logic-based language such as X-DEVICE in SWIM we are able to dynamically modify the workflow and the dataflow of the composite Web Services at run-time. However, the power of DAML-S lies at the strong semantic background of description logics. Furthermore, DAML-S offers a complete solution to the whole spectrum of Web Service creation, use and management. Our future goal is to integrate DAML-S Web Service descriptions in SWIM and use X-DEVICE deductive rules to dynamically infer service compositions.

Concerning the Service Domain composition model, two other systems exist that employ it, namely SELF-SERV [12] and the Service Domain Toolkit of IBM's Web Service Toolkit [24]. In SELF-SERV, the process model underlying a composite service is specified as a statechart whose states are labeled with invocations to Web services, and whose transitions are labeled with events, conditions, and variable assignment operations. Statecharts offer most of the constructs found in the Web Service composition languages discussed above. Furthermore, SELF-SERV exploits the concept of Service Domains, which are called Service Communities, in order to compose potentially large and changing collections of Web services. When a community receives a request for executing an operation, it delegates it to one of its current members. The choice of the delegatee is based on a selection policy involving parameters of the request, the characteristics of the members, the history of past executions, and the status of ongoing executions. The set of members of a community can be fixed when the community is created, or it can be determined through a registration mechanism, thereby allowing service providers to join, quit, and reinstate the community at any time.

IBM's Service Domain Toolkit provides prefabricated Service Domains for composing, operating, and managing Grid services, i.e. special Web services that automatically dispatch the best service available from a pool of dynamically assembled service providers in order to meet the user's need. A Grid Service can select the

appropriate Web service instance to process a Web service request, monitor the performance of the Web service instance it selected, and perform failover processing if required. Selection of a Web service instance is not just based on availability, but can also be based on Quality of Service characteristics, as specified in Web Service Level Agreements and business arrangements. Selection of a Web service is performed automatically based on a service policy, and the features provided by a Service Domain enable Service Site owners to conceal the implementation complexity required to handle multiple client requests over heterogeneous environments. A Service Domain has a nested architecture composed of heterogeneous autonomic service processing units, which themselves are either *service desks* (collections of services) or *service hubs* (collections of service desks).

Table 1. SWIM support for workflow patterns

Workflow Pattern	**SWIM support**
Sequence	1. Supported as Sequencing of Auxiliary Mediator Services (Section 6.2.1) 2. Sequencing of simple Web Services can be supported as in (1)
Parallel Split	3. Supported as message broadcasting to multiple registered Web Services from a Mediator Service (Section 6.1.1) 4. Supported in complex Mediator Services if auxiliary Mediator Services are considered independent processes (Section 6.2.1)
Synchronization	5. Supported as aggregation of responses received from multiple Web Services (Section 5.2.1) 6. See (4)
Exclusive Choice	7. Supported as re-routing from a Mediator Service to one Web Service (Sections 5.1.1, 6.1.2) 8. See (4)
Simple Merge	9. Supported in simple Mediator Services when the rule inference continues upon the first answer received (Section 5.2.1) 10. See (4)
Multi Choice	11. Supported as aggregation of responses received from multiple Web Services (Section 5.1.2) 12. See (4)
Synchronizing Merge	13. Supported as in (5) (see discussion in Section 5.2.1). 14. See (4)
Multi Merge	15. Supported as "Multiple Instances with a Priori Design Time Knowledge"
Discriminator / N-out-of-M Join	16. Supported in simple Mediator Services when the synchronization of answers received from selected web services depends either on the number of received answers or on an arbitrary, domain-dependent conditions (Section 5.2.1) 17. See (4)
Arbitrary Cycles	18. See (4)
Implicit Termination	19. Supported due to the fixpoint semantics of the X-DEVICE deductive rule inference engine (Section 4.2.2)
Multiple Instances with a Priori Design Time Knowledge	20. Supported by replicating an activity in the workflow model, i.e. by having multiple rules creating multiple output SOAP messages with different contents for the same Web/Mediator Service.
Multiple Instances with a Priori Runtime Knowledge	21. Supported by dynamically replicating an activity based on a counter, i.e. by having one rule that its condition depends on a counter. The rule each time it runs creates a different output SOAP message for the same Web/Mediator Service.
Multiple Instances without a Priori Runtime Knowledge	22. Supported by dynamically replicating an activity based on an arbitrary condition, i.e. by having one rule with a condition element that can succeed or fail depending on the application logic. The rule each time it runs creates a different output SOAP message for the same Web/Mediator Service.
Multiple Instances with Synchronization	23. Supported as a combination of "Multiple Instances without a Priori Runtime Knowledge" and "Synchronization"
Interleaved Parallel Routing	24. Supported when Auxiliary Mediator Services are selected in an arbitrary order (Section 6.2.1)
Deferred Choice	25. Depends on the "Cancel Activity" pattern
Milestone	26. Depends on the "Deferred Choice" pattern
Cancel Activity	27. Supported only if a Service has been selected for execution but an outgoing SOAP message has not yet been sent to it. This is possible due to the truth maintenance capabilities of the X-DEVICE deductive rule inference engine (Section 4.2.2). Cannot be supported in the current system, if a Web Service has started execution (Section 7).
Cancel Case	28. Depends on the "Cancel Activity" pattern

The SWIM system is capable of building more complex Service Domains than the above systems, which merely re-route the requestor message to a single service. Specifically, SWIM can delegate a single request to multiple Web Services and then aggregate the results into a single response message. The algorithms for selecting Web services and aggregating the results are defined using a declarative rule language offering great flexibility in expressing arbitrary business policies.

Compared to previous work of ours ([26], [10]), in this paper we present a complete knowledge-based system for Web Service composition that supports most of the workflow patterns found in a full-scale workflow management system. Specifically, in [26] a knowledge-based Web information system for the fusion of syntactically heterogeneous classifiers induced at geographically distributed databases, called WebDisC, was presented. WebDisC featured a domain-dependent implementation of Service Domains. Specifically, WebDisC had one Mediator Service that received classification requests from clients, forwarded the request to selected remote classifiers (web services), based on syntactical similarities of input/output data, and, finally, aggregated the results based on pre-defined or user-defined fusion algorithms.

In [10], the ideas first explored in WebDisC were generalized into the SWIM system, a domain-independent Service Domain composition framework. The work in [10] was mainly focused on managing Service Domains, giving as a specific example how the WebDisC system could be developed as a Service Domain of the SWIM system. In this paper the SWIM system is further modularized to domain-independent rules, that support common workflow patterns, and domain-dependent rules that implement the application logic for each specific Service Domain. Furthermore, this paper goes beyond the simple Service Domains of [10], by allowing Mediator Services to be composed into complex Mediator Services, leading to arbitrary Web Service composition models.

3. System Architecture and Operation

SWIM is a knowledge-based system for composing Web Services through the use of Service Domains i.e. communities of related Web Services. Each Service Domain consists of one or more Mediator Services. Each Mediator Service either reroutes Web Service requests to the appropriate Web Service (-s) or combines the results of multiple Web Services. 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:

- A deductive rule language for the definition of algorithms for:
  - Selecting Web Services within a Mediator Service;
  - Combining results of multiple Web Services within a Mediator Service;
  - Composing Complex Mediator Services within a Service Domain.
- Web services for creating new mediators and domains and for registering Web Services.

The rest of this section describes the architecture and functionality of the system.

3.1. System Components

The architecture of SWIM comprises 5 basic components as depicted in Figure 1: i) Clients, ii) Domain administrators, iii) Web Service administrators, iv) SWIM server, and v) SWIM Nodes. Furthermore, inside the SWIM server the X-DEVICE system is used as a subcomponent.

3.1.1. 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 in [23], along with the common structure of the input and output messages. A sample input message that conforms to this schema is shown in Figure 4. Web services register their metadata to the SWIM server

through an appropriate web service [10] in order to help the mediator service at selecting Web Services and processing the results.

3.1.2. SWIM Server

The SWIM Server is the coordinating component of the system. It consists of:

- Mediator Services, i.e. composite web services that aggregate the behavior of the Web Services at the SWIM nodes. WSDL descriptions for all the Mediator Services that are used as examples in this paper can be found in [23].
- Web Services for registering all the participating entities, namely Service Domains, simple and complex Mediator Services, and SWIM node Web Services. WSDL descriptions for the registration services can be found in [10] and [23].
- The X-DEVICE deductive XML database system.
- An interface between X-DEVICE and the various Web Services that forwards all the input SOAP messages to X-DEVICE as XML documents and vice-versa, i.e. it constructs output SOAP messages from the XML documents that X-DEVICE outputs.

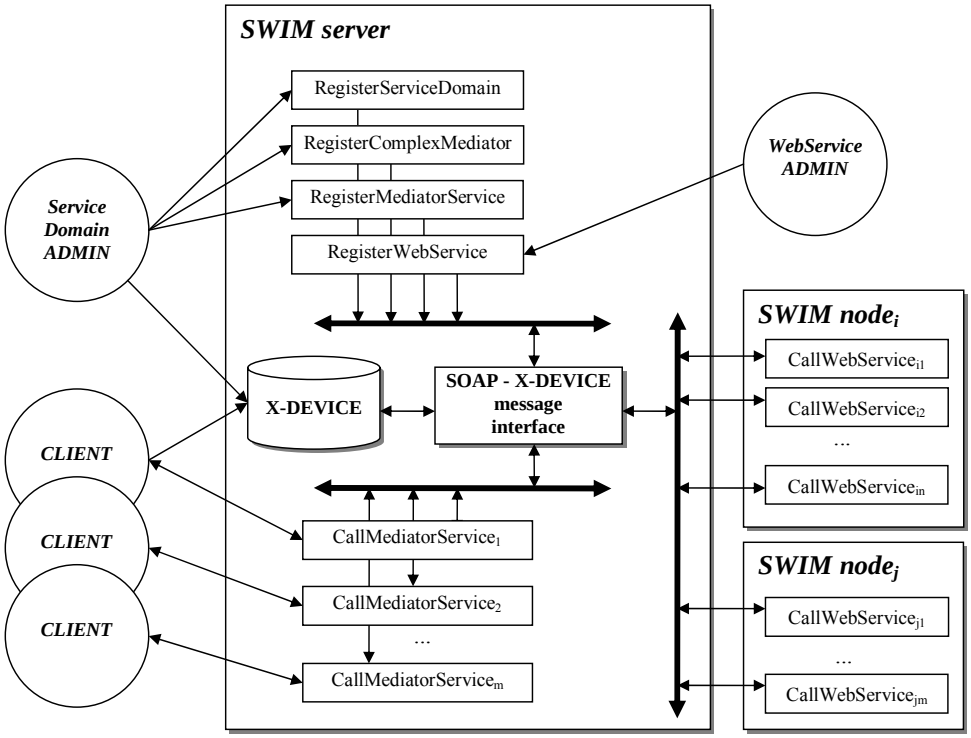

Figure 1. The architecture of SWIM

3.1.3. X-DEVICE

X-DEVICE is an OODB system that stores XML documents by automatically mapping an XML document to a set of objects. Furthermore, X-DEVICE employs a powerful rule-based query language for intelligently querying stored Web documents and data and publishing the results. X-DEVICE's main uses inside the SWIM system include the storage of metadata, the processing of SOAP messages and the definition of workflow and dataflow among component Web Services.

The SWIM system keeps metadata about the registered Service Domains, Mediator Services and Web Services. These metadata include: service names, descriptions, addresses, and organizations that own these

services, names and types of the input and output attributes, plus additional data needed for selecting Web services and combining their results. The metadata DTDs (see [23]) 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. Metadata for all Service Domains, Mediator Services and Web Services that are used as examples in this paper can be found in [23].

X-DEVICE is responsible for processing the input SOAP messages of Mediator Services, by implementing the application logic for each Mediator Service as a logic program, i.e. a set of deductive rules. X-DEVICE rules also define what SOAP messages will each Mediator Service output to the subscribed Web Services and how the results received from those Web Services will be combined to form the result to be returned to the client.

Finally, X-DEVICE coordinates and synchronizes the various distributed services involved in the system. This is achieved by the data-driven nature of the system, i.e. X-DEVICE rules fire only when all the available data in their condition become available.

3.1.4. Clients

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. Clients can also fine-tune the application logic in the form of X-DEVICE rules that can be sent for execution to the SWIM system, if the actual Service Domain allows to. For example, in the WebDisC system [26] users (clients) can define their own classifier selection and fusion strategies in this manner.

3.1.5. Administrators

Service Domain administrators are power 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. Service Domain Administrators are responsible for uploading the X-DEVICE rules which constitute the application logic of the Mediator Services. Web Service administrators register Web Services for a specific Mediator Service within SWIM.

3.2. System Functionality

The system's operation proceeds as follows:

- A Mediator Service receives a request from a client in the form of a SOAP message.
- The Mediator Service propagates the input message (request) to the message interface that further propagates it to X-DEVICE in the form of an XML document.
- X-DEVICE translates the incoming message into a set of objects according to the XML-to-object mapping scheme (see section 4.1).
- X-DEVICE's inference engine is evoked by the presence of new data and starts deriving new objects according to the deductive rules that the Service Domain Administrator and/or the Client has programmed.
- The rules eventually will produce some output XML documents which are actually messages to be sent to some selected Web Services. The output XML documents are propagated to the message interface which will transform them to SOAP messages and will forward them to the appropriate Web Services.
- When Web Services reply back, the message interface propagates the SOAP messages to X-DEVICE as XML documents, which again are translated into objects and evoke the inference engine.
- The rules now combine the results from the Web Services and construct a single answer that is returned to the client by the Mediator Service that received the initial input SOAP message.

- The output XML document is propagated to the message interface which further propagates it to the Mediator Service. The latter send it back to the requestor.

When the original message is received by a complex Mediator Service the same line of processing is followed. The only difference is that the output SOAP messages destined at the auxiliary Mediator Services never actually leave the system, because the message interface propagates them within the SWIM server.

4. Deductive Rule Language

The deductive rule language that is used in the SWIM system for defining complex Web Service compositions is the query language of the X-DEVICE system [9]. 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 [7]. DEVICE integrates deductive and production rules into an active OODB with event-driven rules [18], 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 [13] 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.

4.1. The XML Object Model of X-DEVICE

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 [19]. 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 behavior 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.

Figure 2 shows the X-DEVICE representation of an XML document that describes metadata for a web service (see [23]). More examples of objects and OODB schemata that are generated using the mapping scheme of X-DEVICE can be found in [28].

```
object           1#registeredWebService
   instance      registeredWebService
   attributes
      webServiceName          'DispatchRequest'
      mediatorService         'DispatchRequest'
      organization            'XYZ Courier Company'
      webServiceDescription   'A Dispatch Request service of a local office ...'
      webServiceAddress       'http://office1.XYZcourier.com/DispatchRequest'
      inputAttributes         11#inputAttributes
      outputAttributes        12#outputAttributes
      additionalData          13#additionalData
```

```
object         2#attributePair
   instance    attributePair
   attributes
      attName     type
      attType     xs:string
```

```
object         3#attributePair
   instance    attributePair
   attributes
      attName     weight
      attType     xs:decimal
```

```
...
```

```
...
```

```
object         8#attributePair
   instance    attributePair
   attributes
      attName     pickup
      attType     xs:time
```

```
object         9#attributePair
   instance    attributePair
   attributes
      attName     price
      attType     xs:decimal
```

```
object         11#inputAttributes
   instance    inputAttributes
   attributes
      attributePair     [2#attributePair,
         [3#attributePair,4#attributePair,
         5#attributePair,6#attributePair,
         7#attributePair,8#attributePair]
```

```
object         12#outputAttributes
   instance    outputAttributes
   attributes
      attributePair     [8#attributePair,
         9#attributePair]
```

```
object         10#dataPair
   instance    dataPair
   attributes
      attName     pickup_sectors
      attValue 'Down Town, University'
```

```
object         13#additionalData
   instance    additionalData
   attributes
      attributePair     [10#dataPair]
```

Figure 2. X-DEVICE representation of an XML document

4.2. The Rule Language of X-DEVICE

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 [9]. 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 [8] and [9]. The general algorithms for the translation of the various XML-aware constructs to first-order logic can be found in [9].

4.2.1. Rule Syntax

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, the following rule (actually it is a copy of rule `R1` of subsection 5.1.1) defines that an object exists in class `selected_web_service` with the attributes `request` with value `I`, `wservice` with value `WS` and `mservice` with value `MS`, if several conditions are satisfied. For example, one of the conditions states that the input SOAP message must contain an input attribute with name `address`. Furthermore, the URL address `MSA` associated with the incoming SOAP message must coincide with the `mediatorServiceAddress` attribute of a registered Mediator Service `MS`, whose name (`mediatorServiceName`) must be `'DispatchRequest'`.

```
if      I@input_soap_message(content:C) and
        not selected_web_service(request=I) and
        C@mediatorService(url=MSA,pair.inputVector:P) and
        P@pair(attName=address,attValue:PA)
        MS@registeredMediatorService(mediatorServiceName='DispatchRequest',
                mediatorServiceAddress=MSA) and
        WS@registeredWebService(mediatorService='DispatchRequest',
                webServiceAddress:WSA,dataPair.additionalData:DP) and
        DP@dataPair(attName=pickup_sectors,attValue:LOS) and
        prolog{belongs_to(PA,LOS)}
then    selected_web_service(request:I,wservice:WS,mservice:MS)
```

Class `selected_web_service` 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 (see for example rules `R3` and `R4` in subsection 5.1.2).

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 (`DP`) and simple values (`MSA`), 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 (see rule `R28` in subsection 6.2.1). 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 (see second condition element of the rule above).

Path expressions are composed using dots between the "steps", which are attributes of the interconnected objects, which represent XML document elements. For example, in the third condition element of the rule above the names of the input attributes are retrieved by navigating from the top-level `mediatorService` object-element to the `pair` attribute of an `inputVector` 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 [19]. Under this interpretation the chained "dotted" attributes can be seen as function compositions.

Finally, arbitrary conditions expressed as Prolog goals (e.g. `belongs_to/2`) can be placed at the end of the condition inside a `prolog{}` construct. More features of the X-DEVICE rule language will be presented and explained later, along with their usage for composing Web Services.

4.2.2. Rule Execution

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.

Subsequently the X-DEVICE system works in a hybrid event-driven/data-driven way. When new data arrives (e.g. new SOAP messages that are translated into objects using the mapping scheme of Section 4.1) and inserted in the database certain events are signaled. These events evoke rule processing by forwarding the data into the rule discrimination network that is responsible to find out which rule conditions are satisfied therefore the corresponding rule heads can be evaluated. However, only one rule can be evaluated/executed at each cycle. The rule ordering policy is stratification [27]. Rule evaluation actually constitutes the insertion of new data in the database, which may evoke (using the same event-driven/data-driven mechanism described above) another cycle of execution, until a new fixpoint is reached.

In certain occasions (e.g. cancellation messages) data may be deleted from the database. This again causes deletion-type events to signaled and notify the discrimination network that data has been deleted. The discrimination network finds all rules that their condition was satisfied before the deletion but is not satisfied after the deletion. These rules have been evaluated and inserted derived data that should no longer exist. These data are deleted from the database. Deletions may cause additional deletion cycles and the system continues until a fixpoint is reached. This is usually called *maintenance of materialized views* in database terms or *truth maintenance* in Artificial Intelligence terms.

Notice that since rules can have negation, the insertion of data may cause rule conditions that were true to become false if the inserted data involves the negated condition elements. In this case, the insertion of data can cause the deletion of derived data. On the other hand, if data are deleted that involve negated condition elements certain rules that could not be executed, now can. Therefore, the deletion of data may cause insertion of derived data. Examples of how negated condition elements behave during insertion and deletion of data can be found in section 6.2.1 (rules `R30` to `R33`).

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 [7]. Furthermore, set-oriented rule execution can be used for minimizing the number of inference cycles (and time) for large data sets [8].

5. Composing Simple Mediator Services

In this section we describe how X-DEVICE deductive rules are used in the SWIM system to compose Service Domains. More specifically, we describe use cases on composing simple Mediator Services by selecting Web Services and combining their results. The composition of complex Mediator Services is presented in the next section. The use of deductive rules for maintaining Service Domains and Mediator Services can be found in [10].

5.1. Selecting Web Services

The first task that a Mediator Service performs is to decide which of the registered Web Services are relative to an incoming SOAP request and then to select to which of them the request should be forwarded. Here we

present two such use cases; in the first case only one web service gets the message, which means that the mediator acts as a re-router; in the second case many web services get a copy of the message, which means that the mediator acts as a broadcaster. However, in order to complete the mediator's functionality one should also define the algorithm for constructing the result to be sent back to the requestor out of the responses of one or many web services. This will be tackled in the next subsection.

5.1.1. Selecting One Web Service

In order to demonstrate this use case, we assume a courier Web Service that accepts from clients SOAP messages requesting dispatching of parcels. Figure 3 shows the workflow of information in this use case, which follows the *exclusive choice* workflow pattern of [2].

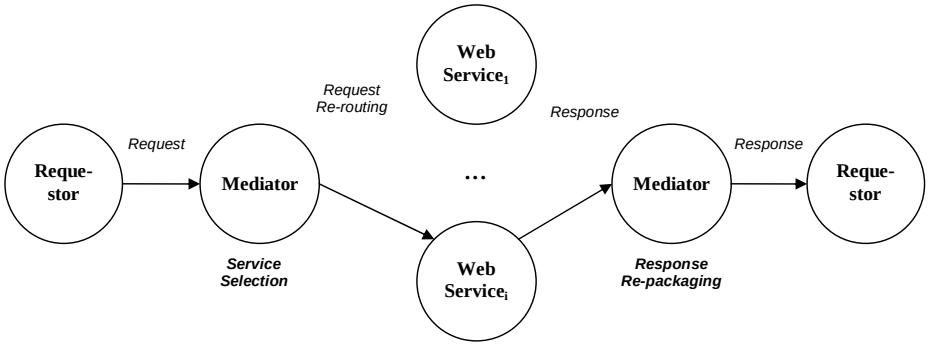

Figure 3. Mediator acting as a request re-router.

```
<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://XYZcourier.com/DispatchRequest.wsdl">
         <m0:inputVector>
            <m0:pair>
               <m0:attName>type</m0:attName>
               <m0:attValue>envelope</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>weight</m0:attName>
               <m0:attValue>1.1</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>destination</m0:attName>
               <m0:attValue>Orbanplein 8, B-1040 Brussels</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>address</m0:attName>
               <m0:attValue>Dept. of Informatics, Aristotle University</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>person</m0:attName>
               <m0:attValue>John Smith</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>delivery</m0:attName>
               <m0:attValue>2003-07-03T17:00:00+01:00</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>pickup</m0:attName>
               <m0:attValue>16:30:00+02:00</m0:attValue>
            </m0:pair>
         </m0:inputVector>
      </m:MediatorService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 4. Sample input SOAP message for the parcel dispatch request Mediator Service of the courier.

The input message contains information about parcel type, weight and destination, the pick-up address and details about the person that will hand-over the parcel. Furthermore, the client specifies constraints about the delivery date and the pick-up time. The Mediator Service of the courier company will assign the pick-up task to one of their local offices, based on the address of the client and will re-route the original message to the appropriate Web Service of the local office. The Web Service of the local office will calculate the price of the parcel, based on its type and weight and the client's delivery constraints. The local office Web Service will respond to the Mediator Service with the estimated parcel price and the expected pick-up time. Finally, the Mediator Service will re-package the response and it will forward it back to the client.

Figure 4 shows an example of a parcel dispatch request SOAP message. Input SOAP messages are stored within the X-DEVICE system using the schema for the SOAP message found in the corresponding WSDL description ([23]). 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`.

Rule `R1` performs the selection of the web service of the closest local office and creates a derived class `selected_web_service` whose instances are the selected Web Services. A local office can be selected if the input attribute `PA` with name `address` of the incoming SOAP message belongs to the sectors `LOS` of the local office. The user-defined Prolog predicate `belongs_to/2` succeeds if the string of its first argument represents an address that belongs to one of the sectors that its second string argument represents. We make no further assumptions on how such a predicate can be implemented as it is domain-dependent and outside the scope of this paper.

R1

```
if      I@input_soap_message(content:C) and
        not selected_web_service(request=I) and
        C@mediatorService(url=MSA,pair.inputVector:P) and
        P@pair(attName=address,attValue:PA)
        MS@registeredMediatorService(mediatorServiceName='DispatchRequest',
                mediatorServiceAddress=MSA) and
        WS@registeredWebService(mediatorService='DispatchRequest',
                webServiceAddress:WSA,dataPair.additionalData:AMV) and
        AMV@dataPair(attName=pickup_sectors,attValue:LOS) and
        prolog{belongs_to(PA,LOS)}
then    selected_web_service(request:I,wservice:WS,mservice:MS)
```

The information that is kept `selected_web_service` objects includes the ID of the original input SOAP message (`I`), which kept for correlation purposes, the ID of the selected Web Service (`WS`) and the ID of Mediator Service that the selected Web Service belongs to. The SOAP message ID is used by the rule condition (negated second condition element) to ensure that only one web service is selected for each dispatch request.

Rule `R1` is domain dependent because the metadata of the Mediator Service and the correlated Web Services are queried in order to ensure that this rule is only applicable for the parcel dispatch request use case. This is a general remark for all the X-DEVICE rules used in this paper; when a rule does not explicit refer to specific Mediator and/or Web Services, then the rule is applicable to any application context.

The address of the selected Web service is returned to the SWIM server along with the corresponding SOAP message that should be sent to the corresponding Web service of the SWIM nodes. Figure 5 shows such a message. Since in this use case the input SOAP message is just re-routed to the corresponding local office web service, the actual content of the message is identical to the message of Figure 4. Notice that the ID of the X-DEVICE object of the incoming SOAP message is also sent to the Web Service for correlation purposes.

The result is returned as an XML document and is calculated by domain independent rules `R5` to `R7` presented later (subsection 5.1.3). However, the leaves of the XML tree that contain the actual information can

only be constructed using domain dependent rules, since the information to be sent to the Web Services depend on the Mediator Service process logic.

```
<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://office1.XYZcourier.com/DispatchRequest.wsdl"
                    m:request="123#input_soap_message">
         <m0:inputVector>
         ...
         </m0:inputVector>
      </m:WebService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 5. Sample input SOAP message for the parcel dispatch request Web Service of a local courier office.

Rule `R2` copies all the attribute-value pairs of the original parcel dispatch request message to new `pair` objects augmented with the ID of the original request and the ID of each selected Web Service (in this case only one). Furthermore, the rule queries the metadata of the Mediator Service to ensure that this rule is only applicable for the parcel dispatch request Mediator Service, i.e. rule `R2` is domain dependent. 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 XML result document.

R2

```
if      SWS@selected_web_service(request:R,wservice:WS,mservice:MS) and
        MS@registeredMediatorService(
                mediatorServiceName='DispatchRequest') and
        R@input_soap_message(content:C) and
        C@mediatorService(pair.inputVector:P) and
        P@pair(attName:AttName,attValue:AttVal)
then    pair(!request:R,!service:WS,attName:AttName,attValue:AttVal)
```

5.1.2. Selecting Many Web Services

Extending the previous courier company example, we now assume that the courier company has external transportation partners (e.g. airlines, truck lines) that transport the parcels and that these partners can send SOAP messages requesting the total weight of parcels to be transported to a certain destination (e.g. USA) in order to schedule their daily cargos. Figure 6 shows the workflow of information in this use case, which follows the *multi choice* workflow pattern of [2]. The input message contains information about destination. The Mediator Service of the courier company will broadcast a similar SOAP message to its local office Web Services and will wait for all the answers to come back.

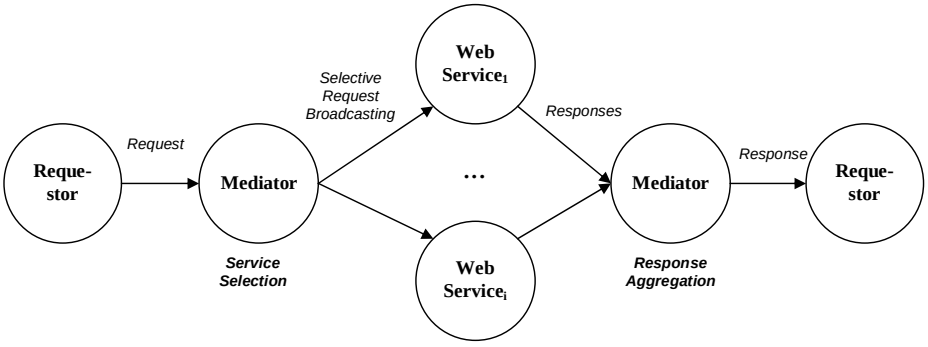

Figure 6. Mediator acting as an information integrator.

```
<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://XYZcourier.com/TotalWeight.wsdl">
         <m0:inputVector>
            <m0:pair>
               <m0:attName>destination</m0:attName>
               <m0:attValue>USA</m0:attValue>
            </m0:pair>
         </m0:inputVector>
      </m:MediatorService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 7. Sample input SOAP message for the total dispatch weight Mediator Service of the courier.

To make things a little more interesting we assume that not all local offices collect parcels for international dispatches, therefore only the web services of the "international" local offices need to be queried for an international destination. On the other hand, when the destination is "national", all local office Web Services need to be queried. The information about the type of local office is kept along its metadata ([23]). Notice that non-international offices still pickup "international" parcels from clients of their responsibility sector but they deliver them in batches during the day to their closest "international" office. This is the reason why the parcel dispatch services of the previous section need not to be altered because of this extension.

The Web Services of a local office will calculate the daily total weight of parcels that are to be dispatched to the specific destination and will return the result back to the Mediator Service. Finally, the Mediator Service will aggregate the responses of the Web Services by summing up the total weight and will return back to the airline client a single response. Figure 7 shows an example of a total dispatch weight request SOAP message.

In order to implement the complex web service selection algorithm, two rules are needed. The first one, rule `R3`, performs the selection of the web services of all the "international" local offices when the destination is also international (user-defined Prolog predicate `is_international/1`). In contrast with the parcel dispatch request Mediator Service (rule `R1`), multiple web services can be selected. The second rule `R4` is complimentary to rule `R3` and selects all the local office web services when the destination is not "international".

R3

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url=MSA,pair.inputVector:P) and
        P@pair(attName=destination,attValue:D)
        MS@registeredMediatorService(mediatorServiceName='TotalWeight',
                mediatorServiceAddress=MSA) and
        WS@registeredWebService(mediatorService='TotalWeight',
                webServiceAddress:WSA,dataPair.additionalData:AMV) and
        AMV@dataPair(attName=office_type,attValue='International') and
        prolog{is_international(D)}
then    selected_web_service(request:I,wservice:WS,mservice:MS)
```

R4

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url=MSA,pair.inputVector:P) and
        P@pair(attName=destination,attValue:D)
        MS@registeredMediatorService(mediatorServiceName='TotalWeight',
                mediatorServiceAddress=MSA) and
        WS@registeredWebService(mediatorService='TotalWeight',
                webServiceAddress:WSA) and
        prolog{not(is_international(D))}
then    selected_web_service(request:I,wservice:WS,mservice:MS)
```

The SOAP message that is broadcasted to the selected web services is almost identical to the one in Figure 7 and is constructed by a domain dependent rule similar to `R2` and the domain independent rules `R5` to `R7` that will

be presented in the next subsection. The only modification to rule R2 is the reference to the name of the Mediator Service that should become 'TotalWeight'.

5.1.3. Constructing Output SOAP Messages for Selected Web Services

After one or more Web Services have been selected to service the Mediator Service, the SOAP message to be sent to them must be constructed. The structure of the XML tree of these SOAP messages is similar for all Mediator Services regardless of their nature (see [23]). The XML tree is build up in a bottom up manner using rules R5 to R7, which are domain independent, therefore they can be used for all use cases presented in this paper.

Rule R5 constructs an inputVector object for each selected Web Service by linking it with the corresponding pair objects, assuming that such objects have been created by other (domain dependent) rules, as we have seen previous subsections. 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 (namely R and WS) define a GROUP BY operation. More details about the implementation of aggregate functions in X-DEVICE can be found in [8] and [9].

Rule R6 constructs a webService object and links it to the single input vector (per selected Web Service and request) that has been constructed by rule R5. Furthermore, the webService object has an XML attribute ^request, designated by the use of the (^) symbol in front of an attribute name that correlates this message to the original input message.

Rule R7 constructs the top-level XML element of the result linking it with the webService object that has been constructed by rule R6, augmented with the address of the Web Service that will be used to create the namespace context of the message and to send the message to this address. 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. The constructed output messages are picked up by the SWIM server and are sent to the corresponding Web Services.

R5

```
if      SWS@selected_web_service(request:R,wservice:WS) and
        P@pair(request=R,service=WS)
then    inputVector(!request:R,!service:WS,pair:list(P))
```

R6

```
if      SWS@selected_web_service(request:R,wservice:WS) and
        IV@inputVector(request=R,service=WS)
then    webService(^request:R,!service:WS,inputVector:IV)
```

R7

```
if      SWS@selected_web_service(request:R,wservice:WS) and
        WS1@webService(request=R,service=WS) and
        WS@registeredWebService(webServiceAddress:URL)
then    xml_result(output_soap_message(!address:URL,content:WS1))
```

5.2. Combining the Results of Web Services

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 is also 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 or just to re-package the answer from a single Web service, depending on the use case. In this subsection, we continue using the examples of the previous subsection.

5.2.1. Synchronizing Web Services

The SWIM system offers a *wait-for-all* synchronization construct, which is called just *synchronization* in [2], in order to start processing Web Service answers only after all of them have replied. This synchronization construct is independent from the application context and is implemented by rules `R8` to `R10`. More specifically, rule `R8` counts the number of selected Web Services in order to compare it to the number of Web Services that responded, which are counted by rule `R9`. Notice that the correlation of the response messages to the original input message is done through the ID of the original message (`R`) that was kept in the `request` attribute. The `count(WS)` construct in the conclusion of rule `R8` (and `R9`) denotes that the attribute `questions` of the derived class `ws_asked` is an attribute whose value is calculated by the aggregate function `count` that counts the number of instantiations of the variable `WS`.

Rule `R10` performs the comparison and derives an `all_ws_answered` object that is used by other rules in order to proceed constructing the answer to the original input message only when all Web Services have responded.

R8

```
if      SWB@selected_web_service(request:R,wservice:WS)
then    ws_asked(request:R,questions:count(WS))
```

R9

```
if      I@input_soap_message(content:C) and
        C@webService(url:WSA,request:R) and
        WS@registeredWebService(webServiceAddress=WSA)
        SWB@selected_web_service(request=R,wservice=WS)
then    ws_answered(request:R,answers:count(WS))
```

R10

```
if      W1@ws_asked(request:R,questions:N) and
        W2@ws_answered(request=R,answers=N)
then    all_ws_answered(request:R)
```

Support for Alternative Workflow Patterns

More synchronization constructs (or workflow patterns) are supported either domain-independently such as the *wait-for-all* synchronization that was presented here, or domain-dependently. The rules that implement the construct generate a “flag” object as soon as the processing of the Web Service results can start. This object is used in the conditions of the rest of the rules as a "flag", i.e. when it exists rules apply otherwise rules cannot be executed.

For example, the *simple merge* workflow pattern [2] is implemented by replacing rule `R10` with rule `R11`, which generates an ok “flag” when just one message is received regardless of how many messages will eventually arrive.

R11

```
if      W@ws_answered(request:R,answers=1)
then    simple_merge_ok(request:R)
```

Furthermore, the *synchronizing merge* workflow pattern is already covered by rules `R8` to `R10` because in rule `R10` if N=1 then only one Web Service has been asked, therefore only one answer is expected and when it arrives, execution will continue. If one the other hand N>1, then all the answers will be expected. Other workflow patterns, such as *Discriminator* or *N-out-of-M Join*, are supported if we replace domain-independent rules `R8` to `R10` above with domain-dependent rules whose conditions decide if the ok flag can be generated

depending not only on the number of received answers but also on their content. In the rest of the paper we only use the wait-for-all synchronization, but its use is clearly marked so it can be replaced by other synchronization constructs easily.

5.2.2. Response Re-packaging

In the parcel dispatch request use case, the local office web service that has been selected to dispatch the parcel responds to the Mediator Service with the estimated delivery price and pickup time. Figure 8 presents such a SOAP message. This message must be re-packaged and returned to the original requestor in the form of Figure 9. The parts of the message that are similar to the one in Figure 8 are marked with dots.

Concerning the construction of the response, rule `R12` copies all the attribute-value pairs of the single local office Web Service response to new `pair` objects augmented with the ID of the original request. Furthermore, the rule traces the original input message of the Mediator Service and uses its URL address in order to query the metadata of the Mediator Service and ensure that this rule is only applicable for the parcel dispatch request Mediator Service, i.e. rule `R12` is domain dependent. Notice that the `all_ws_answered` object is used to start constructing the XML tree of the output SOAP message only when all Web Services (in this case only one) have responded. The rest of the XML tree is constructed by the domain independent rules `R15` to `R17` that will be presented later (subsection 5.2.4).

`R12`

```
if      A@all_ws_answered(request:R) and
        R@input_soap_message(content:C1) and
        C1@mediatorService(url:MSA) and
        MS@registeredMediatorService(mediatorServiceName='DispatchRequest',
                mediatorServiceAddress=MSA) and
        C@webService(request=R,pair.outputVector:P) and
        P@pair(attName:AttName,attValue:AttVal)
then    pair(!request:R,attName:AttName,attValue:AttVal)
```

```
<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://office1.XYZcourier.com/DispatchRequest.wsdl"
                    m:request="123#input_soap_message">
         <m0:outputVector>
            <m0:pair>
               <m0:attName>price</m0:attName>
               <m0:attValue>60.18</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>pickup</m0:attName>
               <m0:attValue>15:30:00+02:00</m0:attValue>
            </m0:pair>
         </m0:outputVector>
      </m:WebService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 8. Sample output SOAP message from the parcel dispatch request Web Service of a local courier office.

```
<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://XYZcourier.com/DispatchRequest.wsdl"
                         m:request="123#input_soap_message">
         <m0:outputVector>
            ...
         </m0:outputVector>
      </m:MediatorService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 9. Sample output SOAP message from the parcel dispatch request Mediator Service of the courier.

5.2.3. Response Aggregation

In the total dispatch weight request use case, each selected local office web service responds to the Mediator Service with its daily total dispatch weight destined to the specified destination. Figure 10 presents such a SOAP message. The numerical contents of all these messages must be added to a single figure which will be returned to the original requestor in the form of Figure 11.

```
<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://office1.XYZcourier.com/TotalWeight.wsdl"
                    m:request="456#input_soap_message">
         <m0:outputVector>
            <m0:pair>
               <m0:attName>weight</m0:attName>
               <m0:attValue>321</m0:attValue>
            </m0:pair>
         </m0:outputVector>
      </m:WebService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 10. Sample output SOAP message from the total dispatch weight request Web Service of a local office.

```
<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://XYZcourier.com/TotalWeight.wsdl"
                         m:request="456#input_soap_message">
         <m0:outputVector>
            <m0:pair>
               <m0:attName>weight</m0:attName>
               <m0:attValue>1765</m0:attValue>
            </m0:pair>
         </m0:outputVector>
      </m:MediatorService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 11. Sample output SOAP message from the total dispatch weight request Mediator Service of the courier.

Rules `R13` and `R14` implement the response aggregation algorithm. Rule `R13` iterates all the answers, collects the weights reported by each local office (instantiations of variable `W`) and sums them up using the `sum` aggregation function. The result is stored at the single (per request) instance of the derived class `total_weight`. Rule `R13` applies only to the specific use case because the metadata of the total dispatch weight request Mediator Service is explicitly mentioned. Rule `R14` constructs the single `pair` object that will be part of the XML tree of the result document by copying the `weight` value of the single `total_weight` object. This rule is domain dependent since it relies on the existence `total_weight` objects which are constructed by the domain dependent rule `R13`. The rest of the XML tree is constructed by the domain independent rules `R15` to `R17` that will be presented in the following subsection.

R13

```
if      A@all_ws_answered(request:R) and
        R@input_soap_message(content:C1) and
        C1@mediatorService(url:MSA) and
        MS@registeredMediatorService(mediatorServiceName='TotalWeight',
                mediatorServiceAddress=MSA) and
        C@webService(request=R,pair.outputVector:P) and
        P@pair(attName=weight,attValue:W)
then    total_weight(request:R,weight:sum(W))
```

R14

```
if      A@all_ws_answered(request:R) and
```

```
        TW@total_weight(request=R,weight:W)
then    pair(!request:R,attName:weight,attValue:W)
```

5.2.4. Constructing the SOAP Message for the Response of the Mediator Service

The structure of the XML tree of the result document (output SOAP message) is similar for all Mediator Services regardless of their nature (see [23]). The XML tree is build up in a bottom up manner using rules `R15` to `R17`.

Rule `R15` constructs an `outputVector` object for each request that all its Web Services have responded by linking it with the corresponding `pair` objects, assuming that such objects have been created by other (domain dependent) rules, as we will have seen in the previous subsections.

Rule `R16` constructs a `mediatorService` object and links it to the single output vector that has been constructed by rule `R15`. Furthermore, the `mediatorService` object returns to the client the ID of the original input message as an XML attribute `^request`. This may seem unnecessary when the client is an external system, unaware of the internal IDs of the SWIM system. Such clients may simply ignore this extra XML attribute. However, if the client of the Mediator Service is another (complex) Mediator Service of the SWIM system (see section 6) then this attribute is used to correlate this output message to the original input message of the complex Mediator Service.

Finally, rule `R17` constructs the top-level element for the output SOAP message, linking it with the `mediatorService` object that has been constructed by rule `R16`, augmented with the address of the Mediator Service that will be used to create the namespace context of the message. The URL address of the Mediator Service is discovered by tracing back the original input message of the Mediator Service. The constructed output message is picked up by the SWIM server and is sent to the original requester of the Mediator Service.

Here we notice that rules `R15` to `R17` are domain independent, i.e. they apply regardless of the nature of the Mediator Service, which means that they are used for all use cases presented in this paper. Furthermore, all rules use the `all_ws_answered` object for two reasons; the first reason is to start executing only when all selected Web Services have answered; the second reason is to be able to link all parts of the XML tree to the ID of the original request message.

R15

```
if      A@all_ws_answered(request:R) and
        P@pair(request=R)
then    outputVector(!request:R,pair:list(P))
```

R16

```
if      A@all_ws_answered(request:R) and
        OV@outputVector(request=R)
then    mediatorService(^request:R,outputVector:OV)
```

R17

```
if      A@all_ws_answered(request:R) and
        MS@mediatorService(request=R) and
        R@input_soap_message(content:C1) and
        C1@mediatorService(url:MSA)
then    xml_result(output_soap_message(!address:MSA,content:MS))
```

6. Composing Complex Mediator Services

In this section we present how complex Mediator Services are composed by combining simple Mediator Services. As a use case, we further extend the courier example. We, now, assume that there is a third-party courier service e-mall that accepts parcel dispatch request messages from clients, selects the cheapest and/or fastest of the courier companies regarding the specific delivery and re-routes the original dispatch request to the selected courier company. The above process is depicted in Figure 12. The Mediator Service of this complex

process uses two other Mediator Services (called *Auxiliary Mediator Services*) to achieve its goal; the first auxiliary Mediator Service selects the cheapest/speediest delivery offer from the courier companies, while the second one requests a parcel dispatch from a specific courier company, whose name is the result of the first Mediator Service.

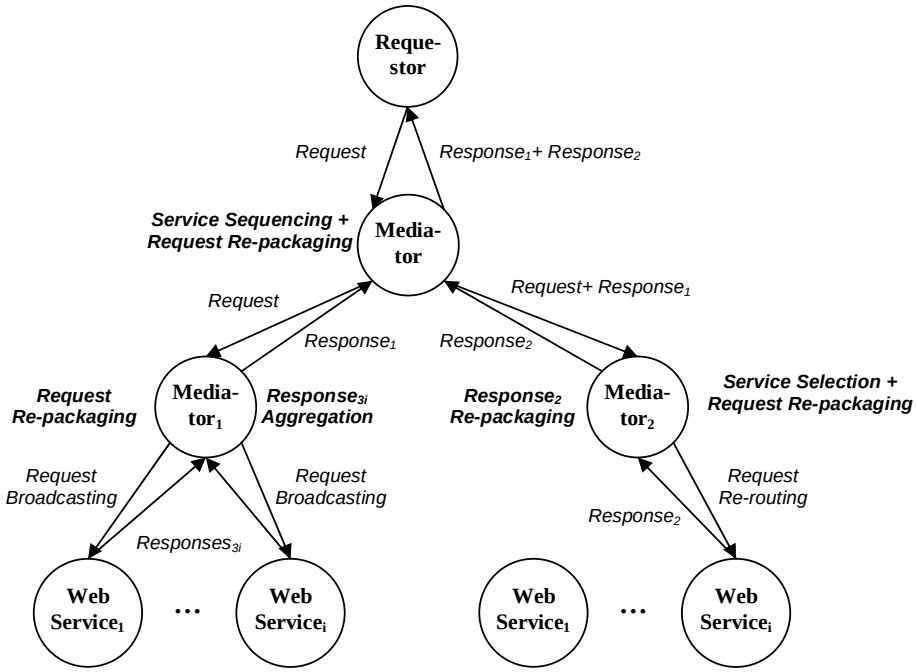

Figure 12. A Complex Mediator Service.

```
<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://courier.org/CheapDispatchRequest.wsdl">
         <m0:inputVector>
         ...
         </m0:inputVector>
      </m:MediatorService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 13. Sample input SOAP message for the cheapest/speediest parcel dispatch request Mediator Service of the courier e-mall.

```
<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://XYZcourier.com/OfferRequest.wsdl"
                    m:request="1111#input_soap_message">
         <m0:outputVector>
            <m0:pair>
               <m0:attName>price</m0:attName>
               <m0:attValue>147.59</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>delivery_date</m0:attName>
               <m0:attValue>2003-08-13T12:00:00-06:00</m0:attValue>
            </m0:pair>
         </m0:outputVector>
      </m:WebService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 14. Sample output SOAP message from the parcel delivery offer request Web Service of a courier company.

The input of the complex Mediator Service is the same with the single company dispatch request service (section 5.1.1). Figure 13 shows such a message, with dots in place of descriptions that are exactly the same with Figure 4. The first Mediator Service also has the same input, but its output is the name of the courier company that offers the cheapest and quicker delivery for this specific parcel. This Mediator Service broadcasts to all courier company Web Services a copy of the input message and gets back as an answer the price of the delivery and the expected delivery date (Figure 14). Then, the mediator service selects the cheapest offer and among them the one with the quicker delivery. If more than one such offer exists, then the Mediator Service selects one at random and returns the name of the company that made this offer along with the corresponding price and date of the delivery (Figure 15) to the caller, which in this case is the complex Mediator Service.

In the sequel, the complex Mediator Service forwards the original parcel dispatch message to the second mediator service, augmented with the name of the selected courier company. Figure 16 shows such a message, with dots in place of descriptions that are exactly the same with Figure 4. This Mediator Service just re-packages the original parcel dispatch message (excluding the company's name) and forwards it to the Web Service of the company; the selection is naturally based on the name of the company. The parcel dispatch Web Services of each company are exactly the same with the ones presented in section 5.1.1. Notice that some of the companies may actually use the SWIM system, therefore their re-action to such a message will follow the procedure mentioned in section 5.1.1. However, even in this case the parcel dispatch services of the company will be registered to the e-mall's SWIM system as normal Web Services and not as Mediator Services.

```
<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://courier.org/CheapDeliveryOffer.wsdl"
                         m:request="789#input_soap_message">
         <m0:outputVector>
            <m0:pair>
               <m0:attName>organization</m0:attName>
               <m0:attValue>XYZ Courier Company</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>price</m0:attName>
               <m0:attValue>147.59</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>delivery_date</m0:attName>
               <m0:attValue>2003-08-13T12:00:00-06:00</m0:attValue>
            </m0:pair>
         </m0:outputVector>
      </m:MediatorService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 15. Sample output SOAP message from the cheapest/speediest delivery offer request Mediator Service of the courier e-mall.

```
<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://courier.org/CompanyDispatchRequest.wsdl">
         <m0:inputVector>
         ...
            <m0:pair>
               <m0:attName>organization</m0:attName>
               <m0:attValue>XYZ Courier Company</m0:attValue>
            </m0:pair>
         </m0:inputVector>
      </m:MediatorService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 16. Sample input SOAP message for the parcel dispatch request for a specific courier company Mediator Service of the courier e-mall.

Finally, when the company's Web Service responds with the confirmed price and the expected parcel pick-up time (see e.g. Figure 8) the second Mediator Service will re-package the answer and will forward it to the complex mediator Service. The latter will re-package the answer again, augmenting it with the courier company's name and the delivery date and it will return it back to the client (Figure 17). Notice that even if the three Mediator Services reside at the same SWIM system, they exchange messages normally through the SOAP protocol and not via an internal protocol. This offers transparency and facilitates the migration of some of the services to another remote system.

In the following subsections we present in detail the internal processes of the two auxiliary mediator services and the complex mediator service using X-DEVICE rules. Notice that the two auxiliary Mediator Services are stand-alone, i.e. they function independently of the complex Mediator Service.

```
<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://courier.org/CheapDispatchRequest.wsdl"
                         m:request="777#input_soap_message">
         <m0:outputVector>
            <m0:pair>
               <m0:attName>price</m0:attName>
               <m0:attValue>147.59</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>pickup</m0:attName>
               <m0:attValue>15:30:00+02:00</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>delivery_date</m0:attName>
               <m0:attValue>2003-08-13T12:00:00-06:00</m0:attValue>
            </m0:pair>
            <m0:pair>
               <m0:attName>organization</m0:attName>
               <m0:attValue>XYZ Courier Company</m0:attValue>
            </m0:pair>
         </m0:outputVector>
      </m:MediatorService>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 17. Sample output SOAP message from the cheapest/speediest parcel dispatch request Mediator Service of the courier e-mall.

6.1. Auxiliary Mediator Services

6.1.1. Cheapest/Speediest Delivery Offer Request

The first Mediator Service does not require selection of Web Services to broadcast the original dispatch message. Furthermore, the original message does not require any transformations but only needs re-packaging. This workflow pattern is called *parallel split* in [2]. Rule `R18` iterates over all company delivery offer web services that have subscribed for the cheapest/speediest delivery offer Mediator Service and marks them as "selected". The SOAP message that is broadcasted to the selected web services is constructed by a rule similar to rule `R2` (section 5.1.1) by changing the name of the Mediator Service to `'CheapDeliveryOffer'` and rules `R5` to `R7` that have been presented earlier (section 5.1.3).

R18

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA) and
        MS@registeredMediatorService(mediatorServiceAddress=MSA,
                mediatorServiceName='CheapDeliveryOffer') and
        WS@registeredWebService(mediatorService='CheapDeliveryOffer',
                webServiceAddress:WSA)
```

```
then    selected_web_service(request:I,wservice:WS,mservice:MS)
```

Rules R19 to R25 implement the response aggregation algorithm of this Mediator Service. Rule R19 iterates all the answers, collects the price offers of each courier company (instantiations of variable PR) and keeps the minimum of them using the min aggregation function. The result is stored at the single (per request) instance of the derived class cheapest_price. Rule R19 ensures that the original request was for the specific Mediator Service, therefore it is domain dependent. Since rule R19 cannot keep the IDs of the minimum offer responses but only the value of the minimum offer, rule R20 is needed, which iterates again all the answers and keeps the IDs of all the Web Service responses that have an offer equal to the minimum one, as cheapest_price_ws objects.

Rule R21 iterates all the minimum price offer answers, collects the delivery dates of these offers (instantiations of variable D) and keeps the minimum of them using the min aggregation function. The result is stored at the single (per request) instance of the derived class quickest_delivery. Again, rule R21 cannot keep the IDs of the quickest delivery responses but only the value of the quickest delivery, therefore rule R22 iterates again the minimum price offer answers and keeps the IDs of all the Web Service responses that have a delivery date equal to the quickest one, as best_offer_ws objects. Notice that even if many offers exist with the same minimum price and quickest delivery date only one of them is selected, because of the negated last condition element of rule R22; the rule is applicable only if a best offer web service has not yet been recorded.

The last three rules R23 to R25 construct three pair objects which are the leaves of the output XML tree. Rule R23 traces the metadata of the best offer web service in order to include the organization that owns the web service in the result. Rule R24 include in the result the cheapest delivery price and quickest delivery date, respectively. The final output SOAP message is constructed by rules R15 to R17 that have been presented earlier (section 5.2.4).

R19

```
if      A@all_web_services_answered(request:R) and
        C@webService(request=R,pair.outputVector:P) and
        R@input_soap_message(content:C1) and
        C1@mediatorService(url:MSA) and
        MS@registeredMediatorService(mediatorServiceAddress=MSA,
                mediatorServiceName='CheapDeliveryOffer') and
        P@pair(attName:price,attValue:PR)
then    cheapest_price(request:R,price:min(PR))
```

R20

```
if      A@all_web_services_answered(request:R) and
        CP@cheapest_price(request=R,price:PR) and
        I@input_soap_message(content:C) and
        C@webService(request=R,pair.outputVector:P) and
        P@pair(attName:price,attValue=PR)
then    cheapest_price_ws(request:R,price:PR,webService:C)
```

R21

```
if      A@all_web_services_answered(request:R) and
        X@cheapest_price_ws(request=R,webService:C) and
        C@webService(pair.outputVector:P) and
        P@pair(attName:delivery_date,attValue:D)
then    quickest_delivery(request:R,delivery:min(D))
```

R22

```
if      A@all_web_services_answered(request:R) and
        QD@quickest_delivery(request=R,delivery:D) and
        X@cheapest_price_ws(request=R,price:PR,webService:C) and
        C@webService(pair.outputVector:P) and
        P@pair(attName:delivery_date,attValue=D) and
        not B@best_offer_ws(request=R)
```

```
then    best_offer_ws(request:R,price:PR,delivery_date:D,webService:C)
```

R23

```
if      A@all_web_services_answered(request:R) and
        B@best_offer_ws(request:R,webService:C) and
        C@webService(url:WSA) and
        WS@registeredWebService(webServiceAddress=WSA,organization:O)
then    pair(!request:R,attName:organization,attValue:O)
```

R24

```
if      A@all_web_services_answered(request:R) and
        B@best_offer_ws(request:R,price:PR)
then    pair(!request:R,attName:price,attValue:PR)
```

R25

```
if      A@all_web_services_answered(request:R) and
        B@best_offer_ws(request:R,delivery_date:D)
then    pair(!request:R,attName:delivery_date,attValue:D)
```

6.1.2. Specific Courier Company Parcel Dispatch Request

The second Mediator Service selects a single Web Service to re-route the original dispatch message based on the name of the requested courier company. This workflow pattern is called the *exclusive choice* pattern in [2]. The only transformation required for the original message is to exclude the name of the company; the rest of the input attributes are re-packaged unchanged. Rule `R26` matches the value of the `organization` attribute of the input message with the corresponding attribute of the courier companies parcel dispatch request web services that have subscribed for the specific Mediator Service and normally selects only one of them. The SOAP message that is re-routed to the selected web service is constructed by rule `R27` which is quite similar to rule `R2` (section 5.1.1). Their differences are the name of the Mediator Service (`'CompanyDispatchRequest'`) and the condition `attName:AttName\=organization` that excludes the `organization` attribute from the generation of the `pair` objects. The rest of the XML tree of the SOAP message is constructed by rules `R5` to `R7` that have been presented earlier (section 5.1.3).

R26

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA,pair.inputVector:P) and
        P@pair(attName=organization,attValue:O) and
        MS@registeredMediatorService(mediatorServiceAddress=MSA,
                mediatorServiceName='CompanyDispatchRequest') and
        WS@registeredWebService(mediatorService='CompanyDispatchRequest',
                organization=O,webServiceAddress:WSA)
then    selected_web_service(request:I,wservice:WS,mservice:MS)
```

R27

```
if      SWS@selected_web_service(request:R,wservice:WS,mservice:MS) and
        MS@registeredMediatorService(
                mediatorServiceName='CompanyDispatchRequest') and
        R@input_soap_message(content:C) and
        C@mediatorService(pair.inputVector:P) and
        P@pair(attName:AttName\=organization,attValue:AttVal)
then    pair(!request:R,!service:WS,attName:AttName,attValue:AttVal)
```

The response that will be returned by the single selected Web Service is re-packaged and returned unchanged to the client. The domain-dependent rule that performs this re-packaging is similar to rule `R12` except for the name of the Mediator Service. The rest of the response is constructed by the domain independent rules `R15` to `R17` that have been presented earlier (subsection 5.2.4).

6.2. Complex Mediator Service

In this subsection we describe how the two auxiliary Mediator Services are combined to form the complex Mediator Service. There are two aspects to service composition; how services are synchronized and how their results are combined, transformed and transferred from one service to another. In the following we describe how auxiliary service sequencing is achieved in X-DEVICE, independently of the application context, whereas result combination and transfer is being described using the specific courier example, since it is a highly application dependent process.

6.2.1. Synchronizing Auxiliary Mediator Service Sequences

In this subsection we describe how the execution of auxiliary Mediator Services is synchronized as a *sequence* ([2]) within a complex Mediator Service, without taking into account the application context. The details about how the specific complex Mediator Service example achieves the automatic parcel dispatch request using the cheapest/speediest delivery offer are presented in the next subsection.

First of all, complex Mediator Services are registered in the SWIM system in a slightly different manner than normal Mediator Services. In [23] all the details about the complex Mediator Service metadata, in general, and the metadata for the specific example can be found. Complex Mediator Services have an additional element `auxiliaryServices` that hosts the sequence of names of the auxiliary Mediator Services that they utilize, under a strict order.

When a complex Mediator Service receives a request, rules `R28` and `R29` are responsible to iterate the auxiliary Mediator Services sequence and create pairs of neighboring Mediator Services in the sequence, as `ms_sequence` objects. These objects keep the IDs of the complex Mediator Service (`complex_ms`), and of any two neighboring auxiliary Mediator Services in the sequence (`first_ms`, `second_ms`). If a complex Mediator Service is implemented by a sequence of *n* auxiliary Mediator Services, then *n* such objects will be created. However, the last `ms_sequence` object that has as its first Mediator Service the last Mediator Service of the sequence will not have a second Mediator Service object to point to, because rule `R29` will not fire for it.

Getting into the details, rule `R28` generates `ms_sequence` objects for all auxiliary Mediator Services, putting them as the first service `MS1` of the sequence. Notice that the name `MSN1` of the first service of the sequence is also kept in order to be used in rule `R29`. Rule `R29` is a derived attribute rule, which defines a new attribute `second_ms` for class `ms_sequence`. The values for this attribute are derived by this rule. Objects of class `ms_sequence` that do not satisfy the condition of this rule will have null value for this attribute. More details on derived attribute rules can be found in [8].

Rule `R29` also exhibits one more interesting feature of X-DEVICE, namely ordering expressions, i.e. expressions that query an XML tree based on the ordering of elements. The expression `mediatorServiceName.auxiliaryServices` $\ni_{\{after(MSN1),=<1\}}$ `MSN2` denotes that `MSN2` is a value stored in the multi-valued attribute `mediatorServiceName` of an `auxiliaryServices` object immediately after the value of variable `MSN1`, which is the name of the first auxiliary service of the sequence. More details on the ordering expressions of X-DEVICE can be found in [9].

R28

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA1) and
        MS@complexMediatorService(mediatorServiceAddress=MSA,
                mediatorServiceName.auxiliaryServices ∋ MSN1) and
        MS1@registeredMediatorService(mediatorService=MSN1)
then    ms_sequence(complex_ms:MS,first_ms:MS1,first_n:MSN1)
```

R29

```
if      SEQ@ms_sequence(complex_ms:MS,first_n:MSN1) and
```

```
        MS@complexMediatorService(
                mediatorServiceName.auxiliaryServices ∋{after(MSN1),=<1} MSN2) and
        MS2@registeredMediatorService(mediatorService=MSN2)
then    SEQ@ms_sequence(second_ms:MS2)
```

The ms_sequence objects are used for synchronization purposes. Specifically, each auxiliary mediator service in the sequence will be selected for sending it an input message only if the previous service in the sequence has responded (rules R30 and R31). Rule R30 starts by selecting the first service in the sequence MS1, which occurs in a sequence pair as the first mediator service but there is no sequence pair having MS1 as its second service. The class selected_mediator_service plays a role similar to the class selected_web_service. The half_sequence objects indicate that the response for the first mediator service of a sequence in the context of a specific request has been received. Therefore, the negation in rule R30 denotes that the first service in the sequence will be selected only if the answer for it has not been received yet.

Rule R31 always selects the second service (MS2) of an ms_sequence pair when the first service of the sequence has sent a response (presence of related half_sequence object) but the second service has not yet sent a response (absence of related full_sequence object). The derivation of half_sequence and full_sequence objects will be described later. Notice that rule R31 will not fire for the last ms_sequence object because the second_ms attribute of this object is not instantiated. However, the last auxiliary service of the sequence will be selected due to the penultimate ms_sequence object whose second_ms service is the last service of the sequence.

R30

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA) and
        MS@complexMediatorService(mediatorServiceAddress=MSA) and
        SEQ@ms_sequence(complex_ms=MS,first_ms:MS1) and
        not SEQ1@ms_sequence(complex_ms=MS,second_ms=MS1) and
        not HS@half_sequence(request=I,sequence=SEQ)
then    selected_mediator_service(request:I,mservice:MS1,cservice:MS)
```

R31

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA) and
        MS@complexMediatorService(mediatorServiceAddress=MSA) and
        SEQ@ms_sequence(complex_ms=MS,first_ms:MS1,second_ms:MS2) and
        HS@half_sequence(request=I,sequence=SEQ) and
        not FS@full_sequence(request=I,sequence=SEQ)
then    selected_mediator_service(request:I,mservice:MS2,cservice:MS)
```

When the SWIM system receives answers from the auxiliary mediator services, half_sequence and/or full_sequence objects are derived to indicate the partial or full completion of a sequence pair. Rule R32 derives a half_sequence object when a message is received for the first mediator service of an ms_sequence pair. Rule R33 derives a full_sequence object when a message is received for the second mediator service of an ms_sequence pair and a half_sequence object for the same sequence pair already exists.

All but the first auxiliary services of a sequence participate in two ms_sequence objects; in the first one they are the second service of the pair, while in the second one they are the first service of the pair. Naturally, both rules R32 and R33 will fire for these services; completing the previous sequence and commencing the next one. However, the last ms_sequence object that has the last service of the sequence as its first service will not satisfy rule R33 because there is no second service in this object.

R32

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA1,request:R) and
```

```
        MS1@registeredMediatorService(mediatorServiceAddress=MSA1) and
        SMS@selected_mediator_service(request=R,mservice=MS1,
                cservice:MS) and
        SEQ@ms_sequence(complex_ms=MS,first_ms=MS1)
then    half_sequence(request:R,sequence:SEQ)
```

R33

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA2,request:R) and
        MS2@registeredMediatorService(mediatorServiceAddress=MSA2) and
        SMS@selected_mediator_service(request=R,mservice=MS2,
                cservice:MS) and
        SEQ@ms_sequence(complex_ms=MS,second_ms=MS2) and
        HS@half_sequence(request=R,sequence=SEQ)
then    full_sequence(request:R,sequence:SEQ)
```

We notice here that the combination of rules `R30` to `R33` have as effect that only one mediator service is selected at any time, i.e. that only one `selected_mediator_service` object exists at any time. This is explained as follows: in the general case, rule `R31` will select a Mediator Service if a `half_sequence` object exists but a `full_sequence` object does not exist. After this selected service responds rule `R33` generates a `full_sequence` object. Due to the truth maintenance capabilities of the X-DEVICE's inference engine (section 4.2.2) the condition of rule `R31` is no longer satisfied for the selected Mediator Service, therefore the corresponding `selected_mediator_service` object must be deleted. In the meantime, rule `R32` has created a new `half_sequence` object that will cause `R31` to be evaluated again for the next service in the sequence and to generate a new `selected_mediator_service` object, and so on so forth.

When all Mediator Services in the sequence have been activated and responded no `selected_mediator_service` object exists. In order to decide if the sequence has terminated we must use a different line of reasoning than we did in section 5.2.1 for simple Mediator Services. Specifically, we notice that rule `R32` will fire for all auxiliary services of the sequence; therefore, eventually there will be as many `half_sequence` objects as the `ms_sequence` objects, i.e. equal to the number of auxiliary services. When this occurs, the sequence has terminated and the construction of the output SOAP message of the complex Mediator Service can begin. Rules `R34` to `R36` generate one `all_ms_answered` object per request to indicate this.

Rule `R34` counts the number of sequence pairs per complex Mediator Service using the `count` aggregate function. Rule `R35` counts the number of `half_sequence` objects, i.e. the number of auxiliary Mediator Services that have responded for a specific original request `R` of a complex Mediator Service `MS`, again using the `count` aggregate function. Finally, rule `R36` compares the two numbers and if they are equal generates the `all_ms_answered` object.

R34

```
if      SEQ@ms_sequence(complex_ms:MS)
then    total_ms(complex_ms:MS,questions:count(SEQ))
```

R35

```
if      HS@half_sequence(request:R,sequence:SEQ) and
        SEQ@ms_sequence(complex_ms:MS)
then    ms_answered(request:R,complex_ms:MS,answers:count(SEQ))
```

R36

```
if      X@ms_answered(request:R,complex_ms:MS,answers:N) and
        Y@total_ms(complex_ms=MS,questions=N)
then    all_ms_answered(request:R)
```

Support for Alternative Workflow Patterns

More synchronization constructs (or workflow patterns) are supported for auxiliary Mediator Services, when the list of auxiliary Services is not considered as a sequence but as a list of independent services. For example, the *parallel split* workflow pattern [2] is supported if rules `R30` and `R31` are replaced by rule `R37` that ignores the `half_sequence` and `full_sequence` objects. Notice that the termination rules `R34` to `R36` are still valid because they are based on the `half_sequence` and `ms_sequence` objects that are still generated and they are not based on the `selected_mediator_service` objects.

R37

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA) and
        MS@complexMediatorService(mediatorServiceAddress=MSA) and
        SEQ@ms_sequence(complex_ms=MS,first_ms:MS1)
then    selected_mediator_service(request:I,mservice:MS1,cservice:MS)
```

Hybrid approaches, where some of the services in the auxiliary list are in sequence while others are independent, can only be supported if the representation scheme (i.e. DTD) of the complex Mediator Service's metadata is extended to express various workflow patterns schemes. For example, instead of just lining up service names, extra element tags can exist around groups of service names to indicate which groups are sequences or splits or even choices (see below).

The *exclusive choice* and *multiple choice* patterns require domain-dependent modifications to the above scheme, since they both involve conditional routing. For example, the condition of rule `R38` identifies a complex Mediator Service `MS` and two of its auxiliary Mediator Services `MS1` and `MS2` through conditions related to their names and/or other of their metadata. Furthermore, the Mediator Service `MS1` has finished execution since a corresponding `half_sequence` object is present. On the other hand, Mediator Service `MS2` has not been executed yet, therefore the corresponding `half_sequence` object does not exist. Finally, the rule has a domain-dependent condition that involves the content of the response received from service `MS1`. If the condition is satisfied, then the service `MS2` is selected as the next Mediator Service.

R38

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA) and
        MS@complexMediatorService(mediatorServiceAddress=MSA,
                mediatorServiceName=Name of MS Service,
                Other condition on MS Service) and
        SEQ1@ms_sequence(complex_ms=MS,first_ms:MS1) and
        MS1@registeredMediatorService(mediatorService=Name of MS1 Service,
                Other condition on MS1 Service) and
        HS1@half_sequence(request=I,sequence=SEQ1) and
        SEQ2@ms_sequence(complex_ms=MS,first_ms:MS2\=MS1) and
        MS2@registeredMediatorService(mediatorService=Name of MS2 Service,
                Other condition on MS2 Service) and
        not HS2@half_sequence(request=I,sequence=SEQ2) and
        Domain-dependent condition involving the content of the response received from Service MS1
then    selected_mediator_service(request:I,mservice:MS2,cservice:MS)
```

Multiple such rules can exist for each different execution path that should be followed after the termination of service `MS1`. If conditions overlap then there is a *multiple choice* pattern, otherwise the pattern is *exclusive choice*. If the choice is to be made just after the original input message is received by the complex Mediator Service, the above rule needs some modifications, because there are no `half_sequence` objects (rule `R39`). Furthermore, the domain-dependent condition should now involve the content of the original message.

R39

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA,
                Domain-dependent condition involving the content of the original input message) and
        MS@complexMediatorService(mediatorServiceAddress=MSA,
                mediatorServiceName=Name of MS Service,
                Other condition on MS Service) and
        SEQ1@ms_sequence(complex_ms=MS,first_ms:MS1) and
        MS1@registeredMediatorService(mediatorService=Name of MS1 Service,
                Other condition on MS1 Service) and
        not HS@half_sequence(request=I)
then    selected_mediator_service(request:I,mservice:MS1,cservice:MS)
```

Since in the above scheme not all auxiliary mediator services are selected the total number of `half_sequence` objects will not reach the total number of `ms_sequence` objects, therefore termination rules `R34` to `R36` cannot be used. Furthermore, there is no other safe domain-independent criterion on which to base termination, which can only be decided by domain-dependent rules that will match the state that should be reached by the workflow in order to generate an ok flag that identifies that the construction of the output SOAP message of the complex Mediator Service can begin.

Arbitrary cycle patterns can also be supported using such domain-dependent service selection and termination rules, provided that the programmer is responsible for expressing appropriate conditions to avoid infinite loops. However, the above domain-dependent scheme needs a slight modification, because if an auxiliary service `MS2` has already been selected for execution and has responded, then rule `R38` cannot re-select it because of the condition `not HS2@half_sequence(request=I, sequence=SEQ2)` which insists that the service to be selected has not responded yet.

To avoid this, the structure of `half_sequence` objects should be extended with another attribute `iteration` which will keep a counter of how many times the same Mediator Service has been executed, reflecting the number of iterations performed over a set of auxiliary Mediator Services that form the loop. In this way, the re-selection of an already executed service `MS2` can be made if there is no existing `half_sequence` object whose `iteration` attribute value equals `N`: `not HS2@half_sequence(request=I, iteration=N, sequence=SEQ2)`. Value `N` is retrieved by the existing `half_sequence` object for the already executed Mediator Service `MS1`: `HS1@half_sequence(request=I,iteration:N,sequence=SEQ1)`.

In addition to the above two modifications in rule `R38`, rule `R32` that generates new `half_sequence` objects should be modified to cater for the `iteration` attribute. Rules `R40` and `R41` replace rule `R32`. Specifically, rule `R40` generates a new `half_sequence` object for a specific request and sequence with its iteration counter set to one, while rule `R41` generates a new `half_sequence` object with its iteration counter increased by one, when there are already existing `half_sequence` objects for a specific request and sequence. Notice that in order to retrieve the largest iteration counter `N`, the negated condition in rule `R41` is needed to ensure that there is no other `half_sequence` object for the same request and sequence with a largest iteration counter.

R40

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA1,request:R) and
        MS1@registeredMediatorService(mediatorServiceAddress=MSA1) and
        SMS@selected_mediator_service(request=R,mservice=MS1,
                cservice:MS) and
        SEQ@ms_sequence(complex_ms=MS,first_ms=MS1) and
        not HS@half_sequence(request=R,sequence=SEQ)
then    half_sequence(request:R,iteration:1,sequence:SEQ)
```

R41

```
if      I@input_soap_message(content:C) and
```

```
        C@mediatorService(url:MSA1,request:R) and
        MS1@registeredMediatorService(mediatorServiceAddress=MSA1) and
        SMS@selected_mediator_service(request=R,mservice=MS1,
                cservice:MS) and
        SEQ@ms_sequence(complex_ms=MS,first_ms=MS1) and
        HS@half_sequence(request=R,iteration:N,sequence=SEQ) and
        not HS1@half_sequence(request=R,iteration>N,sequence=SEQ) and
        prolog{N1 is N + 1}
then    half_sequence(request:R,iteration:N1,sequence:SEQ)
```

The *interleaved parallel routing* workflow pattern is supported if services in the sequence of auxiliary Mediator Services are selected in an arbitrary order and not following their position in the sequence. Rule `R42` below replaces rules `R30` and `R31` and achieves this unordered sequence workflow by selecting an arbitrary auxiliary service `MS1` and ensuring (via the negated last condition element) that no other selected service exists.

R42

```
if      I@input_soap_message(content:C) and
        C@mediatorService(url:MSA) and
        MS@complexMediatorService(mediatorServiceAddress=MSA) and
        SEQ@ms_sequence(complex_ms=MS,first_ms:MS1) and
        not HS@half_sequence(request=I,sequence=SEQ) and
        not S@selected_mediator_service(request=I,mservice\=MS1,cservice=MS)
then    selected_mediator_service(request:I,mservice:MS1,cservice:MS)
```

Finally, merging and synchronization patterns (see Table 1) is supported using rules similar to rules `R8` to `R10` (and their alternatives) in section 5.2.1.

6.2.2. Combining and Transferring Results between Mediator Services

After dealing with the synchronization details of the complex Mediator Service and its auxiliary Mediator Services we can fill-in the domain dependent rules that actually implement the application logic of the cheapest/speediest parcel dispatch request. What are actually missing are the rules that construct the messages to be sent to the auxiliary Mediator Services and the message to be returned to the requestor of the complex Mediator Service.

Concerning the first Mediator Service, rule `R43` copies all the attribute-value pairs of the original input message to new `pair` objects in order to start constructing the XML message to be sent. This rule is quite similar to the well-discussed rule `R2` (section 5.1.1). However, in this case the `'CheapDeliveryOffer'` Mediator Service has not subscribed to the `'CheapDispatchRequest'` Mediator Service, as Web Services do, therefore, the `'CheapDeliveryOffer'` Mediator Service has to be identified. This is needed to differentiate it from the second Mediator Service that also receives messages from the complex Mediator Service.

Finally, the rest of the XML tree of the SOAP message is constructed by domain independent rules that are quite similar to rules `R5` to `R7` that have been presented earlier (section 5.1.3). The modifications that take place actually include the replacement of all Web Service references with Mediator Service ones.

R43

```
if      SMS@selected_mediator_service(request:R,mservice:MS1,
                cservice:MS) and
        MS@complexMediatorService(
                mediatorServiceName='CheapDispatchRequest') and
        MS1@registeredMediatorService(
                mediatorService='CheapDeliveryOffer') and
        R@input_soap_message(content:C) and
        C@mediatorService(pair.inputVector:P) and
        P@pair(attName:AttName,attValue:AttVal)
then    pair(!request:R,!service:MS1,attName:AttName,attValue:AttVal)
```

When the response of the first Mediator Service arrives, the name of the company with the best offer is sent to the second Mediator Service along with the attribute-value pairs of the original input message. Rule `R44` copies the attribute-value pairs of the original input message to new `pair` objects, preparing the XML message to be sent to the second Mediator Service (`'CompanyDispatchRequest'`). Additionally, rule `R45` finds the `organization` attribute of the response of the first Mediator Service and copies it as a new `pair` object for the XML message to be sent to the second Mediator Service. The rest of the XML tree for the SOAP message is constructed by the modified version of rules `R5` to `R7` that has been discussed above.

R44

```
if      SMS@selected_mediator_service(request:R,mservice:MS2,
                cservice:MS) and
        MS@complexMediatorService(
                mediatorServiceName='CheapDispatchRequest') and
        MS2@registeredMediatorService(
                mediatorService='CompanyDispatchRequest') and
        R@input_soap_message(content:C) and
        C@mediatorService(pair.inputVector:P) and
        P@pair(attName:AttName,attValue:AttVal)
then    pair(!request:R,!service:MS2,attName:AttName,attValue:AttVal)
```

R45

```
if      SMS@selected_mediator_service(request:R,mservice:MS2,
                cservice:MS) and
        MS@complexMediatorService(
                mediatorServiceName='CheapDispatchRequest') and
        MS2@registeredMediatorService(
                mediatorService='CompanyDispatchRequest') and
        SEQ@ms_sequence(complex_ms=MS,first_ms:MS1,second_ms=MS2) and
        MS2@registeredMediatorService(mediatorServiceAddress:MSA2) and
        C1@mediatorService(url=MSA2,request=R,pair.outputVector:P) and
        P@pair(attName=organization,attValue:O)
then    pair(!request:R,!service:MS2,attName:organization,attValue:O)
```

After the second Mediator Service has also responded the responses of both auxiliary Mediator Services must be tied together and be returned to the original requestor. Rule `R46` copies all attribute-value pairs of the response of both Mediator Services to new `pair` objects for the output SOAP message of the complex Mediator Service. Finally, the rest of the output SOAP message is constructed by rules similar to rules `R15` to `R17` that have been presented earlier (section 5.2.4). The only difference is the replacement of the synchronization construct `all_ws_answered` with `all_ms_answered`.

R46

```
if      A@all_ms_answered(request:R) and
        SMS@selected_mediator_service(request=R,mservice:MS1,
                cservice:MS) and
        MS@complexMediatorService(
                mediatorServiceName='CheapDispatchRequest') and
        MS1@registeredMediatorService(mediatorServiceAddress:MSA1) and
        C1@mediatorService(url=MSA1,request=R,pair.outputVector:P) and
        P@pair(attName:AttName,attValue:AttVal)
then    pair(!request:R,attName:organization,attValue:AttVal)
```

7. Conclusions and Future Work

In this paper a knowledge-based Web Service composition system, called SWIM, was presented. SWIM is based on the Service Domain model, which 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 rule language, called X-DEVICE. 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. Finally, the system allows the definition of arbitrary workflow and synchronization models among Web Services, based on the data-driven nature of the underlying rule inference engine. Therefore, the system goes even beyond the composition of Service Domains allowing the development of arbitrary Web Service composition models, supporting most of the workflow patterns that have been identified to cover the entire spectrum of workflow functionality [2].

A great advantage of Complex Mediator Services supported by the SWIM system, is that it allows the user to define high-level abstract descriptions of e-business workflows without worrying with details concerning the exact instantiation of the actual Web Services that will carry out the workflow. The user only expresses properties that the selected Web Services should have using a logic-based language.

In the future, we aim to increase the usability and adjustability of the system with a user-profiling system that will keep the history of the user-defined selection, aggregation and synchronization strategies for each different user of SWIM. In this way, strategies that have been successfully used in the past by a user can be retrieved and re-used in the future. Furthermore, we will support predefined ready-to-use abstract workflow patterns in the form of built-in derived classes “a la Prolog”.

We expect to support the full range of workflow patterns in [2] by coping with cancellation of Web Service execution after the latter has started execution. This will be achieved by extending WSDL descriptions of all the involved Web/Mediator Services with fault messages that will handle cancellation events and by catering for the behavior of Web/Mediator Services when they receive such SOAP messages. In addition, an extension of the complex Mediator Service metadata scheme and its processing in X-DEVICE are needed in order to facilitate the representation and interoperability of the multiple workflow patterns, as it has been discussed in section 6.2.1.

We plan to extend the current system for supporting richer Web Service metadata expressed in an ontology language like DAML-S [3], utilizing an RDF-aware extension of our own X-DEVICE system [4]. 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 and will allow the automatic discovery and dynamic subscription of Web Services to Mediator Services of the SWIM system. Furthermore, deductive meta-rules could be used to express heuristic knowledge that will allow Mediator Services to be combined dynamically into complex Mediator Services.

Finally, concerning our ongoing research on trusted internet, quality of the knowledge in the Semantic Web may be ensured by a distributed framework comprising different services. Several of these services will be implemented as web services, forming service domains. In such cases, the abstraction and declarativeness offered by SWIM enable focusing only on the high level definitions of service selection strategies.

8. 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] Aalst W. van der, Hofstede A. ter, Kiepuszewski B., Barros A., "Workflow Patterns", *Distributed and Parallel Databases*, Vol. 14, No. 1, pp. 5-51, 2003.

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

[4] Arkin A. et al., "Web Service Choreography Interface (WSCI) 1.0", W3C Note, Aug. 2002. http://www.w3.org/TR/wsci/

[5] Arkin A., "Business Process Modelling Language (BPML)", BPMI Proposed Recommendation, Jan. 2003. http://www.bpmi.org

[6] Bassiliades N., Vlahavas I., "Capturing RDF Descriptive Semantics in an Object Oriented Knowledge Base System", *12th Int. WWW Conf. (WWW2003)*, Budapest, Hungary, 2003.

[7] 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.

[8] 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.

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

[10] Bassiliades N., Vlahavas I., "A Knowledge-based Framework for Building Web Service Domains", accepted for presentation at *9th Panhellenic Conference in Informatics (PCI'2003)*, 21-23 November 2003, Thessaloniki, Greece.

[11] Bellwood T. et al., "UDDI version 3.0", UDDI Spec Technical Committee Specification, July 2002. http://uddi.org/pubs/uddi-v3.00-published-20020719.htm

[12] 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.

[13] Boag S. et al., "XQuery 1.0: An XML query language", W3C Working Draft, May 2003. http://www.w3.org/TR/xquery/

[14] Box D. et al., "Simple Object Access Protocol (SOAP) version 1.1", W3C Note, May 2003. http://www.w3.org/TR/SOAP/

[15] Booth D. et al., "Web Services Architecture", W3C Working Draft, Aug. 2003. http://www.w3.org/TR/ws-arch/

[16] Chinnici R. et al., "Web Services Description Language (WSDL) version 1.2", W3C Working Draft, June 2003. http://www.w3.org/TR/wsdl12/

[17] Thatte S. (ed.),"Business Process Execution Language for Web Services (Version 1.1)", May 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel

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

[19] Gray P.M.D., Kulkarni K.G., Paton N.W., *Object-Oriented Databases, A Semantic Data Model Approach*. Prentice Hall, 1992.

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

[21] 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.

[22] Piccinelli G., "Service Provision and Composition in Virtual Business Communities", Tech. Report HPL-1999-84, Hewlett-Packard, Palo Alto, CA, 1999. http://www.hplhp.com/techreports/1999/HPL-1999-84.html

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

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

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

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

[27] J. Ullman, *Principles of Database and Knowledge-Base Systems*, Rockville, Maryland, Computer Science Press, 1989.

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