This is to certify that the
thesis entitled

WEB SERVICE-ORIENTED COLLABORATIVE WORKFLOW MANAGEMENT FOR DESIGN AND MANUFACTURING

presented by

HONG SUK JUNG

has been accepted towards fulfillment
of the requirements for the

M.S. degree in Department of Computer Science and Engineering

Major Professor's Signature

July 22, 2004

Date

WEB SERVICE-ORIENTED COLLABORATIVE WORKFLOW MANAGEMENT FOR DESIGN AND MANUFACTURING

By

Hong Suk Jung

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science and Engineering

2004

ABSTRACT

WEB SERVICE-ORIENTED COLLABORATIVE WORKFLOW MANAGEMENT FOR PRODUCT DESIGN AND MANUFACTURING

By

Hong Suk Jung

The web technology has been allowing the formation of "virtual" organizations to handle wide range of design and manufacturing process. Many collaborative systems can coordinate distributed processes among manufacturers, and the Web service technology can provide more scalable and flexible collaborative workflow management in such systems. The manufacturing processes of each collaborative system can be posted as a globally-understandable service by using Web service technology and used by other systems. The service-oriented process model provides the fundamental foundation of globally-acceptable representation and management for distributed processes as services. In this model, the OWL has been used to represent the process and describe execution status of process in this model. The Web service-oriented process model has been successfully deployed in the Manufacturing Integration and Design Automation System (MIDAS). The authoring environment of MIDAS enables users to create and advertise the OWL process definitions, and deploy Web services for those process definitions. The execution environment of MIDAS enables users to search, select and execute the posted process by using Web service.

Table of Contents

Table of Contents iii

List of Figures vi

Chapter 1 Introduction 1

Chapter 2 Background 4

2.1 Requirements for Collaborative Workflow Management System 4

2.2 Process Grammar 7

2.3 WfMC's Workflow Reference Model 9

Chapter 3 Web Services 14

3.1 Definition of Web service 14

3.2 Three Tier Model 14

3.3 Characteristics of Web service 16

3.4 Benefits from Web Service 17

3.5 Web Service Choreography 19

3.6 Markup Languages for Collaborating Web services 20

Chapter 4 Service-Oriented Process Model 22

4.1 OWL 22

4.2 Process Definition Model 23

4.2.1 Class Service 25

4.2.2 Class LogicalService and Class AtomicService 29

4.2.3 Class ServiceComposite *33*

4.3 Workflow Management Model 35

4.3.1 Common Operations *36*

4.3.2 Common Execution States *37*

4.3.3 Asynchronous Collaboration Scenarios *39*

4.4 Execution Monitor Model 39

Chapter 5 MIDAS Framework 42

5.1 Features of MIDAS 44

5.2 MIDAS Architecture 45

5.2.1 Process enactment engine *46*

5.2.2 Cockpit *48*

5.2.3 Process Library and Tool Library *49*

5.2.4 Web Service Modules *50*

5.3 Web service Server System 50

5.4 The Service Registry 51

Chapter 6 MIDAS Authoring Environment Using Web service 54

6.1 Creating Process Definition 55

6.1.1 Process Definition for Atomic Service *55*

6.1.2 Process Definition for Logical Service *57*

6.2 Deploying Web service 58

6.3 Registering Web service 59

Chapter 7 MIDAS Execution Environment Using Web service 61

7.1 Service Discovery 62
7.2 Negotiation for Collaboration 63
7.3 Process Enactment Using Web service 64
7.3.1 Task assignment using Web service 65
7.3.2 Use Web service as public process library 68
7.3.3 Tool invoking by Web service 68
7.4 Monitoring Enactment 70
APPENDIX A. JAVA Package for SOM 72
APPENDIX B. Converting Graph to Process Definition 80
APPENDIX C. Example of Process Definition 83
APPENDIX D. WSDL for Service Registry 87
APPENDIX E. Example of jws codes and its WSDLs 93
Summary 96
References 97

List of Figures

Figure 1 Four Companies in a Die Casting Process 6

Figure 2 Example of top-level process and production 8

Figure 3 WfMC's high-level functionalities for WfMS 11

Figure 4 WfMC reference model 13

Figure 5 Three Tier Model 15

Figure 6 OWL information modelling 23

Figure 7 UML notation for class Service 25

Figure 8 UML notation for class AtomicService 30

Figure 9 UML notation for class LogicalService 31

Figure 10 UML notation for ServiceComposite and ComponentService 34

Figure 11 UML notations for execution monitor model 38

Figure 12 UML notations for execution monitor model 41

Figure 13 Web service-oriented interoperation between business entities 43

Figure 14 MIDAS architectures 45

Figure 15 Entry of service registry 52

Figure 16 Creating definition for atomic service 56

Figure 17 creating definition for logical service 57

Figure 18 Deploying Web service on the Internet 58

Figure 19 Registration of a service and format of entry 60

Figure 20 Overview of Web service-oriented interoperation in MIDAS 61

Figure 21 Overview of service discovery in the MIDAS 63

Figure 22 Overview of task assignment using Web service 66

Figure 23 Detail view of task assignment using Web service 67

Figure 24 Overview of using Web service as process library 68

Figure 25 Overview of tool invoking using Web service 69

Figure 26 Detail view of tool invoking using Web service 70

Figure 27 Overview of monitoring using Web service 71

CHAPTER 1 INTRODUCTION

The advancement of information technologies, dynamic markets, and changes in production business have set a new stage for manufacturing practices in the fiercely competitive industry [Alsop 1998, Baldwin 1995]. To stay competitive, manufacturers must be able to 1) manage increasing product complexity and product innovation from market demands, 2) have faster and more flexible product development cycle, and 3) control globally distributed/outsourced operations. Collaborative systems for product design and manufacturing have been introduced so that manufacturing organizations obtain competitiveness by creating products in less time, at less cost, and with fewer defects. For example, the Collaborative Product Commerce solution unifies the product life cycle by enabling the sharing of product knowledge and incumbent manufacturing applications [Aberdeen 2002].

A variety of collaborative systems to manage manufacturing processes to the heterogeneous business environment [Bourke 2000] have been proposed. However, these frameworks mostly focus on system integration in a closely coupled design and manufacturing environment. Therefore, such systems may show weakness in terms of scalability and extensibility which a loosely-coupled component architecture would not.

Recently the term, *service-oriented interoperability* has been introduced to depict the behavior of collaborating systems in the loosely-coupled way. The service-oriented interoperability means that no matter what execution logic has been used inside each system, no matter what implementation has been used for each system, each system exists as a self-contained service so that the collaboration proceeds without any concern about heterogeneous working environment of each system. The Web service technology is the most recent and most appropriate web technology to implement such service-oriented interoperability to collaborating systems. The processes of each collaborative system can be posted as a globally-understandable service and used by other systems through Web service.

The service-oriented process model that I am proposing in this thesis is a model to provide the fundamental foundation that enables a globally-acceptable management for distributed processes as services. In this model, the OWL, which is a global ontological markup language, has been used to represent a process definition and describe execution status of on-going process.

To demonstrate the globally-acceptable management for distributed processes as services, the service-oriented process model has been implemented to the framework, *Manufacturing Integration and Design Automation System* (MIDAS). The MIDAS framework was originally developed to support collaborative design and manufacturing by integrating design engineering, process engineering and business plan [Chung 2003]. In the previous version, the MIDAS framework

supported the dynamic nature of manufacturing, such as a run-time process reconfiguration. However architecture wise, the old version still restricted its scalability and flexibility like other collaborative workflow management systems. To overcome such restrictions and get the service-oriented interoperability, the service-oriented process model has been implemented to the old MIDAS framework. With the service-oriented interoperability, the new MIDAS framework can post a manufacturing process as a service, actively locate a posted service, and integrates collaborative services into an optimized process workflow. Through such service-oriented operations, a user of MIDAS can get large scalability and more sophisticated aid on process design and management.

In following chapters I will discuss about background of my study, how the Web service-oriented interoperability can be realized by modeling a globally-acceptable process definition and representation of enactment, and how the MIDAS framework adopts such modeling to achieve the Web service-oriented interoperability. In chapter 2, the backgrounds of my research are discussed. In chapter 3, the Web service technology is discussed. The chapter 4 talks about a service-oriented process model. The chapter 5 summarizes the architecture of new MIDAS. The chapter 6 explains how the new MIDAS authoring environment creates a service definition, deploys Web service and register a service. The chapter 7 displays how the new MIDAS execution environment discovers Web service and executes the Web service.

CHAPTER 2 BACKGROUND

2.1 Requirements for Collaborative Workflow Management System

Much research has been dedicated to design and manufacturing from the perspective of process management. The reason being that process management plays a central role in coordination among collaborative companies [Chung 1998, Lavana 1997, Schey 1987]. I identify the following requirements to be provided by the collaborative business process management system [Chung 2003]:

1. The engineering process is understood to be tentative and iterative by nature

2. The process should be easily reconfigured when changes in user requirements occur or when the results may not conform to the constraints

3. A distributed data server is required to access data transparently, and to prevent unauthorized use

4. Companies should be able to execute their own process concurrently with others during collaboration.

Managing processes in collaborative systems is highly dynamic and poses problems completely different from conventional workflow management where flows are static. In [Chung 2003], they call such type of processes an *enacted processes*. These involve sub-processes which are designed "on the fly", by the participants, as part of the main process that is being executed. These characteristics pose challenging problems as the scale of the system increases. The issue not only concerns transaction volume, but also involves a number of participating organizations, the number of interdependent parts that are being created, the number of alternative manufacturing processes involved, and so on.

In theory, web technology allows the formation of "virtual" organizations to handle a wide range of design and manufacturing processes. Companies that take advantage of each other's distinct production strengths would benefit the most [Aberdeen 2000]. Therefore, the whole design and manufacturing process - not just the pieces – needs to be configured in response to changes in technical considerations such as features, methods, materials, costs, and other critical decision parameters. This means planning and executing a process that would extend across formal organizational boundaries.

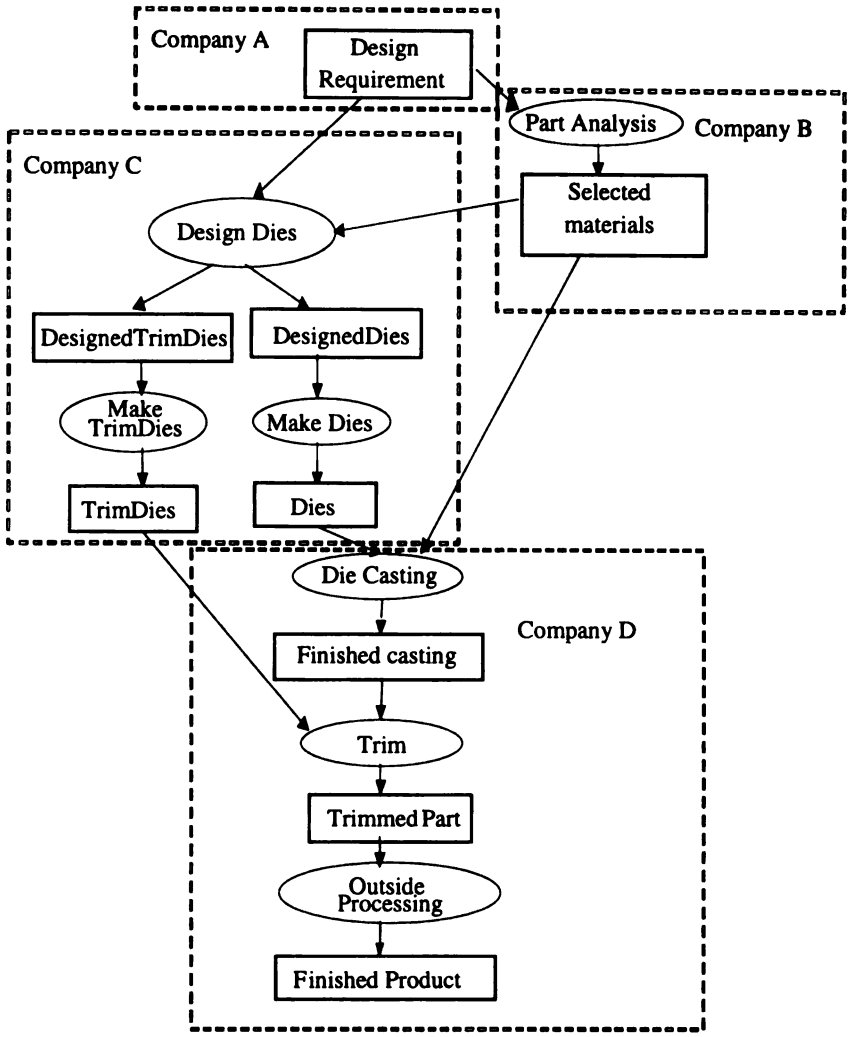


Figure 1 Four Companies in a Die Casting Process

Figure 1 illustrates a scenario where companies collaborate together to make a die. In this scenario, Company A designs the basic part. Company B performs some specialized analytical work to confirm that the design will meet the requirements. Company C is the die maker, which prepares dies for both casting and trimming. Finally, Company D actually produces and finishes the parts once the dies are ready. Each step along the way requires that a "checklist" be satisfied, before the work is started and after it is done. Before the

commencement of work, the vendor can provide useful feedback about the feasibility of the proposed work. After the work is finished, it is necessary to confirm that the requirements have been met. It should be noted that there may be many alternatives for each of these process steps: different technologies, different vendors, and so on. And at each step, there is a "make or buy" decision, as well. Also, for the overall process to function smoothly, all the participants need access to certain critical information (such as design changes). Ideally, they should be able to provide feedback early in the process concerning the feasibility and schedule for their part of the work.

2.2 Process Grammar

Process Grammar [Baldwin 1995, Chung 2002] has been proposed to represent design and manufacturing process and to generate process flow dynamically. MIDAS framework relies on the Process Grammar to enable "on the fly" sub-process configuration. In MIDAS, process flow graphs describe the information flow of a design methodology, and process grammars provide the means for transforming high-level task into progressively a more detailed set of tasks as well as selecting a method among many alternatives for a task.

The process flow graph consists of two types of entities: tasks and data specifications. A task is a single unit of design activity as defined with the process flow diagram. Data specifications are design data, where the output specification produced by a task can be consumed by another task as an input

specification. The flow diagram shows how to compose a task and the input and output specifications of the task. There are two types of tasks, a logical task and an atomic task. A logical task can be decomposed into a set of subtasks. An atomic task is the simplest form of the task, which cannot be decomposed any further. In the MIDAS, a logical task isn't bound to any executable application at build-time because a logical task will be decomposed at run-time. However, an atomic task must be bound by an executable application - typically a manufacturing tool – at build-time, and is responsible for executing the assigned tool at the run-time.

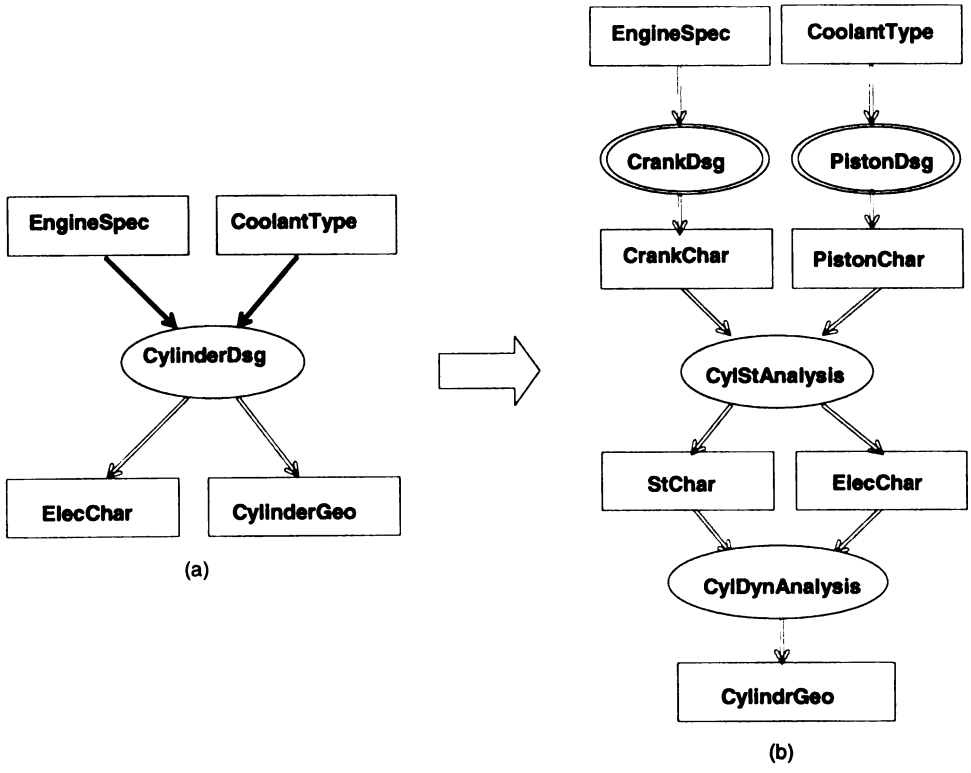


Figure 2 Example of top-level process and production

A *production* is a substitution that permits the replacement of a logical task with a flow graph that represents a possible way of performing the task. The concept of applying productions to logical tasks is somewhat analogous to the idea of productions in traditional (i.e., non-graph) grammars. If there are several production rules with the same left side flow graph, it implies that there are alternative production rules for the logical task. Figure 2 shows an example (A) top level process flow of "cylinder design" and (B) a production of logical task "CylinderDsg". These flow graphs allow designers to visualize process alternatives.

The process grammar provides an abstraction mechanism so that designers are not overly burdened with details. It allows a user to represent and manipulate a small number of abstract, higher-level tasks that can be expanded into detailed, executable alternatives. This can be especially valuable when engineers from different disciplines are working together on a project to build up an optimized distributed workflow to a group requirement. During the execution of a process, if the expansion of a certain abstract task does not meet the group requirement, a roll back can occur to an appropriate point and a new production can be applied to generate alternative process flow dynamically.

2.3 WfMC's Workflow Reference Model

As a process management system in product designing and manufacturing, MIDAS provides both manual and automatic workflow management function

based on rules given by Process Grammar. The Workflow Management Coalition (WfMC) defines *workflow* as an automation of procedure where information and tasks are passed between participants according to the defined set of rules, and the *workflow management system* (WfMS) aims to provide procedural automation of processes by management of the sequence of work activities and the invocation of human and/or IT applications associated with various activity steps [WFMC]. The MIDAS framework is designed to fulfill the WfMC's high-level requirement for WfMS functionalities.

WfMC suggests following high-level functionalities for WfMS [OMG 2000]:

1. The *Build-time functions*, concerned with defining the workflow process and its constituent activities

2. The Run-time control functions, concerned with managing the processes in an operational environment and sequencing a various activities to be handled as a part of each process

3. The Run-time interaction, concerning with monitoring steps of various activities between human and IT application tools

Figure 3 illustrates how the above functionalities work. Process design and definition are prepared at build-time by workflow management system. Then, at run-time, the workflow management system instantiates process and enacts the

process instance under interaction with human user or IT tool/application. Workflow enactment service is consisted of one or more workflow engines, and is responsible for run-time process instantiation and control. Sometimes workflow enactment service may change process definition at run-time. In MIDAS, this feature is supported sophisticatedly by "on the fly" sub-process configuration.

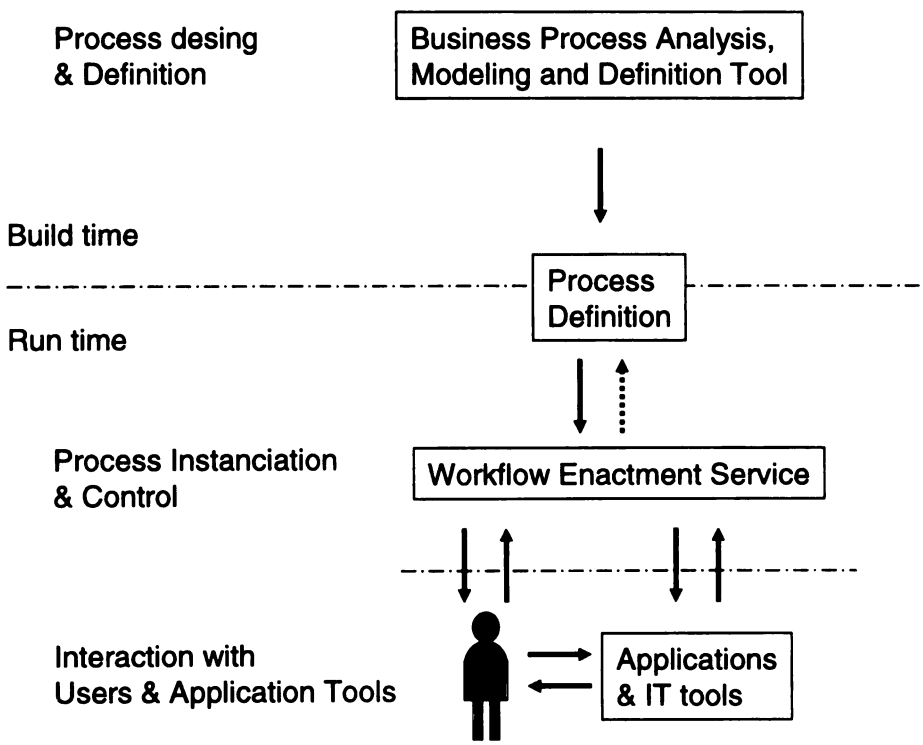


Figure 3 WfMC's high-level functionalities for WfMS

MIDAS framework also fulfills the facility design scheme in WfMC's *Workflow Reference Model*. This reference model identifies characteristics, functions and interface of workflow systems. Figure 4 illustrates WfMC's reference model. The reference model suggests five different functional facilities, which interact with the workflow enactment service:

1. *Workflow client application* contains the work list handler and process control software that need interaction with the human end-user.

2. *Process definition tool* creates process definition and transfers the process definition to the workflow enactment service or stores it in separate repository.

3. *Invoked application* is a specific tool undertaking a particular activity. Invoked application would typically be a server-based application with no user interface in many cases.

4. *Other workflow enactment service* is a heterogeneous workflow system produced by different vendors. The interoperability interface defines a way for different vendors to pass work items seamlessly between one another.

5. *Administration and monitoring tool* allows one vendor's management application to work with another's engine. The administration and monitoring interface enables several workflow services to share of common administration and system monitoring functions.

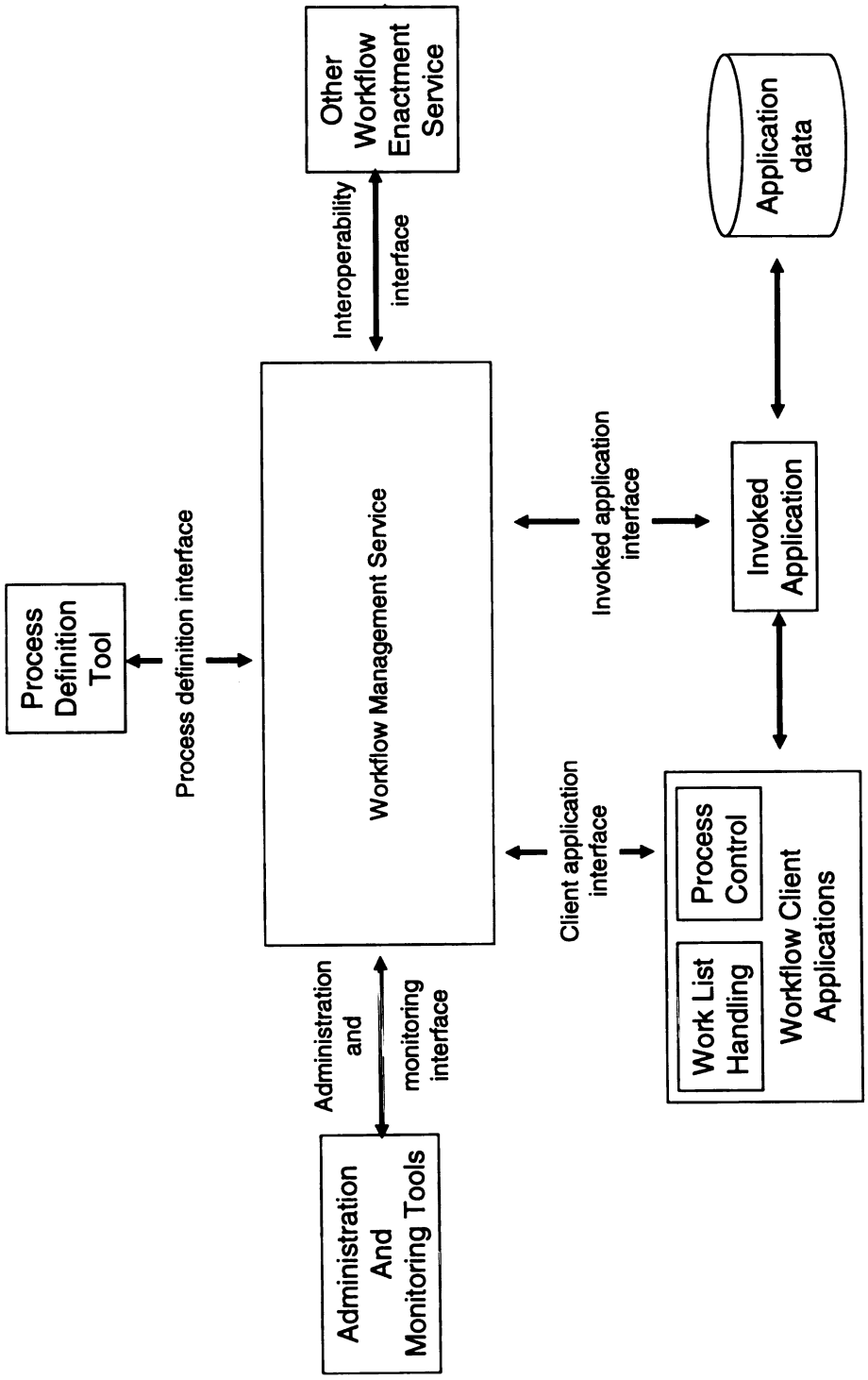


Figure 4 WfMC reference model

CHAPTER 3 WEB SERVICES

Web service is a key technology that enables MIDAS framework to have service-oriented interoperability. In this chapter, we will discuss Web service in detail.

3.1 Definition of Web service

Even though there are various definitions of Web service have been proposed and discussed by different point of views, Web service can be introduced as 'a content and software process that provide service to customer over Internet' in a broad sense [Sun 2003]. But today's information industry narrows down its definition of Web Service to 'a web-based software application whose definition can be found on the web as an open standard such as WSDL [W3C 2001], and its user interact by using XML-based messaging conveyed by Internet protocol, such as SOAP [W3C 2002]'.

3.2 Three Tier Model

The three tiers model is often mentioned to explain general structure of Web Service system and how each tier interacts. *Service Provider*, *Service Requester* and *Service Broker* are the three tiers.

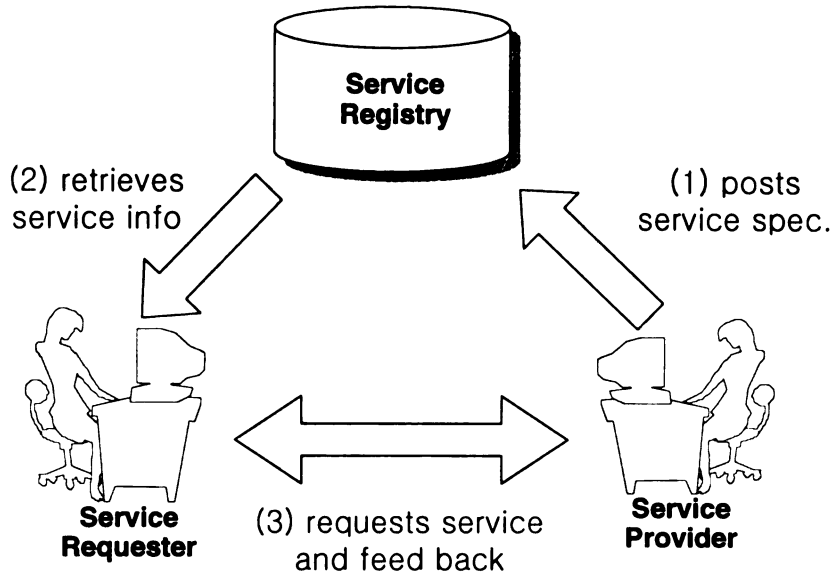


Figure 5 Three Tier Model

A service provider implements the web service and definition of its web service. WSDL is currently the most widely used standardized description language for Web Service. A service requester, then, uses service provider's web service by invoking the service in a predefined manner in WSDL. However, at the moment of invocation, the service requester usually does not have any idea where to get this WSDL of web service. A service broker is needed to introduce the service provider's web service and definition of its web service to the service requester. The service broker exists in a form of public registry, and UDDI is most popular standard protocol for a Web Service brokering registry. Figure 5 shows the three tier model and interactions between tiers.

3.3 Characteristics of Web service

The information industry is expecting that more than $15.2 billion will be spent on web service hardware and software by 2007 [McMillan 2003]. With no doubt, Web Service is standing in the spotlight of information technology industry. Why are IT developers so exited about Web Service? Here, I briefly summarizes distinct properties of Web service.

Interoperability:

The most important property of web service is to provide seamless and automatic connections from one software application to another over the web [Cohen 2002]. By using SOAP, WSDL, and UDDI [UDDI 2001] protocols defining a standardized way to discover and to call procedures in a web-based application, the interoperation between applications is possible without regarding location or implementation of platform.

Usability:

Service Broker (UDDI) provides a 'yellow-page type' business searching. Industrial categorizations based on standard taxonomies will increase usability of software components. The consumer of web-based applications can more easily and quickly discover applications best-matching to this purpose.

Applicability:

Development of web services can be an ongoing, iterative process that actively involves the opinions of the users [Bloomberg 2002]. The developer can construct the only exposed facade of web services as simple as possible, then later he can continually re-implement the web services to comply users' requirement so that web service are as broadly applicable as practical.

Modularity:

Non-fully encapsulated components in a complex system make the system very hard to replace or upgrade its components. Instead of simply exposing APIs, components wrapped in Web Services expose dynamic service descriptions. If the underlying API changes, then the service description adjusts by itself, and the other components of the system can adjust to the changes at runtime [Cohen 2002].

3.4 Benefits from Web Service

The collaborative workflow management framework will get the following benefits by using Web Service.

- ***To achieve implementation-neutral interoperability between various collaborative unit.***

  In the real manufacturing environment such as supply chain, decision making for product manufacturing does not rely upon a single unit. Usually it

demands multiple planning points such as multiple manufacturing departments or multiple manufacturing companies. Building a framework for managing such multiple decision-making entities, each supposed to operate mostly on their unique platform is not an easy and low-cost job. A Web service-oriented approach can reduce expenses and time spent on rewriting existing applications of each of the collaborating units. Web services interoperate each other only through interfaces capable to translate XML-based messaging into application's parameters. This technology provides ultimate implementation-neutral interoperability to the framework.

- ***To achieve more efficient collaboration by discovering counterparts fast and agreeing to requirements between collaborating units.***

  The web service registry and discovery technology such as UDDI, standardizes the way for web service to be exposed itself to the public. Through the UDDI or a similar web service searchable-registry, the enterprise can find its collaborating counterpart in the manufacturing industry, and can discover the best matching enterprise to its interest in precise and fast manners. Integration is also made in a fast manner since the discovery of WSDL or any extended description language (semantic web service) provides essential information for the web service to be agreed by user.

- ***To achieve more flexible and faster decision-making system to react to ever changing customer requirements.***

Even if the framework has successfully interconnected applications of each of the collaborating units, sometimes each applications' configuration may have to be changed in a hurry to catch up with the altered demands for product design by customers. For example, modern mass customization is requiring a generalized manufacturing line to produce various custom-made products. Designing a process that can flexibly change its configuration up to the customer's demand is very hard on a system whose components are tightly bound each other. In such systems, one small change on a part throws ripples throughout the whole system. On the other hand, web service-oriented approach provides lots of flexibility on changing process configuration. Since it totally encapsulates its implementation, each modularized application is able to interoperate in a loosely coupled manner so that one part's change doesn't impact elsewhere.

3.5 Web Service Choreography

The coordination of collaborating low level service is often mentioned as the next step in the development of Web service [W3C 2002]. The service choreography is the activity to define and represent the complex behavior of the set of collaborating services. The Web Service Choreography Interface (WSCI) 1.0 [WSCI 2002], which is still in progress and being discussed by W3C's working group, suggests the guideline of Web service choreography language. WSCI divides the behavior of Web service into two activities; *Atomic* and *Complex*. Atomic activities are the basic unit of behavior of a Web service, and Complex

activities are the behavior recursively composed of other activities. WSCI specifies four kinds of choreography for Complex activity; *sequential execution*, *parallel execution*, *looping* and *conditional execution*.

3.6 Markup Languages for Collaborating Web services

To support inter-operability of business process, a variety of standards and languages have been proposed. WSFL [WSFL 2002] is a workflow language that provides recursive composition of web services. WSFL takes a directed-graph model approach to process definition and execution. It also defines a public interface with which business processes can advertise. Based on WSFL, BPEL4WS was developed to model workflow management in terms of Web services choreography and flow modeling [Andrews]. BPEL4WS allows a composer to aggregate two or more web services into processes which may be abstract for a high-level business transaction or executable as a compiled process [Shapiro 2002, Weerawarana 2002]. BPML [Peltz 2003, Shapiro 2002] specifies web services orchestration and choreography. Orchestration in this context refers to an executable business process that can interact with both internal and external Web services, while choreography describes relationship and process flow among multi parties or multi organizations. Reliable and large-scale interoperation among trading partners is being attempted by creating a semantic web for each trading partner's service whose properties, capabilities, and interfaces are encoded in an unambiguous, computer-understandable form [Ding 2002, Fensel 2001, Hendler 2001]

These languages are all capable of providing Web service-oriented interoperability to the collaborative workflow management system. However, they do not provide sufficient process abstraction mechanism, with which users are not overly burdened with details. In addition, they do not separate the execution details of the process flow definitions. These process flows which the service providers publish should hide the details of execution parameters and scheduling of tasks. Such information should be determined at the time of process enactment. The most notable language that satisfies the above capabilities is the OWL-S specification, a language for ontology definition, manipulation, and reasoning [IBM 2003, Paolucci 2003]. OWL-S provides a mechanism to allow web service autonomy for identifying operational metrics at the design stage and hence facilitates heterogeneous web services discovery and integration. But, OWL-S does not model the iterative nature of collaborative product design and manufacturing process, where if the execution of a certain step of design process does not meet the design requirement, iterations with other alternatives must occur, and a new process flow should be generated dynamically.

CHAPTER 4 SERVICE-ORIENTED PROCESS MODEL

The Service-Oriented Process Model (SOM) is a key that enables the MIDAS framework to locate a remote process, and collaborate on distributed process flow. The ultimate goal of SOM is to provide a standardized way to understand distributed workflows and their executions among heterogeneous systems. To realize such goal, SOM specifies a global semantics of process definition and the way to represent the sequence of process flow enactment.

SOM consists of three sub models: *Process Definition Model*, *Workflow Management Model*, and *Execution Monitor Model*. Process definition model specifies how to represent a globally recognizable process definition. Workflow management model describes how to manage collaborative workflows that occur between separate WfMSs or within same WfMS. Execution monitor model specifies the representation of process enactment, which is understandable globally even by heterogeneous WfMSs.

4.1 OWL

Process definition model and execution monitor model are written with OWL (Web Ontology Language), which is a standard language to describe semantics for Web resources [Paolucci 2003]. OWL has been derived from the RDF language, which is a language for information modeling. OWL and RDF have a modeling structure similar to the directed graph. In this modeling structure, a

node is named as ***Classes***, and an arrow is named as ***Property***. There are two kinds of classes: Domain and Range. The property is a directed edge indeed, so the preceding node is called as domain, and the following node is called as range. Figure 6 illustrates the example of such graph. As you see in (b), property *productNumber* has class *Product* as its domain and class *Integer* as its range. This information modeling has been instantiated in (c).

(a) OWL's information modeling structure

(b) Example of OWL information modelling

```
<Product rdf:ID="WaterBottle">
<rdfs:label>Water Bottle</rdfs:label>
<productNumber>38267</productNumber>
</Product>
```

(c) Example of instance of (b)

Figure 6 OWL information modelling

4.2 Process Definition Model

Process definition model defines the semantics for a service provider's process flow in the context of a service flow. The service flow is represented as a *Service Composite*. Since the process definition model is designed to follow process modeling logic of process grammar, the process definition model defines a

service flow as a combination of *Atomic Services* and *Logical Services*. It is similar to the process grammar which combines *Atomic Tasks* and *Logical Tasks* to make a process.

A Service can include an *Input and output Specification* and a *Pre- and Post-Condition* as its reference marks. MIDAS framework utilized such marks when a service provider posts his service at the registry, and a service requester selects a service from the registry.

Service Composite is a placeholder for service provider's process flow. Service composite consists of a set of component services along with the task dependencies between component services. The dependencies between component services are captured by *linkTo* and *linkFrom* properties of component services.

Similar to a logical task, a logical service can have alternative choices of service composites to be expanded into more complex service flow. The *Alternative Choice* encapsulates such choice of service composites inside.

All of ontological concepts above are written in OWL classes, and shown in following sub sections. All OWL classes defined under the process definition model also have been implemented as a JAVA package by me. The highlighted feature of the JAVA package can be found at Appendix A.

4.2.1 Class Service

The service is a primary kind of entity in the process definition model. As mentioned previously, a service has various properties as its referencing markers. The referencing marks of a service are: the *name of service*, the *input specification*, the *output specification*, the *pre-condition*, the *post-condition* and *description of service*.

Since the process definition model has been written with OWL, all above referencing markers are represented as OWL classes, too. So the following OWL classes have been proposed: class Spec, class SpecList, class ConditionList, class Condition, and class Desc. Figure 7 illustrates the UML notation of class Service and its referencing makers.

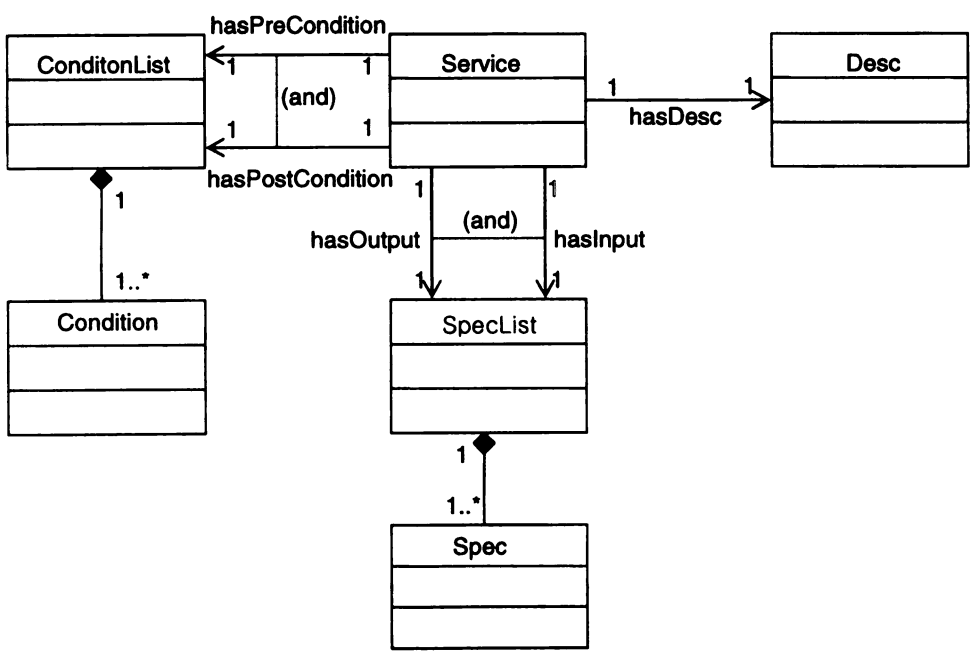


Figure 7 UML notation for class Service

Class Service uses two properties in order to mark up its conditions. The OWL property, *hasPreCondition* has class Service as its domain, and its range is class *ConditionList*. The OWL property, *hasPostCondition* has class Service as its domain, and its range is class *ConditionList*. Both of the two properties together indicate the pre-conditions and post-condition of a service. The OWL definition of hasPreCondition and hasPostCondition is shown as below.

```
<!-- Ontology for hasPrecondition -->
<rdf:Property rdf:ID = "hasPreCondition">
  <rdfs:domain rdf:resource = "#Service"/>
  <rdfs:range rdf:resource = "#ConditionList"/>
</rdf:Property>

<!-- Ontology for hasPostcondition -->
<rdf:Property rdf:ID = "hasPostCondition">
  <rdfs:domain rdf:resource = "#Service"/>
  <rdfs:range rdf:resource = "# ConditionList "/>
</rdf:Property>
```

Class ConditionList is a placeholder for multiple conditions. Class ConditionList inherits the OWL built-in class *collection*, so class ConditionList has multiple instances of class Condition as its items. Condition must be an ontology object that can be understandable by machine evaluator. This requires a machine evaluator to have a ontology dictionary for Conditions, but the service-oriented model doesn't propose such detail yet. The OWL definition of ConditonList and Condition is shown as below.

```
<!-- Ontology for ConditonList-->
<owl:Class rdf:ID = "ConditionList">
 <rdfs:subClassOf rdf:resource="owl+oil#collection"/>
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="owl+oil#Item"/>
   <owl:toClass rdf:resource="#Condition"/>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>

<!—Ontology for Condition -->
<owl:Class rdf:ID = “Condition”/>
```

Class Service also uses two properties in order to mark up its input and output. The OWL property, *hasInput* has class Service as its domain, and its range is class *SpecList*. The OWL property, *hasOutput* has class Service as its domain, and its range is class *SpecList*. Both of two properties together indicate the pre-conditions and post-condition of a service. The OWL definition of hasInput and hasOutput is shown as below.

```
<!-- Ontology for hasInput -->
<rdf:Property rdf:ID = "hasInput">
  <rdfs:domain rdf:resource="#Service"/>
  <rdfs:range rdf:resource="#SpecList"/>
</rdf:Property>
<!-- Ontology for hasOutput -->
<rdf:Property rdf:ID = "hasOutput">
  <rdfs:domain rdf:resource="#Service"/>
  <rdfs:range rdf:resource="#SpecList"/>
</rdf:Property>
```

Same as class ConditionList, SpecList is a placeholder for multiple conditions. Class ConditionList inherits the OWL built-in class *collection*, so class ConditionList has multiple instances of class Condition as its items. The OWL definition of Spec and SpecList is shown as below.

```
<!-- Ontology for specification -->
<owl:Class rdf:ID = "Spec"/>
<!-- Ontology for specification list-->
<owl:Class rdf:ID = "SpecList">
  <rdfs:subClassOf rdf:resource="owl+oil#collection"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="owl+oil#Item"/>
      <owl:toClass rdf:resource="#Spec"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Class Service may use class Desc as its optional referencing maker. This maker is not necessary for workflow management system to identify a given service. Class Desc holds the functional description of a service. Since its description will be written up in natural language, the machine agent in workflow management system won't process it. The primary reason to have this maker is to help a human user understand the functionality of given service. Similar with other classes, class Desc is linked to class Service by property *hasDesc*. The OWL definition of Desc and *hasDesc* is shown as below.

```
<!-- Ontology for description -->
<owl:Class rdf:ID = "desc"/>

<!-- Ontology for hasDesc -->
<rdf:Property rdf:ID = "hasDesc">
  <rdfs:domain rdf:resource = "#Service"/>
  <rdfs:range rdf:resource = "#description"/>
</rdf:Property>
```

4.2.2 Class LogicalService and Class AtomicService

Class Service has two children classes: class *LogicalService* and class *AtomicService*. As shown in Figure 8 and Figure 9 both of the two classes inherit all referencing makers of class service. So, both class LogicalService and class AtomicService have hasPreCondition, hasPostConditon, hasInput, hasOutput, and hasDesc as common.

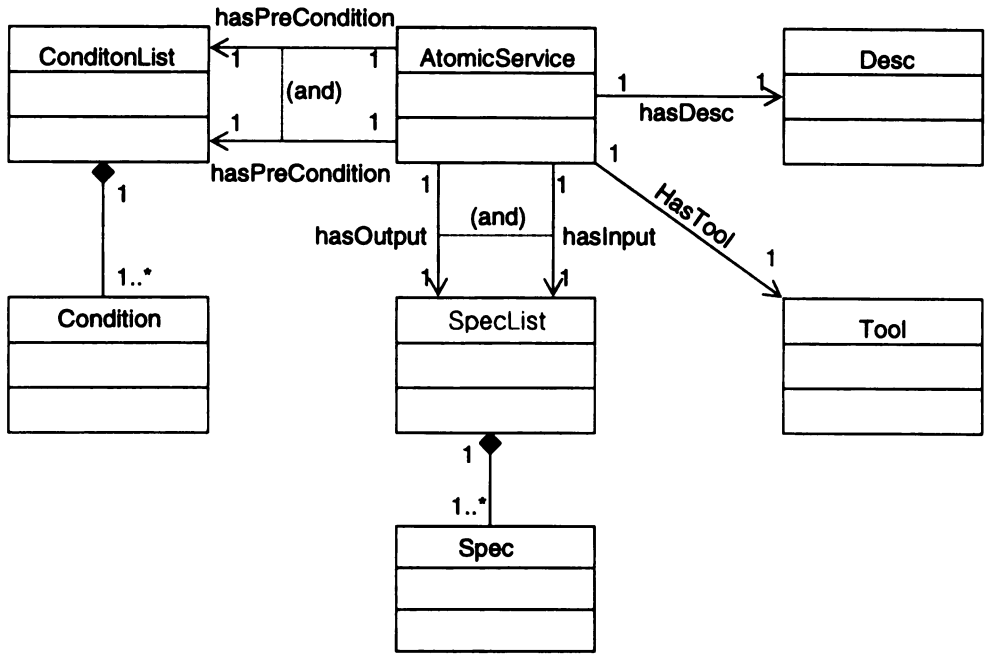


Figure 8 UML notation for class AtomicService

However, class AtomicService is distinguished from class LogicalService since it has additional property *hasTool*.

```
<!-- Ontology for hasTool
    It's a property between AtomicService and Tool -->
<rdf:Property rdf:ID=" hasTool ">
  <rdfs:domain rdf:resource="#AtomicService"/>
  <rdfs:range rdf:resource="#Tool"/>
</rdf:Property>
```

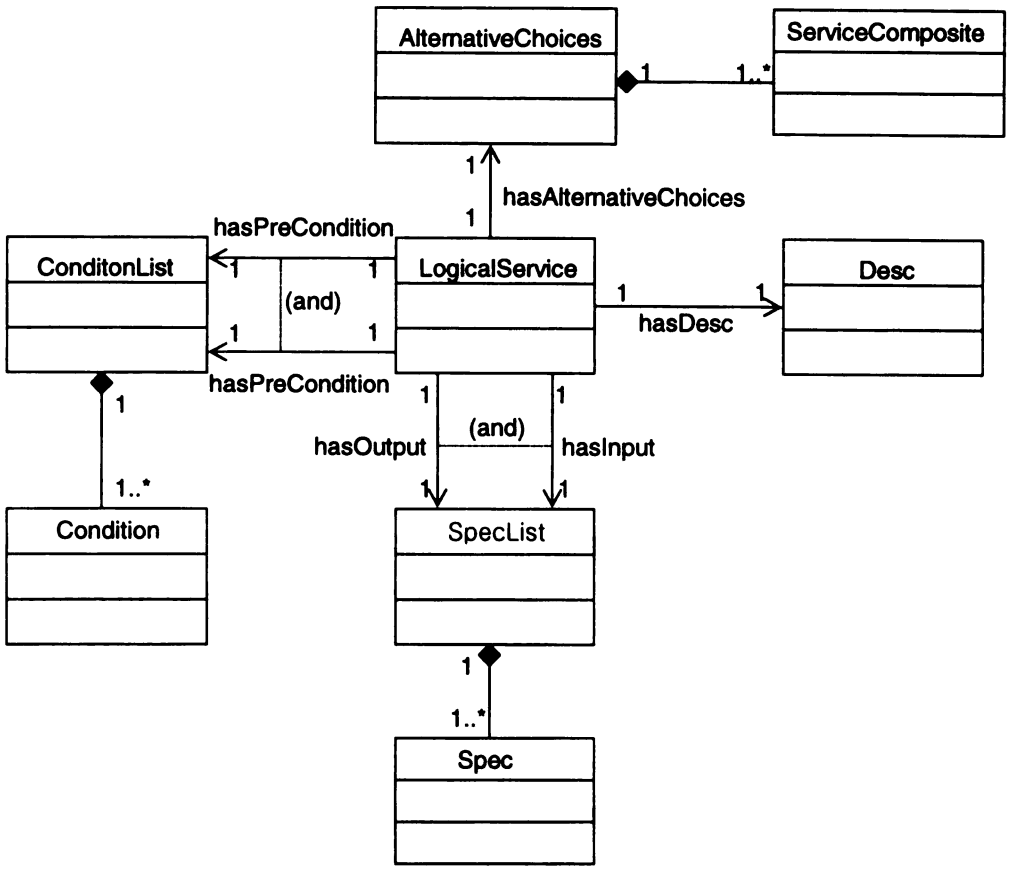


Figure 9 UML notation for class LogicalService

In the other hand, class LogicalService is also distinguished from class AtomicService since it has additional property *hasAlternativeChoices*. Property hasAlternativeChoices allows class LogicalService enlisting alternative choices for "on the fly" sub-process configuration. As shown as below, the hasAlternativeChoices has class *AlternativeChoices* as a range.

```
<!-- Ontology for hasAlternativeChoices
    It's a property between LogicalService and AlternativeChoices -->
<rdf:Property rdf:ID="hasAlternativeChoices">
 <rdfs:domain rdf:resource="#LogicalService"/>
 <rdfs:range rdf:resource="#AlternativeChoices"/>
</rdf:Property>

<!-- Ontology for AlternativeChoices
    It encapsules multiple production rules.
    It can have the list of Servicecomposites inside. -->
<owl:Class rdf:ID = "AlternativeChoices">
 <rdfs:subClassOf rdf:resource="owl+oil#collection"/>
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="owl+oil#Item"/>
   <owl:toClass rdf:resource="#ServiceComposite"/>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

Class AlternativeChoices is a placeholder for a collection of alternative choices. Similar with other collective placeholders such as SpecList or ConditionList, AlternativeChoices inherits its properties from the OWL built-in class *collection*. Class alternativeChoices collects instances of class ServiceComposite. The details of class ServiceComposite will be discussed at next section.

4.2.3 Class ServiceComposite

The process definition model specifies a complex service flow by combining instances of the class Service together. The process definition model marks up this complex service flow in class ServiceComposite. Class ServiceComposite is actually OWL's collection class, so class ServiceComposite can merely list the service consisting a complex service flow. Each of the Services composed of a complex service flow is marked up by class *ComponentService*. Class ComponentService is recognized as a same class as class Service by the process definition model. Its definition is derived from class Service by using OWL's equivalentTo property, so ComponentService can be directly substituted by Service instance in the process definition document. The OWL definition of ComponentService and *hasDesc* is shown as below.

```
<!-- Ontology for ComponentService. -->
<daml:Class rdf:ID="ComponentService">
  <daml:equivalentTo rdf:resource="#Service"/>
</daml:Class>
```

As same as other collective OWL classes, ServiceComposite has multiple instances of class ComponentService as its items. Figure10 illustrates the UML notation of class ServiceComposite and componentService.

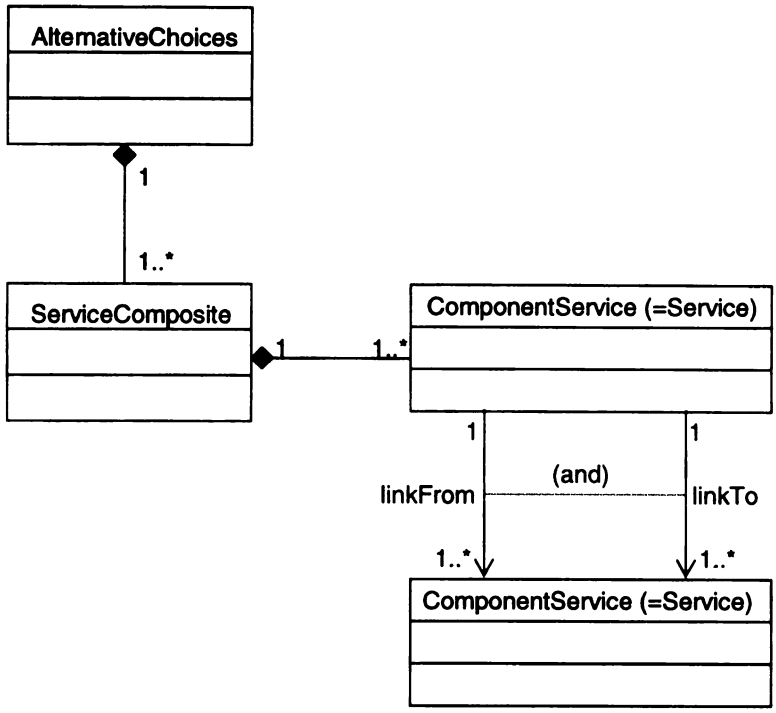


Figure 10 UML notation for ServiceComposite and ComponentService

According to the process modeling logic of Process Grammar, a process flow can be presented as a graph. Since the process definition model sees a task as a service, a complex service flow can be represented as a graph as well. In order to represent the flow information, class ComponetService uses two properties; *linkFrom* and *linkTo*. The OWL definitions of *linkFrom* and *linkTo* are shown as below.

```
<!-- Ontology for linkFrom -->
<rdf:Property rdf:ID="linkFrom">
  <rdfs:domain rdf:resource="#ComponentService"/>
  <rdfs:range rdf:resource="#ComponentService "/>
</rdf:Property>

<!-- Ontology for linkTo -->
<rdf:Property rdf:ID="linkTo">
  <rdfs:domain rdf:resource="#ComponentService"/>
  <rdfs:range rdf:resource="#ComponentService "/>
</rdf:Property>
```

The example of service flow-to-OWL conversion is shown in Appendix B.

4.3 Workflow Management Model

The workflow management model provides a standardized way of managing workflow between heterogeneous workflow management systems. Since different vendors of workflow management systems may use different workflow management schemes, each vendor must make their system comprise a global standard scheme of workflow management in order to make collaborations possible. To realize such global standard scheme of workflow management, the workflow management model specifies what kind of common operations is needed for workflow management, and what kind of common execution states should be defined.

The workflow management model also brings possible scenarios that could occur between heterogeneous workflow management systems during ***asynchronous collaboration***. The workflow management model does not consider any scenario for synchronous mode of software collaboration because, in synchronous mode of collaboration, one software component calling another component will freeze its system until it gets a response back. Since most of typical feedbacks take time in workflow management, the synchronous mode of collaboration is not suitable for collaborative workflow management. For this reason, the workflow management model proposes only the asynchronous scheme of workflow management.

4.3.1 Common Operations

The workflow management model stipulates seven standard operations, which are essential to gear up workflows of heterogeneous workflow management systems. These operations are described as follows:

- ***Provide input*** delivers input data to a service.

- ***Invoke enactment*** brings a cue to start process enactment. It carries out applying of production or tool execution.

- ***Rollback*** delivers rollback event to a service.

- ***Enforced rollback*** delivers aborting event to a service.

- ***Retrieve workflow graph*** delivers workflow graphs from a service to a viewer. This operation could be used for delivery of process definition or process enactment monitoring.

- ***Retrieve output*** transports output data from a service to a viewer.

All of above operations are to be implemented as a remote procedure of Web service and delivered to users of different workflow management system. The workflow management system does not need to implement all common operations above. It is up to the functionalities allowed to outsiders by the system.

4.3.2 Common Execution States

The workflow management model also defines the following five basic execution states:

- ***Un-initialized*** indicates that nothing has been initialized in a service.

- ***Ready*** shows that input data has been bound to a service, but service execution is not invoked yet.

- ***Running*** points out that the execution of a task has been invoked and keeps on going.

- ***Finished*** is the state that execution of a service has been finished. Two possible sub states are *success* and *fail*.

- ***Exception*** indicates that unexpected event has occurred during proceeding state.

Each execution state will advance forward or backward to another state when operations proceed. Figure11 illustrates the relationship between common execution states and operations. The *provide input* turns un-initialized state into ready state. The ready state then turns into running state by the *invoke execution*. When the workflow system finishes the execution of its process, the running state turns into the finished state. If the system wants to reconfigure its process, the *rollback* brings finished state back to ready state.

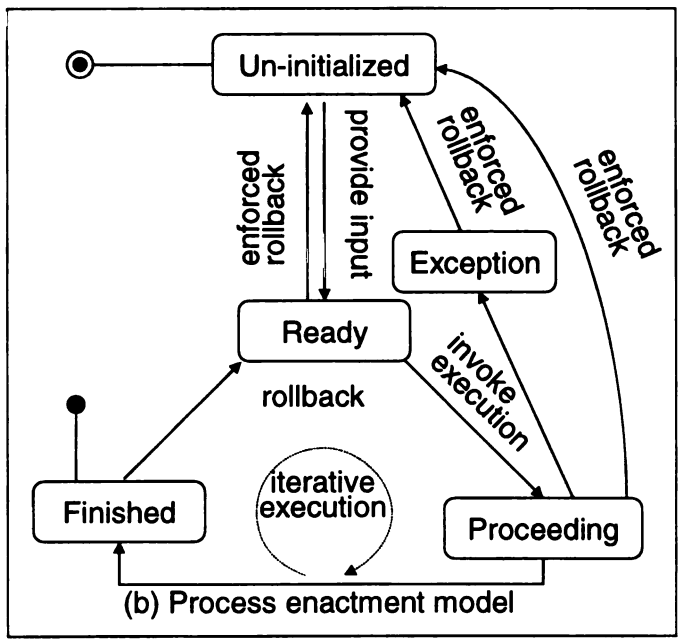


Figure 11 UML notations for execution monitor model

4.3.3 Asynchronous Collaboration Scenarios

The workflow management model considers two possible scenarios of asynchronous mode of software collaboration: *Invoke-then-polling* and *Invoke-then-listening*. These two scenarios have the same mechanism to invoke enactment. A service requester calls service provider's operations to invoke enactment. However, they use different mechanisms to feed back.

In the invoke-then-polling scenario, a service provider doesn't inform his finished state to the service requester. The service requester keeps checking the state changes at the provider's side by polling mechanism, but the service provider doesn't do anything to feed back. This scenario can be implemented by using only common operations. To get the most-updated feedback, the service requester calls the *retrieve workflow graph* operation periodically.

In the invoke-then-listening scenario, a service provider reports actively his state changes to the service requester. This scenario needs additional call back mechanism between the service provider and service requester. Currently SOM doesn't define how to implement such call back mechanism. SOM leaves it for the workflow management systems to negotiate each other on how to call back.

4.4 Execution Monitor Model

Execution monitor model provides globally acceptable standardized view of how to capture and deliver the process execution to viewers. Users of heterogeneous

workflow management systems can monitor a process by capturing traces of task executions and data binding across the system in terms of the Execution Monitor Model.

Execution monitor model follows the process enactment logic of Process Grammar. According to the mathematical model of Process Grammar, the execution of a logical task means an applying of a production to that logical task. Attaching a service composite instance to a logical service will capture this action.

The execution state of a component service is captured by the *Execution State*. The execution state takes one of the following instances, which is defined by the workflow management model: *Un-initialized*, *ready*, *running*, *finished* and *exception*.

Input specifications and output specifications are bound by *Data* during execution of tasks. Data is an entity that holds the URI of actual data. The input specification is bound by data instance when a user binds initial data to it or upstream task produces output data. The output specification is bound by the data instance when the task finishes its execution and produces results.

Figure12 shows UML notation of OWL classes in execution monitor model.

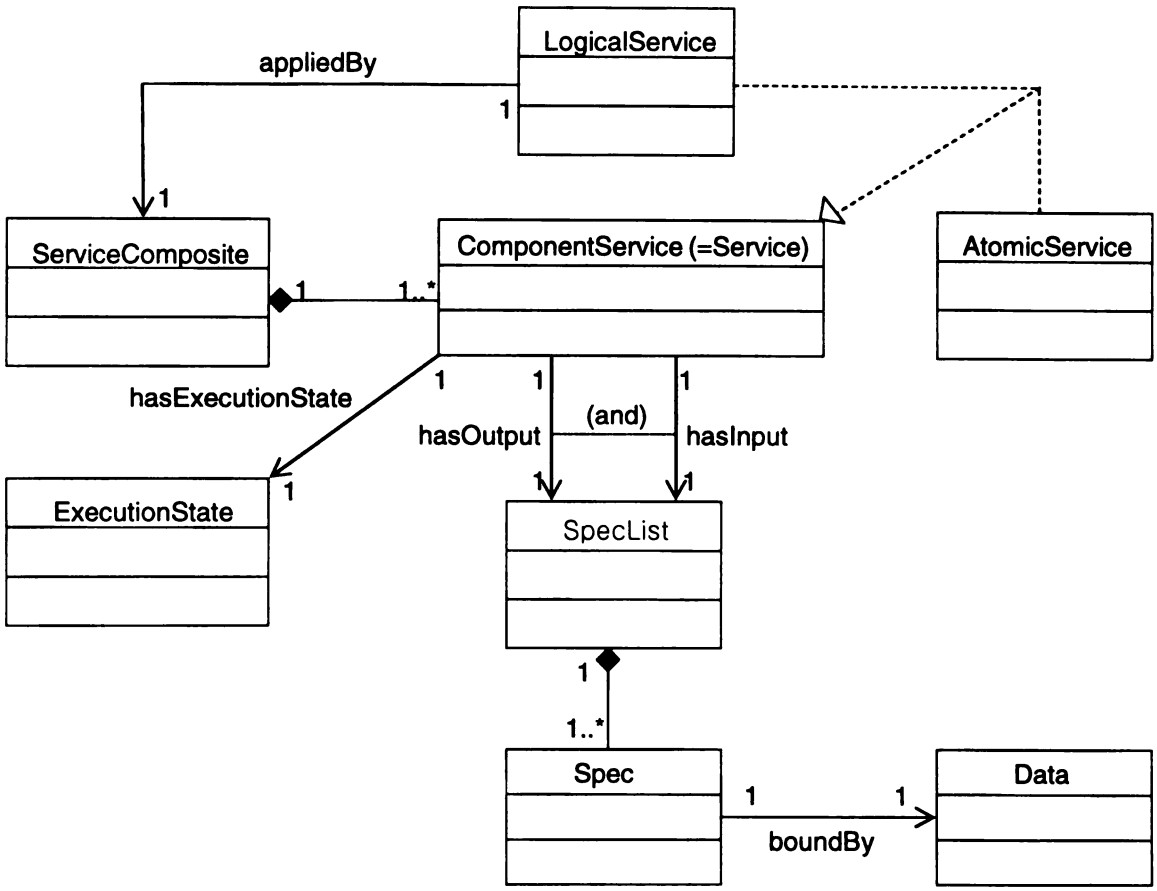


Figure 12 UML notations for execution monitor model

CHAPTER 5 MIDAS FRAMEWORK

The MIDAS framework is a collaborative engineering framework that coordinates various tasks in design and manufacturing with Web service-oriented interoperability. In previous version, MIDAS had been developed as a collaborative engineering framework without Web service-oriented interoperability [Chung 2002, Curbera 2003]. However, with the Web service-oriented interoperability, the new version of MIDAS gains the ability to provide the means to (1) locate manufacturers dynamically, (2) select and make contracts with particular manufacturer in agreement with requirements, (3) create the collaborative process by incorporating distributed services among manufacturers, and (4) provide a flexible and interoperable execution environment for the collaborative process.

The MIDAS framework provides a truly distributed architecture for management of manufacturing process composition and inter-operation in two aspects; (1) The process itself and its enactment are described in terms of global semantic language (OWL). (2) Process integration and execution are achieved by using the Process Grammar, which is a process modeling and enacting logic which helps MIDAS to configure manufacturing processes dynamically.

Collaborative process generation in MIDAS proceeds by interactions between manufacturing companies. For each alternative for sub-process generation, there

are manufacturing service providers who can participate internal collaboration or it can be outsourced for external collaboration. Service providers offer a manufacturing process as a service with global standard interface, and MIDAS guides a designer to select an appropriate service provider and his service flow. Figure13 illustrates how business entities use MIDAS framework to participate business collaboration. The contractor performs referencing for adequate service providers. After selecting one service provider, the contractor negotiates with the service provider, and reaches an agreement on cooperation. Incorporating subcontractor's process or monitoring the sequence of execution for assigned job to the subcontractor will be simply done by calling operations of subcontractor's Web services.

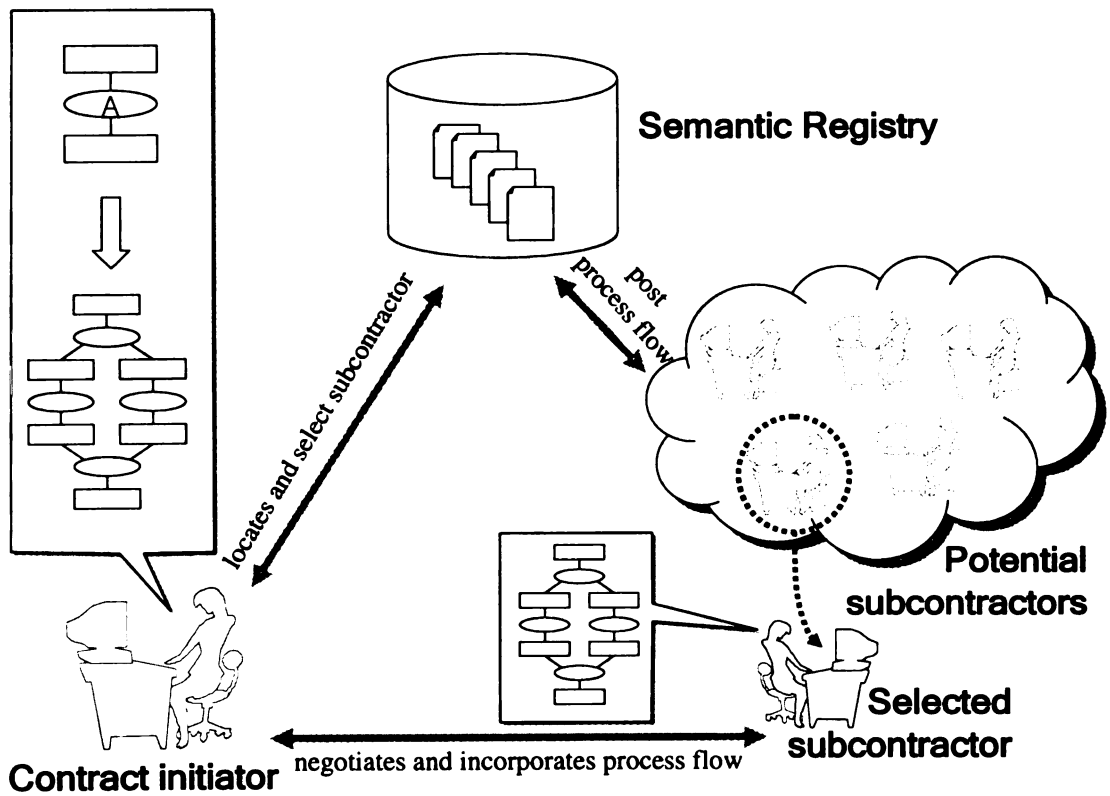


Figure 13 Web service-oriented interoperation between business entities

5.1 Features of MIDAS

Even though MIDAS does support all requirements required by collaborative workflow management system, MIDAS is distinguished from other workflow management system by the following distinct features:

- Separation of process specification from the execution environment. Syntactic structures, such as dependency among tasks and input output requirement, together with alternatives are specified using the process grammar. Execution details and constraints are encoded as a part of execution environment. By separating specifications from execution, MIDAS handles process run-time process reconfiguration.

- At the execution environment, a task execution can be accomplished by a Web service. A Web service can be located within same organization or at an external organization where its workflow management system could be totally heterogeneous. A Web service can provide either of a process enactment service, simple process library function or tool invoking function.

- MIDAS guides the user to select appropriate sub-process. Sub-process can be obtained from the user's personal library or outsourced from a public library, such as a process library Web service. All processes are provided along with attributes including pre-conditions and post-conditions. Using the pre- and post- conditions a user may reconfigure a process if an already

configured one does not provide the desired output. Through such reconfiguration steps, the framework generates an optimal process configuration within a given set of constraints.

5.2 MIDAS Architecture

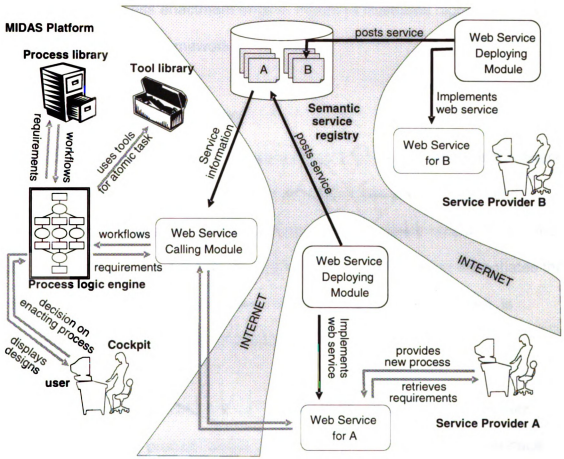


Figure 14 MIDAS architectures

The MIDAS framework consists of four major components: *Process enactment engine*, *Cockpit*, *Process Library/Tool Library* and *Web service Modules*. As mentioned at chapter 2, the components of the MIDAS framework are designed to fulfill the WfMC's workflow reference model. As a result, the MIDAS process

enactment engine implements the workflow enactment service of WfMC's reference model. The MIDAS Process library/tool implements the invoked applications of the reference model. WfMC's process definition tools function area is implemented as part of cockpit in MIDAS. WfMC's Administration and monitoring tools function area is implemented as a part of MIDAS cockpit and a part of MIDAS process enactment engine. Figure14 illustrates high-level architecture of MIDAS framework.

5.2.1 Process enactment engine

A process enactment engine's purpose is primarily to create, manage and enact a process instance. The process enactment engine can load up a process definition from two different kinds of sources: a user's private repository and the public accessible process library. The process enactment engine instantiates the process after it loads up the definition. The instance of a process, then, is managed and executed by the process enactment engine.

In previous version of MIDAS framework [Qin 2002], the process enactment engine accessed the process definition directly without any common interface. Such access without a common interface restricted the interoperability of the process enactment engine when the enactment engine needs to import a process definition from a different vendor's system. In the Web service-oriented version of the MIDAS framework, this restriction has been overcome by accessing a process definition source through a Web service-oriented common interface. In the new version, the process enactment engine has gained an ability

to import OWL process representation via Web service and to parse its OWL representation.

Atomic task

Invoking manufacturing tools by the process enactment engine also can be done either directly without any common interface or through a Web service-oriented common interface. In the case of direct invoking, the process enactment engine relies on a separate *Tool server* to execute and retrieve results. In the case of indirect invoking through common interface, the process enactment engine needs help from the Web service module instead of the tool server.

The logic of process enactment came from the Process Grammar's "on the fly" process configuration. The process enactment engine generates a new process flow by adding one of possible alternative sub-processes onto a logical task. If needed, the process enactment engine rollbacks execution and reconfigures the sub-process. In manual mode of process execution, the process enactment engine doesn't involve making a decision on the process enactment. All of the decisions come from a human user through the Cockpit. However, in automatic mode of the process execution, the machine agent in the enactment engine decides what to do for sub-process configuration and reconfiguration.

The process enactment engine is implemented as a JAVA RMI server. One organization should have at least one process enactment engine. If the organization wants to release a burden of a monolithic enactment engine,

multiple engines can run on the multiple JAVA RMI servers. In this case, multiple RMI servers should be linked and synchronized by a MIDAS distributed server infrastructure. The MIDAS distributed server infrastructure is consists of a *Yellow page server* and other RMI servers running enactment engines.

The process enactment engine is also responsible for checking the permissions needed when users access and execute resources belonged to the process instance. Full-scale of access control functionalities come from a separate access control system. The process enactment engine relies on the access control system to check the user's permission. Discussion about access control system is beyond this thesis's scope, but extensive discussion about MIDAS access control scheme can be found in [Zhang 2003].

5.2.2 Cockpit

A cockpit is a communication interface connecting a user to the process enactment engine. It couples a user and an engine by transmitting the user's decision on the process creation and enactment to the process enactment engine. The cockpit also provides graphical information about the process definition and enactment to the user.

The cockpit interacts with the human user in following situations: *Creating process definition*, *Displaying and maintaining process information archive*, and *Displaying enactment sequence of a process*. When a user defines a process,

the cockpit provides a graphical authoring environment. A user can actually draw a graph of a process flow, and the cockpit has an ability to document it in OWL.

The cockpit has been implemented as a downloadable JAVA applet. Since the cockpit should connect to the JAVA RMI server, which runs MIDAS process enactment engine, the cockpit is a JAVA RMI client as well.

5.2.3 Process Library and Tool Library

Process library

The *process library* is kind of repository that holds and distributes process flow definitions. The MIDAS framework uses two kinds of process libraries: the *JAVA RMI server-based library* and the *Web service-based library.* The JAVA RMI server-based library is implemented without a common interface. The access to the library must be done only via JAVA RMI client. On the other hand, the Web service-based library is to be used for collaborative workflow management among heterogeneous systems. As you expect, the Web service-based library provides a globally acceptable common interface. Furthermore, the process definition will be distributed in an OWL document in order to achieve interoperability.

Tool Library

The tool library provides manufacturing tools. The MIDAS framework considers two kinds of tools: the JAVA RMI server-based tool and the Web service-based tool. As mentioned previously, the MIDAS framework includes the tool server.

The JAVA RMI server-based tool is executed from the tool server when a user invokes the execution of an atomic service. The Web service-based tool is a public accessible application, which has a globally accessible common interface. Similar with the case of process library, the Web service-based tool can be utilized by different vendor's workflow management system.

5.2.4 Web Service Modules

Web service modules provide facilities such as browsing service semantics, and calling a Web service to enact process. MIDAS web service modules are composed of *Web Service Discovering Module*, *Service Registering Module*, *Web Service Deploying Module*, and *Web Service Calling Module*. The functionalities of these modules are implemented by using JAVA AXIS API (version 1.1) [AXIS 2003], xerces XML processing API [Xerces2] and SOM API (Appendix A). The detail functionalities of each module will be discussed at chapter 6 and 7.

5.3 Web service Server System

The MIDAS framework uses the *Tomcat-AXIS server system* [AXIS 2003] to realize the Web service working environment. The Tomcat-AXIS system is constructed by gearing up the Tomcat Web application server with *AXIS JAVA API*. AXIS API provides SOAP messaging functionality and Web service deploying/ invoking/ running environment, and Tomcat server posts such functionalities at the Internet.

Tomcat-AXIS system provides the unique method of Web service deployment that other systems cannot provide. Unlike other Web service systems that support only pre-compiling deployment scheme, the Tomcat-AXIS system supports both the pre and post-compiling deployment scheme for Web service. Under pre-compiling scheme, the implementation of the Web service must be pre-compiled by a user before it is loaded into the server. Under post-compiling scheme, the server will compile the implementation of Web service right before it is invoked. The user does not need to load up pre-compiled code.

The MIDAS framework deploys its Web service under the post-compiling scheme. The implementation of the Web service will be prepared in JAVA code and named with a jws extension, then, the deployment will be completed by importing the *jws* file into the specialized folder under the Tomcat-AXIS system.

5.4 The Service Registry

The service registry takes a very important place over MIDAS framework because it is one of facilities realizing Web service-oriented interoperability of a workflow management system. However, the service registry itself is a separate independent system, which is not a part of the workflow management system. The service registry exists in between collaborating workflow management systems to facilitate interoperation between collaborating workflow management systems by matchmaking them quickly.

It is important for the service registry to tell the service seekers which Web service can provide which service. For this, the service providers enroll their services with a description at the service registry, and the service registry lets out the simplified description of a service to the service seeker.

The service registry supports three basic functionalities: *Service enrollment*, *service disenrollment* and *browsing enrolled service*. These functionalities are implemented as remote procedures having a Web service common interface. The WSDL document for such remote procedure can be found at the Appendix D.

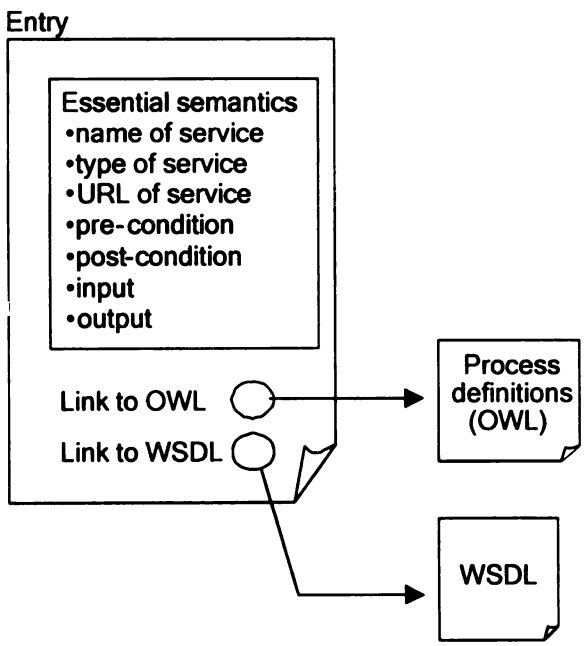


Figure 15 Entry of service registry

To reduce the registry size and improve efficiency of searching, entries includes only the minimum size of descriptions. The minimum size of a description includes the following information: *name of service*, *type of service*, *URL of*

service, *input and output data specifications*, and *the pre- and post-conditions of service* (figure 15).

The service seeker may need to reference the full description of service for the future use. The full description of service is given by separate document, and the link to this document is provided to the service seeker as well. The WSDL document is also provided for the service seeker's Web service calling module.

Entry of service registry is written in generic XML. Following box shows an example of an entry enrolling the LogicalService created for the logical task shown in figure 2(a). The entry form includes all minimum descriptions for a service and links to the OWL document and the WSDL document.

```
<entry>
  <entryID> 00001 </entryID>
  <serviceType> logicalService</serviceType>
  <serviceName> CylinderDsg</serviceName>
  <precondition> NotKnowCylinderDsg</precondition>
  <postcondition> NotKnowCylinderDsg</postconditon>
  <inputSpec> EngineSpec</inputSpec>
  <inputSpec> CoolantType</inputSpec>
  <outputSpec> ElecChar</outputSpec>
  <outputSpec> CylindarGeo</outputSpec>
  <wsdl_binding>http://midas/wsdl/CylindarDsg.wsdl</wsdl_binding>
  <owl_binding>http://midas/wsdl/services.daml</owl_binding>
</entry>
```

CHAPTER 6 MIDAS AUTHORING ENVIRONMENT USING WEB SERVICE

The primary goal of the *MIDAS authoring environment* is to provide a process flow authoring facility to a user. A user can create a *process definition* by authoring logical tasks and atomic tasks, then, combining those tasks into a process flow along with the data specification. In the MIDAS authoring environment, the process definition will be written in OWL as the process definition model describes. Once a process definition has been created, the process definition is stored into the private repository or public library.

The *cockpit* is the major component that is responsible for the authoring process. The MIDAS cockpit provides a user a sophisticated authoring tool with a graphical interface, which enables a user to create a process flow graph and converts the process flow graph into a process definition document.

The MIDAS authoring environment has an ability to deploy and publish a Web service to support the Web service-oriented interoperability. The MIDAS *Web service registering module* and *Web Service deploying module* together realize a Web service-oriented interoperability. The Web Service deploying module generates an implementation of a Web service based on the process definition. If the authoring environment needs to advertise its Web services, the Web service registering module registers Web services to the public Service registry.

Web service registration is only necessary when inter-organizational collaboration proceeds, because the MIDAS framework assumes that every participant within one organization know one another very well so that they don't need to advertise any Web service or task. However, during the collaboration between different organizations, they can hardly get to each other and both organizations need a rendezvous point to meet one another. So in this case, the MIDAS authoring environment registers Web services as a process flow or a tool at the *Service Registry*.

6.1 Creating Process Definition

The MIDAS authoring environment brings up two creation patterns of process definition in association with Web service; *Creating a process definition for an atomic service* and *Creating a process definition for logical service and its service composites*. These separate patterns regard two different kinds of Web services; A Web service invoking a manufacturing tool and other kinds of Web service, which aims either of process assignment or process flow library.

6.1.1 Process Definition for Atomic Service

As mentioned at chapter 2, an atomic task (atomic service in terms of the service-oriented model) is responsible for invoking a tool application, so the first step should be creating an atomic task in the cockpit. Through the cockpit, a user creates an atomic task by adding input data and output data specifications. After that, the atomic task must be bound by an actual tool application. Once the

atomic task has been prepared, the cockpit writes out the OWL document into the temporary storage. The example of OWL file can be found in Appendix C, and figure 16 illustrates all above process.

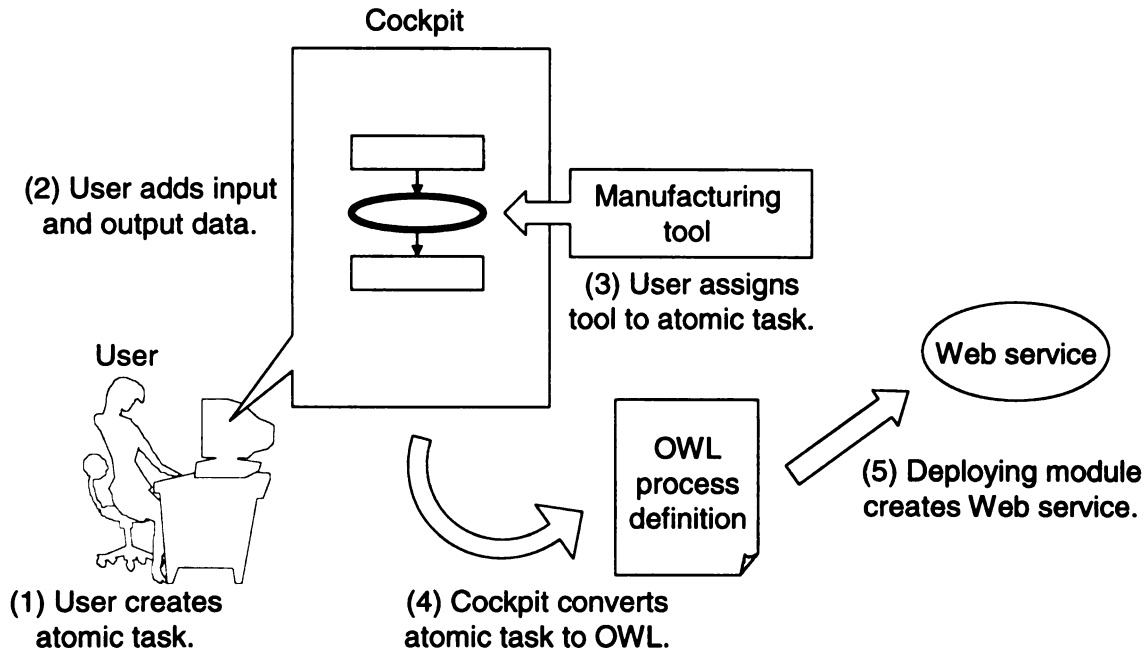


Figure 16 Creating definition for atomic service

Tool applications are typically a server-based software component. However, the MIDAS framework can invoke non server-based tool applications as well. As discussed in chapter 4, the process definition itself does not specify how an atomic service invokes a manufacturing tool. The process definition simply identifies the tool bound to an atomic service. The invoking mechanism of an actual tool totally depends on the Web service enactment design of each workflow management system.

6.1.2 Process Definition for Logical Service

In addition to an atomic service, a logical service performs an important role in Web service-oriented collaborative workflow management. A logical service is to be assigned for process enactment, or is to serve a public process library function. Either case, a logical service is designed to serve one or more service composites to the requester. So, the process definition should be placed where the Web service can reach and get the definitions.

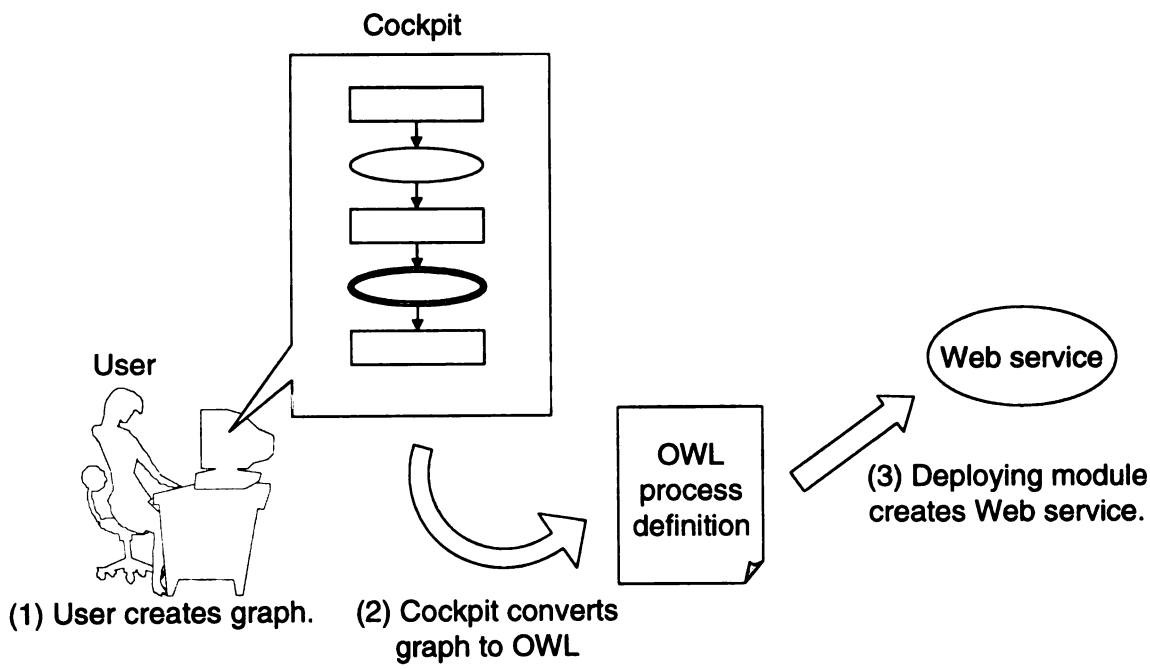


Figure 17 creating definition for logical service

The process definition for atomic service is rather simple since it specifies only a single tool application, but, for a logical service, it is not that simple, as it must specify service composites and its components. The cockpit helps users complete this complicated job fast. As shown in figure 17, a user combines tasks

to make a process flow graph at the first step. Before being combined into a process graph, the user must prepare all of the tasks. Once the process flow graph is ready, the cockpit writes out a process definition in OWL and stores it at the repository. The example of the OWL file can be found in Appendix C.

6.2 Deploying Web service

In MIDAS framework, the deployment of a Web service is accomplished by co-work of the Web service deploying module and Tomcat-AXIS server system. The Web service deploy module prepares the Web service implementation, and the Tomcat-AXIS server actually compiles the implementation and deploys it as public-accessible Web service on the Internet. Figure 18 illustrates such co-work for Web service creation and deployment.

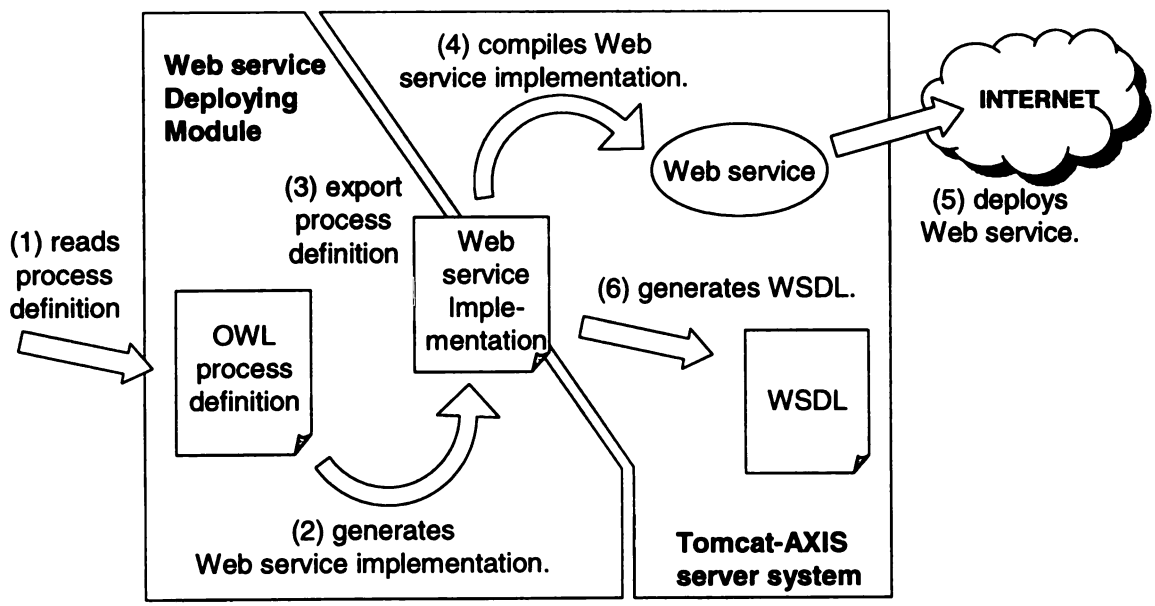


Figure 18 Deploying Web service on the Internet

The Web service deploy module performs three steps to prepare a Web service deployment; (1) Reads a process definition, (2) generates the implementation of Web service from the definition, (3) Copy the implementation of Web service at the specialized folder of the AXIS. Throughout step 1 and 2, the Web service deploy module uses the OWL parser to extract semantic information of process from the OWL document. Based on the semantic information, the Web service deploy module generates a jws file automatically. At step 3, the Web service deploy module copies the jws file at the specialized folder of the AXIS.

From the specialized folder of the AXIS, the jws file will be compiled by the AXIS compile engine and deployed as a Web service. When the jws file is compiled, the AXIS engine also generates WSDL document automatically. The example of the jws file and its WSDL document can be found in Appendix E.

6.3 Registering Web service

The MIDAS framework requires the service providers register their services at the public registry, and then, the service requester discovers it through the public registry. As mentioned previously, this happens only when inter-organizations collaborate. Intra-organizational collaboration doesn't require such registering and discovery.

As discussed in chapter 5, the remote procedure that enrolls the service has been implemented as part of the service registry. The Web service registering module interacts with that remote procedure of service registry to enroll its

service. By the remote procedure call, a new entry is registered in the registry. Figure 19 illustrates this enrolling process.

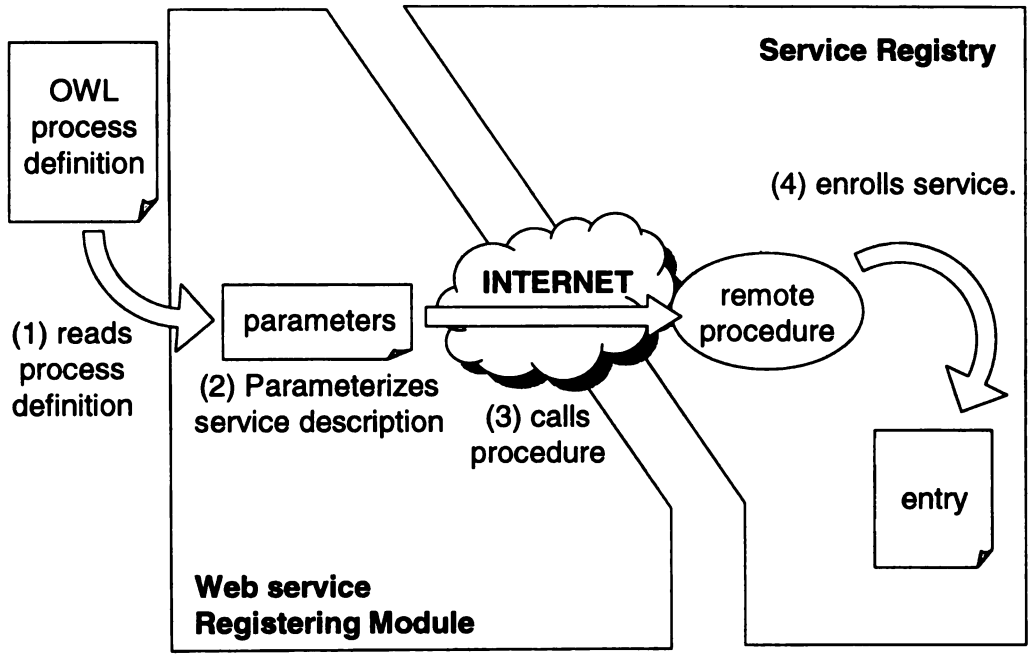


Figure 19 Registration of a service and format of entry

CHAPTER 7 MIDAS EXECUTION ENVIRONMENT USING WEB SERVICE

MIDAS makes a distinction between process definition details and execution details. As mentioned in chapter 6, the authoring environment presets process flow details. In the other hand, execution details are not preset and often change at runtime as the workflow is generated through the process enactment.

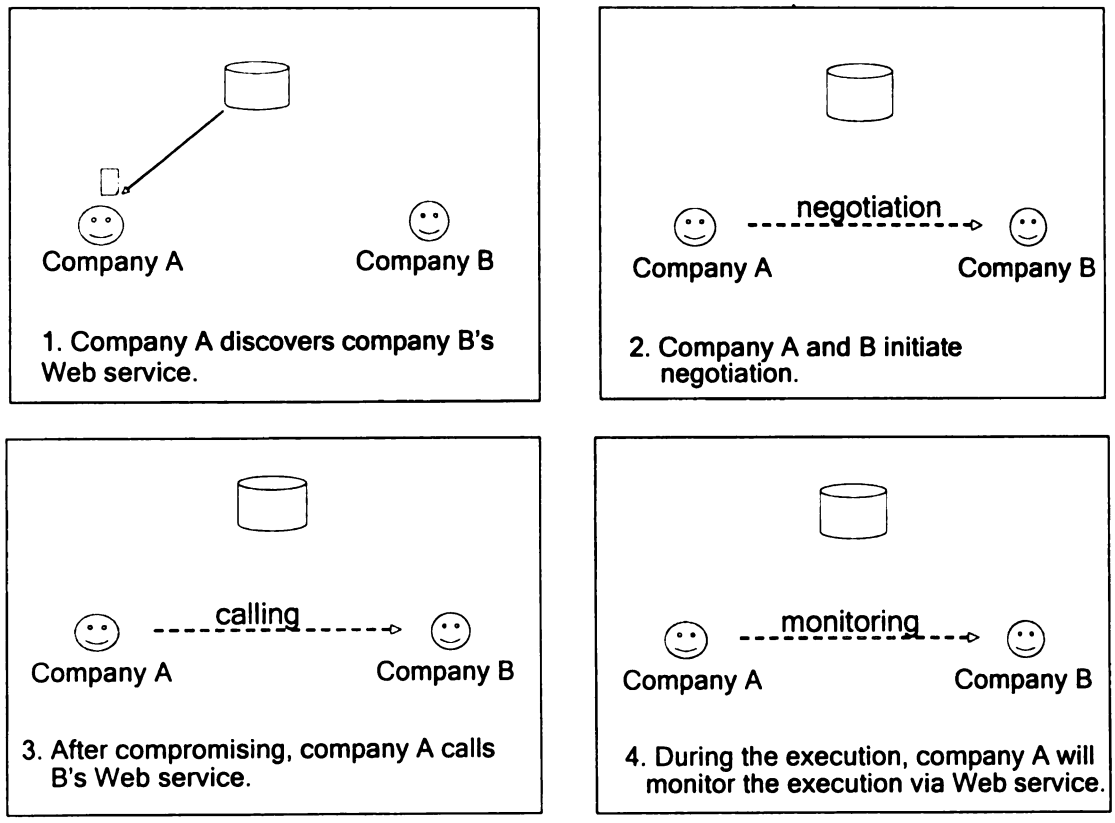


Figure 20 Overview of Web service-oriented interoperation in MIDAS

The execution environment enables user to discover a service, negotiate with a service before execution, execute the service, and monitor the execution of a

service (Figure 20). In this chapter, the author will discuss how the MIDAS execution environment builds up such execution detail at runtime.

7.1 Service Discovery

To discover a service that fulfills the service seeker's requirement, a service registry compares the semantic information registered at the entry with the parameters sent from the service seeker. Figure 21 shows how a service seeker discovers a service, which meets the service seeker's requirements. The service registry has a remote procedure that returns any services matching with the user's query. The service seeker calls this remote procedure with a parameterized query. The parameterized queries include the *type of service*, *input and output specifications* and the *pre- and post-conditions*. As described in chapter 5, the entry form also contains the type of service, input and output specifications and the pre- and post-conditions as referencing markers. The parameterized queries are compared to these referencing markers of a registered service at each entry, and all of matching entries are returned back to the service seeker.

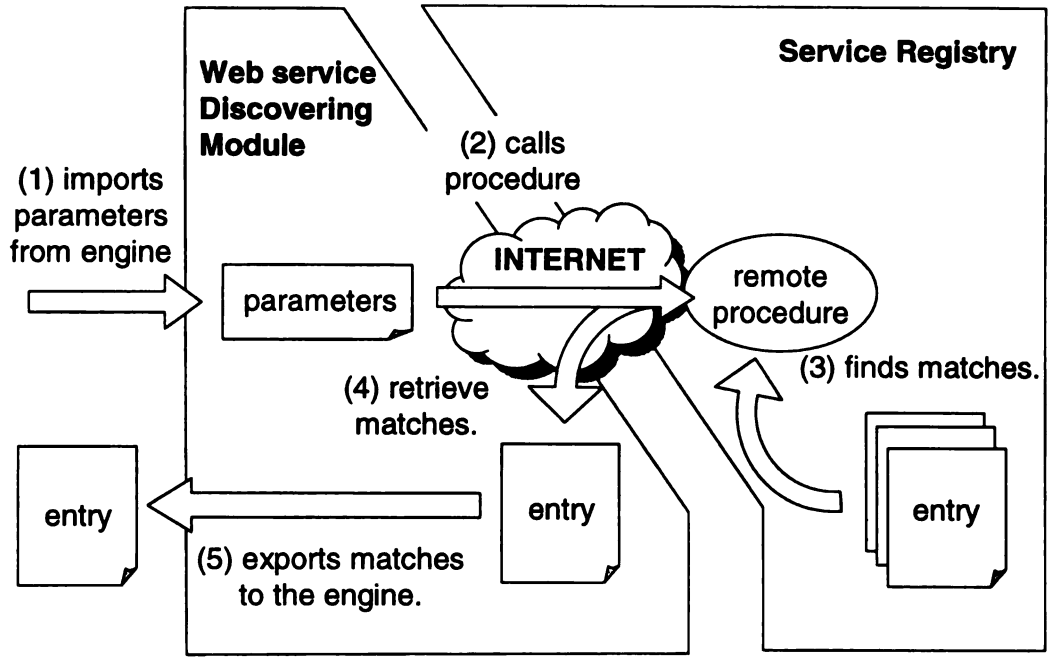


Figure 21 Overview of service discovery in the MIDAS

A service seeker may need the full scale of an OWL document to make a more sophisticated decision on selecting the best service among returned matches. A service seeker can refer to the full scale of an OWL document since entries retrieved from the registry include a link to the OWL document.

7.2 Negotiation for Collaboration

Discovery of a service provider does not necessarily involve perfect customization of the services to a requester's demands. Uncertainties are always bound to arise when a service requester encounters a service. Negotiation therefore helps a service requester meet the most ideal service by going through an iterative process to customize the service.

The negotiation proceeds through direct contact between a service requester and a service provider. MIDAS has a module for sending email notification to a specific participant, but unfortunately MIDAS does not support more than that currently.

Negotiation can be used for different purpose as well. In company-to-company collaboration on a process, there are always lots of things that must be configured together before collaboration begins. Such things could be the price of the product, deadline of the project or permission to access a certain resource. These jobs cannot be supported by other MIDAS facilities except the negotiation facility, but currently MIDAS doesn't have any sophisticated negotiation tool. I leave this job for future work.

7.3 Process Enactment Using Web service

In order to reflect dynamic nature of process management, MIDAS supports for an iterative process enactment using Web service. The Apply and the Roll Back are events invoked by a user or a machine agent during the execution of a process in the execution environment of the MIDAS framework. The iterative process enactment is defined as an iterative combination of Apply and Roll Back at run time.

The MIDAS process enactment consists of four steps: (1) Load up top-level process and initialize it. (2) Execute each task within top-level process. If you meet a logical task, expand it with sub-process. If you meet an atomic task,

invoke a tool and get the result (3) Execute tasks in an expanded process. (4) Check if expanded process meets constraints, and if not, rollback and reapply.

The process enactment engine is responsible for control in the above steps, but the process enactment engine itself does not have functionality to communicate with the Web service. The Web service calling module helps this process.

The MIDAS execution environment allows three enactment patterns using a Web service. (1) A logical task can be assigned to other user via Web service. (2) Someone's process definition can be retrieved through a Web service, and the service requester enacts imported process. In this case, the Web service works like a simple public process library. (3) Tool can be invoked via Web service. All those enactment patterns except (2) are **asynchronous** as discussed in chapter 4.

7.3.1 Task assignment using Web service

When the process enactment engine meets a logical task at the time of enacting the process, it should decide weather it enacts this logical task by itself or delegate its execution to the other user. The terminology of MIDAS framework defines the former case as ***Apply***, and later case as ***Assignment***. The MIDAS framework utilizes Web service to make the *assignment of a logical task* across the heterogeneous system from different vendors.

Since the engine itself does not have Web service interoperability, it leans on the *Web service calling module* to interact with the Web service. The logical service discovered through the service discovery stage will be invoked by the Web service calling module across the Internet. The enactment engine is responsible for providing input data if it is necessary. The Web service got call from the service provider initiates the enactment. After the service provider completes its process enactment, the service requester's engine retrieves the enactment results and updates its process flow.

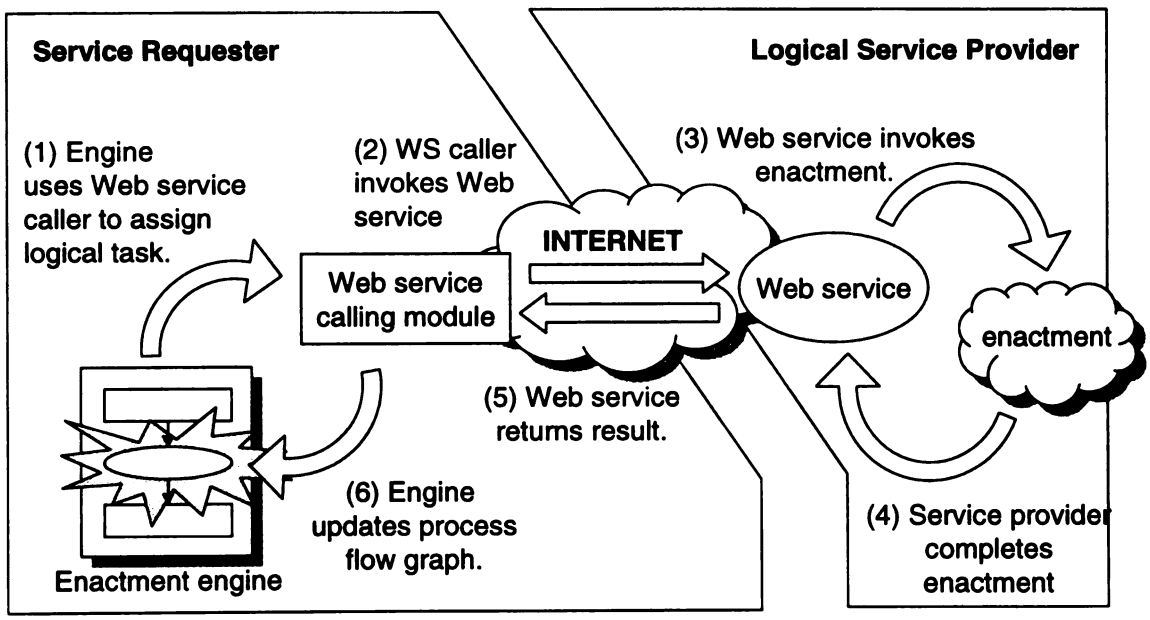


Figure 22 Overview of task assignment using Web service

Figure 23 shows details of Service requester-Web service interaction. The MIDAS implements the *invoke-then-listening asynchronous scenario* for this interaction. A logical service provider must make his Web service implementing at least following 4 operations to realize the invoke-then-listening task

assignment: *provideInput( )*, *invokeEnactment( )*, *getGragh( )* and *getOutput( )*. The provideInput( ) delivers input data to the service and turns the un-initialized state of the service into the ready state. The invokeEnactment( ) makes the service begin to work on enactment. The service requester waits for a call back after calling this operation, and the call back will return to the service requester when the service completes its job. Once the call back arrives, the service requester retrieves a process flow graph and output data by calling getGraph( ) and getOutput( ).

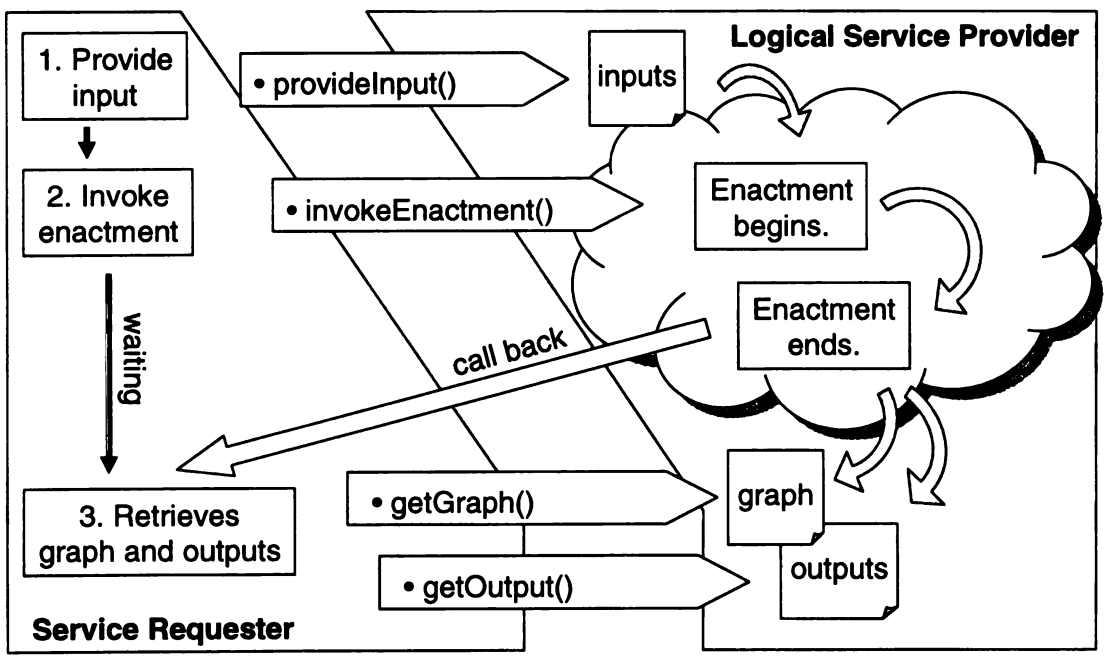


Figure 23 Detail view of task assignment using Web service

7.3.2 Use Web service as public process library

When a logical service is utilized as a public process library, it simply returns the definition of process flow, which is stored at the service-side repository. Then, the definition will be loaded up as a top-level process or used to expand a logical task (figure 24). Because the return of feedback to the service requester will be immediate, the asynchronous Web service calling isn't necessary in this case. The getGraph( ) will be used to retrieve a process definition from the service.

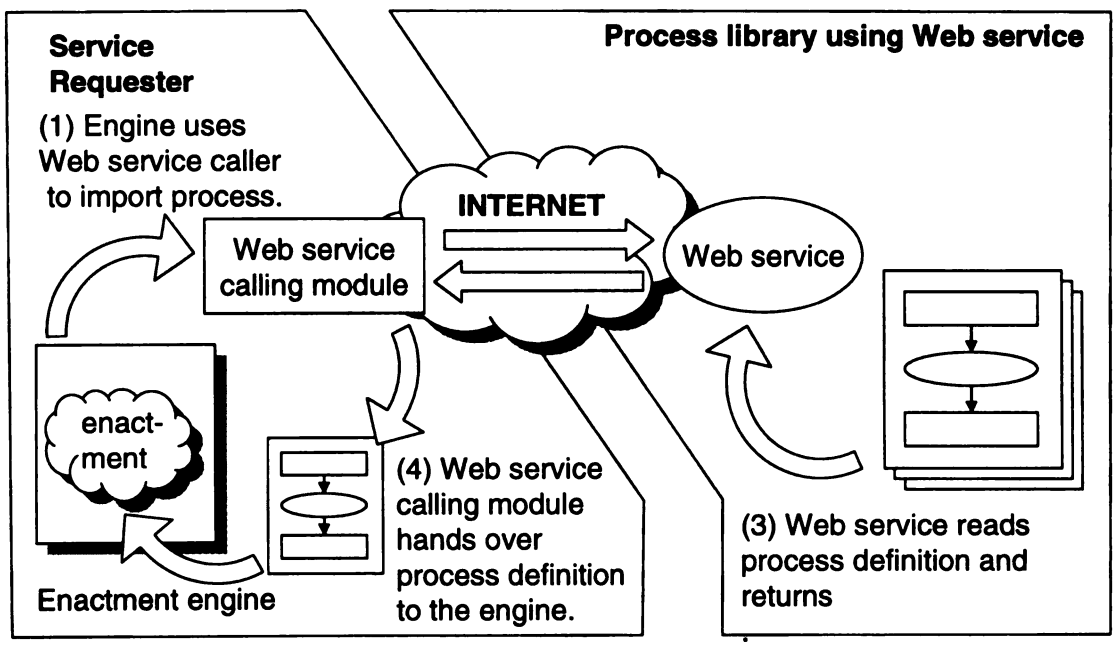


Figure 24 Overview of using Web service as process library

7.3.3 Tool invoking by Web service

The tool invoking through a Web service proceeds in the similar way that task assignment through Web service does (Figure 25). The Web service calling module is also involved to help the enactment engine in this case. The

enactment engine is responsible for providing input data if it is necessary. The enactment engine calls the Web service to initiate tool. After the tool completes its process enactment, the service requester's engine retrieves output data from the service and updates its process flow.

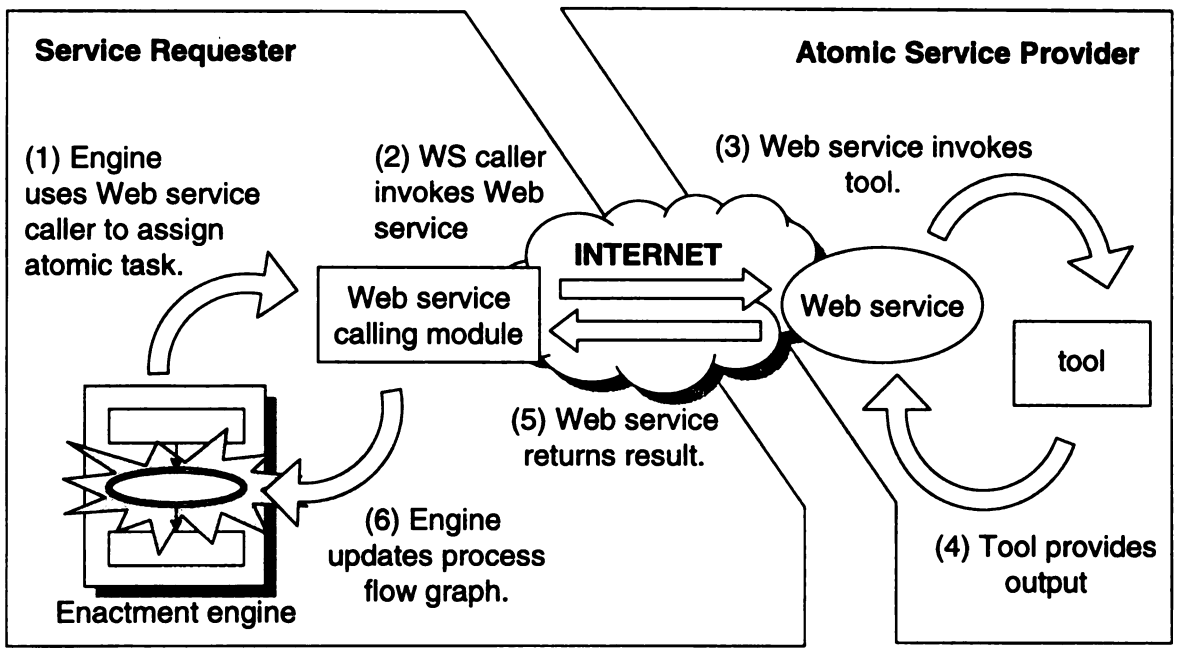


Figure 25 Overview of tool invoking using Web service

A atomic service provider must make his Web service implementing at least following 3 operations to realize the invoke-then-listening task assignment: *provideInput( )*, *invokeEnactment( )*, and *getOutput( )*. As seen in the figure 26, the interaction between atomic service and the service requester is almost similar except the absence of *getGraph( )*.

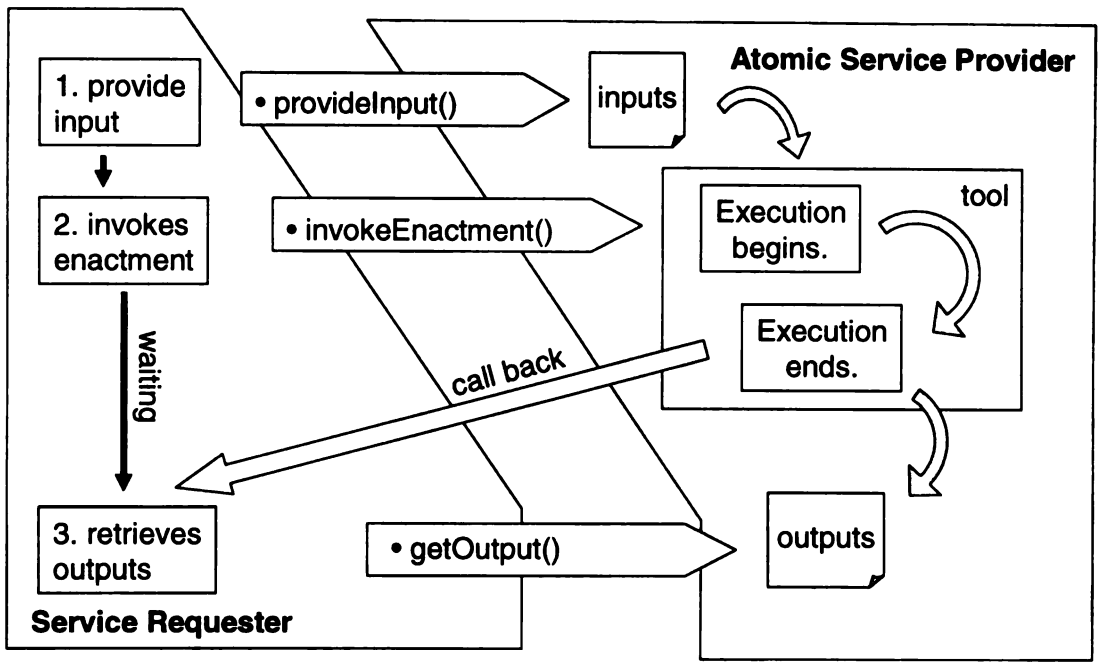


Figure 26 Detail view of tool invoking using Web service

7.4 Monitoring Enactment

During the process enactment is going on across the heterogeneous systems, users may want to monitor its execution status. Since the process being configured could possibly be a complex and large graph, visualizing the process being executed is not simple job. Furthermore, such visualization must be inter-understandable across heterogeneous workflow management systems. The MIDAS framework realizes the global visualization of collaborative enactment by adapting monitor model of SOM. An OWL document using terms defined by monitor model of SOM is capturing execution status of process graph.

As seen in Figure 27, once the enactment begins at the service provider side, the enactment engine at the provider's side begins to updates its graph with new

execution results. Whenever the graph is updated, the engine also updates the OWL document that captures the shape of process flow and execution status of each task composing the service provider's process.

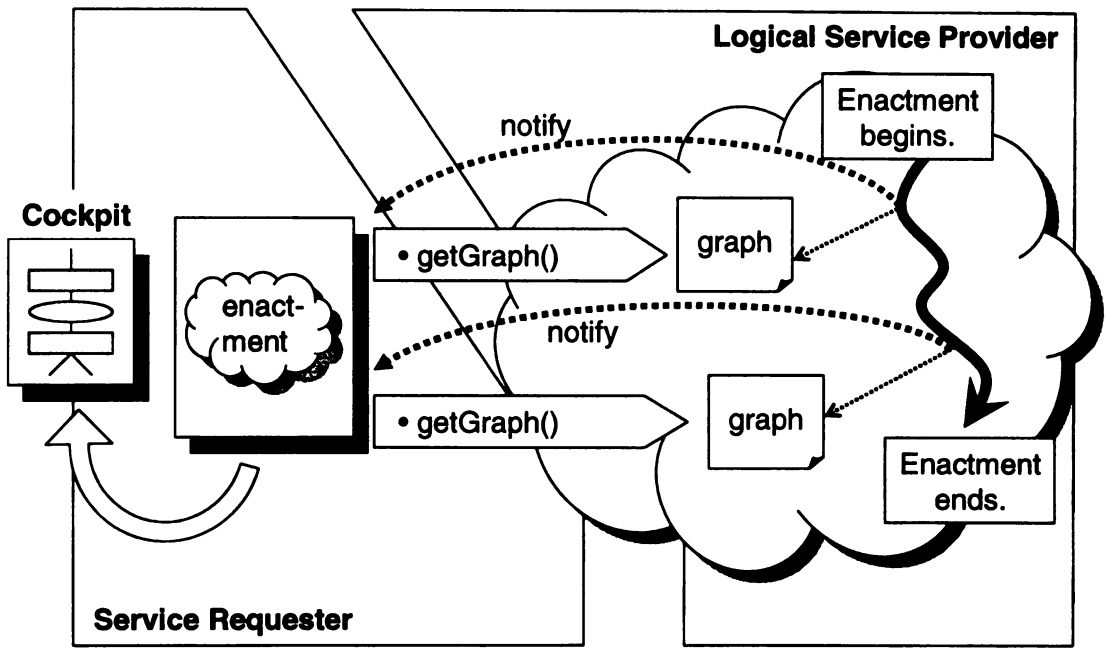


Figure 27 Overview of monitoring using Web service

The getGraph( ) operation returns this OWL document to the service requester. The service provider is responsible for notifying his process updates to the service requester, and the service requester call getGraph( ) operation to retrieve the OWL document whenever he gets the notification. The service requester's enactment engine makes the cockpit visualizing the graph after parsing its information.

APPENDIX A. JAVA Package for SOM

All following java sources are the highlighted feature of JAVA package for SOM.

This JAVA package is provided as "services.jar".

Only the declaration of constructer and public method are shown here. Every details have been omitted.

service.java

```
package services;


// Imports
import java.io.Serializable;
import java.util.Vector;


public class service implements Serializable {


  // Fields
  public String name;
  public String owner;
  public Vector preconditionList;
  public Vector postconditionList;
  public Vector inputList;
  public Vector outputList;
  public String URL;
  public String semanticURL;
```

```
  // Constructors
  public service(String name) { ... }
  public service() { ... }

  // Methods
  public void addPreCondition(condition cond) { ... }
  public void addPostCondition(condition cond) { ... }
  public void addinputList(Spec input) { ... }
  public void addoutputList(Spec input) { ... }
  public boolean equals(service Service) { ... }
}
```

atomicservice.java

```
package services;

public class atomicservice extends service {

  // Constructors
  public atomicservice(String name) { ... }
}
```

logicalservice.java

```
package services;


// Imports
import java.util.Vector;


public class logicalservice extends service {


  // Fields
  public Vector scList;
  public servicecomposite appliedSC;


  // Constructors
  public logicalservice(String name) { ... }


  // Methods
  public void addServicecomposite(servicecomposite sc) { ... }
}
```

Spec.java

```
package services;


// Imports
import java.io.Serializable;


public class Spec implements Serializable {


  // Fields
  public String name;
  public String data;


  // Constructors
  public Spec(String name) { ... }
}
```

condition.java

```
package services;


// Imports
import java.io.Serializable;


public class condition implements Serializable {


  // Fields
  public String name;


  // Constructors
  public condition(String name) { ... }
}
```

servicecomposite.java

```
package services;


// Imports
import java.io.Serializable;
import java.util.Vector;


public class servicecomposite implements Serializable {


  // Fields
  public String name;
  public String type;
  public String status;
  public Vector CSlist;


  // Constructors
  public servicecomposite() { ... }
  public servicecomposite(String name) { ... }
}
```

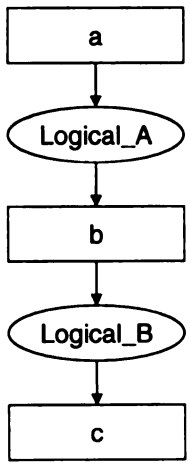
componentservice.java

```
package services;


// Imports
import java.io.Serializable;


public class componentservice implements Serializable {


 // Fields
 public String linkFrom;
 public String linkTo;
 public String status;
 public servicecomposite appliedBy;
 public service Service;
 public String name;
 public String type;


 // Constructors
 public componentservice() { ... }
 public componentservice(service Service) { ... }
 public componentservice(service Service, String linkFrom, String linkTo) { ... }
```

```
  // Constructors
  public componentservice() { ... }
  public componentservice(service Service) { ... }
  public componentservice(service Service, String linkFrom, String linkTo) { ... }

  // Methods
  public void setStatus(String ststus) { ... }
  public void setLinkFrom(String linkFrom) { ... }
  public void setLinkTo(String linkTo) { ... }
}
```

APPENDIX B. Converting Graph to Process Definition

In this example, the service flow illustrated in (a) can be represented by an OWL markup in (b). As you see in (b), the LogicalService or AtomicService must be defined ahead of ComponentServices. According to the definition, class Service and ComponentService are equivalent, so the ID of LogicalService or AtomicService can be referred from ComponentService tag.



(a) a service flow

```
<service:LogicalService rdf:ID="Logical_A">
    <service:hasInput>
      <service:SpecList rdf:parseType="collection">
        <service:Spec rdf:resource="#a"/>
      </service:SpecList>
    </service:hasInput>
```

```
    <service:hasOutput>
      <service:SpecList rdf:parseType="collection">
        <service:Spec rdf:resource="#b"/>
      </service:SpecList>
    </service:hasOutput>
</service:LogicalService>
<service:LogicalService rdf:ID="Logical_B">
    <service:hasInput>
      <service:SpecList rdf:parseType="collection">
        <service:Spec rdf:resource="#b"/>
      </service:SpecList>
    </service:hasInput>
    <service:hasOutput>
      <service:SpecList rdf:parseType="collection">
        <service:Spec rdf:resource="#c"/>
      </service:SpecList>
    </service:hasOutput>
</service:LogicalService>
<service:ServiceComposite rdf:parseType="collection">
 <service:ComponentService rdf:ID="Logical_A">
    <service:linkFrom/>
    <service:linkTo>
      <service: ComponentService rdf:resource="#Logical_B"/>
```

```
    </service:linkTo>
  </service:ComponentService>
  <service:ComponentService rdf:ID="Logical_B">
    <service:linkFrom>
      <service:LogicalService rdf:resource="#Logical_A"/>
    </service:linkFrom>
    <service:linkTo/>
  </service:ComponentService>
</service:ServiceComposite>
```

(b) OWL markup for the service flow

APPENDIX C. Example of Process Definition

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
  <!ENTITY rdf     "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs    "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY xsd     "http://www.w3.org/2001/XMLSchema">
  <!ENTITY daml    "http://www.daml.org/2001/03/daml+oil">
  <!ENTITY service "MIDASserviceModel.daml">   ]>
<rdf:RDF
  xmlns:rdf=     "&rdf;#"
  xmlns:rdfs=    "&rdfs;#"
  xmlns:xsd =     "&xsd;#"
  xmlns:daml = "&daml;#"
  xmlns:service = "&service;#"
  xmlns =         "&DEFAULT;#"
>

<!-- definition of Specipications -->
<service:Spec rdf:ID="EngineSpec"/>
<service:Spec rdf:ID="WiringSpec"/>
<service:Spec rdf:ID="ControllerSpec"/>
```

```
<!-- definition of LogicalService -->
<!-- definition of "ControllerDsg" -->
<service:LogicalService rdf:ID="ControllerDsg">
 <service:hasInput>
  <service:Spec rdf:resource="#EngineSpec"/>
 </service:hasInput>
 <service:hasOutput>
  <service:Spec rdf:resource="#WiringSpec"/>
  <service:Spec rdf:resource="#ControllerSpec"/>
 </service:hasOutput>
 <service:hasAlternativeChoices>
  <service:AlternativeChoice rdf:parseType="collection">
   <service:ServiceComposite
rdf:resource="def_CntProd2.owl#CntProd2"/>
  </service:AlternativeChoice>
 </service:hasAlternativeChoices>
</service:LogicalService>

<!-- definition of AtomicService -->
<!-- definition of "TimeMPLXModel" -->
<service:AtomicService rdf:ID="TimeMPLXModel">
 <service:hasInput>
  <service:Spec rdf:resource="#modelSpec"/>
```

```
 </service:hasInput>
 <service:hasOutput>
  <service:Spec rdf:resource="#MPSpec"/>
 </service:hasOutput>
</service:AtomicService>

<!-- definition of AtomicService -->
<!-- definition of "MicroProcess" -->
<service:AtomicService rdf:ID="MicroProcess">
 <service:hasInput>
  <service:Spec rdf:resource="#modelSpec"/>
 </service:hasInput>
 <service:hasOutput>
  <service:Spec rdf:resource="#MPSpec"/>
 </service:hasOutput>
</service:AtomicService>

<!-- definition of AtomicService -->
<!-- definition of "Intrument" -->
<service:AtomicService rdf:ID="Intrument">
 <service:hasInput>
  <service:Spec rdf:resource="#MPSpec"/>
```

```
  </service:hasInput>
  <service:hasOutput>
    <service:Spec rdf:resource="#ControllerSpec"/>
    <service:Spec rdf:resource="#WiringSpec"/>
  </service:hasOutput>
</service:AtomicService>


</rdf:RDF>
```

APPENDIX D. WSDL for Service Registry

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
targetNamespace="http://localhost:8080/axis/services/SR"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://localhost:8080/axis/services/SR"
xmlns:intf="http://localhost:8080/axis/services/SR"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns1="http://sr" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><wsdl:types><schema
targetNamespace="http://localhost:8080/axis/services/SR"
xmlns="http://www.w3.org/2001/XMLSchema"><import
namespace="http://schemas.xmlsoap.org/soap/encoding/"/><complexType
name="ArrayOf_xsd_string"><complexContent><restriction
base="soapenc:Array"><attribute ref="soapenc:arrayType"
wsdl:arrayType="xsd:string[]"/></restriction></complexContent></complex
Type></schema><schema targetNamespace="http://xml.apache.org/xml-
soap" xmlns="http://www.w3.org/2001/XMLSchema"><import
namespace="http://schemas.xmlsoap.org/soap/encoding/"/><complexType
name="Vector"><sequence><element maxOccurs="unbounded"
minOccurs="0" name="item"
type="xsd:anyType"/></sequence></complexType></schema><schema
targetNamespace="http://sr"
xmlns="http://www.w3.org/2001/XMLSchema"><import
namespace="http://schemas.xmlsoap.org/soap/encoding/"/><complexType
name="entry"><sequence><element name="name" nillable="true"
type="xsd:string"/><element name="serviceURL" nillable="true"
type="xsd:string"/><element name="owlURL" nillable="true"
type="xsd:string"/><element name="inputs" nillable="true"
type="apachesoap:Vector"/><element name="outputs" nillable="true"
type="apachesoap:Vector"/><element name="inputStr" nillable="true"
type="xsd:string"/><element name="outputStr" nillable="true"
type="xsd:string"/><element name="owner" nillable="true"
type="xsd:string"/><element name="type" nillable="true"
type="xsd:string"/><element name="processURL" nillable="true"
type="xsd:string"/></sequence></complexType></schema></wsdl:types>
```

```
<wsdl:message name="getServiceResponse">
  <wsdl:part name="getServiceReturn" type="xsd:string"/>
 </wsdl:message>
 <wsdl:message name="registerServiceResponse1">
  <wsdl:part name="registerServiceReturn" type="xsd:string"/>
 </wsdl:message>
 <wsdl:message name="registerServiceRequest">
  <wsdl:part name="Entry" type="tns1:entry"/>
 </wsdl:message>
 <wsdl:message name="getServiceRequest1">
  <wsdl:part name="inputs" type="impl:ArrayOf_xsd_string"/>
  <wsdl:part name="outputs" type="impl:ArrayOf_xsd_string"/>
  <wsdl:part name="preCond" type="impl:ArrayOf_xsd_string"/>
  <wsdl:part name="posrCond" type="impl:ArrayOf_xsd_string"/>
 </wsdl:message>
 <wsdl:message name="registerServiceRequest1">
  <wsdl:part name="entryStr" type="xsd:string"/>
 </wsdl:message>
 <wsdl:message name="mainRequest">
  <wsdl:part name="args" type="impl:ArrayOf_xsd_string"/>
 </wsdl:message>
 <wsdl:message name="registerServiceResponse">
  <wsdl:part name="registerServiceReturn" type="xsd:string"/>
 </wsdl:message>
 <wsdl:message name="getServiceResponse1">
  <wsdl:part name="getServiceReturn" type="impl:ArrayOf_xsd_string"/>
 </wsdl:message>
 <wsdl:message name="mainResponse">
 </wsdl:message>
 <wsdl:message name="getServiceRequest">
  <wsdl:part name="name" type="xsd:string"/>
 </wsdl:message>
```

```
<wsdl:portType name="SRWS">
   <wsdl:operation name="main" parameterOrder="args">
     <wsdl:input message="impl:mainRequest" name="mainRequest"/>
     <wsdl:output message="impl:mainResponse"
name="mainResponse"/>
   </wsdl:operation>
   <wsdl:operation name="getService" parameterOrder="name">
     <wsdl:input message="impl:getServiceRequest"
name="getServiceRequest"/>
     <wsdl:output message="impl:getServiceResponse"
name="getServiceResponse"/>
   </wsdl:operation>
   <wsdl:operation name="getService" parameterOrder="inputs outputs
preCond posrCond">
     <wsdl:input message="impl:getServiceRequest1"
name="getServiceRequest1"/>
     <wsdl:output message="impl:getServiceResponse1"
name="getServiceResponse1"/>
   </wsdl:operation>
   <wsdl:operation name="registerService" parameterOrder="Entry">
     <wsdl:input message="impl:registerServiceRequest"
name="registerServiceRequest"/>
     <wsdl:output message="impl:registerServiceResponse"
name="registerServiceResponse"/>
   </wsdl:operation>
   <wsdl:operation name="registerService" parameterOrder="entryStr">
     <wsdl:input message="impl:registerServiceRequest1"
name="registerServiceRequest1"/>
     <wsdl:output message="impl:registerServiceResponse1"
name="registerServiceResponse1"/>
   </wsdl:operation>
  </wsdl:portType>
```

```
<wsdl:binding name="SRSoapBinding" type="impl:SRWS">
   <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="main">
     <wsdlsoap:operation soapAction=""/>
     <wsdl:input name="mainRequest">
       <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://sr" use="encoded"/>
     </wsdl:input>
     <wsdl:output name="mainResponse">
       <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://localhost:8080/axis/services/SR" use="encoded"/>
     </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="getService">
     <wsdlsoap:operation soapAction=""/>
     <wsdl:input name="getServiceRequest">
       <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://sr" use="encoded"/>
     </wsdl:input>
     <wsdl:output name="getServiceResponse">
       <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://localhost:8080/axis/services/SR" use="encoded"/>
     </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="getService">
     <wsdlsoap:operation soapAction=""/>
     <wsdl:input name="getServiceRequest1">
                 <wsdlsoap:body
```

```
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://sr" use="encoded"/>
    </wsdl:input>
    <wsdl:output name="getServiceResponse1">
     <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://localhost:8080/axis/services/SR" use="encoded"/>
    </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="registerService">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="registerServiceRequest">
     <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://sr" use="encoded"/>
    </wsdl:input>
    <wsdl:output name="registerServiceResponse">
     <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://localhost:8080/axis/services/SR" use="encoded"/>
    </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="registerService">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="registerServiceRequest1">
     <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://sr" use="encoded"/>
    </wsdl:input>
    <wsdl:output name="registerServiceResponse1">
```

```
<wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://localhost:8080/axis/services/SR" use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="SRWSService">
    <wsdl:port binding="impl:SRSoapBinding" name="SR">
      <wsdlsoap:address location="http://localhost:8080/axis/services/SR"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

APPENDIX E. Example of jws codes and its WSDLs

```
import java.io.*;
import java.util.Vector;


public class CylinderDsg {
  String path = "C:\\Documents and Settings\\Hong Suk Jung\\My
Documents\\Company_A\\ CylinderDsg";


  public String provideInput ( String input1, String input2 ) throws Exception {
    OutputStream fo;
    fo = new FileOutputStream(path + "\\Input1.txt");
    byte b1[] = input1.getBytes();
    fo.write(b1);
    fo.close();


    return "Delivery was successfully.";
  }
```

```
public String [] getOutput() {
  try {
    InputStream fi;
    fi = new FileInputStream(path + "\\Outputs.txt");
    byte b[] = new byte[fi.available ()];
    fi.read(b);
    String contents = new String(b);
    fi.close();
    String outputs[] = {contents};
    return outputs;
  }
  catch (FileNotFoundException ex) {
    return new String [] {"Output data has not been ready."};
  }
  catch (IOException ex) {
    return null;
  }
}
```

```
  public String invokeEnacting() throws Exception {
   OutputStream fo;
   String temp = "MakeCalendar has been invoked.";
   fo = new FileOutputStream(path + "\\Invoked.txt");
   byte b1[] = temp.getBytes();
   fo.write(b1);
   fo.close();
   return "Enactment has been invoked.";
  }

  public String getGraph() throws Exception {
   InputStream fi = new FileInputStream(path + "\\Graph.txt");
   byte b[] = new byte[fi.available()];
   fi.read(b);
   String contents = new String(b);
   fi.close();
   return contents;
  }
}
```

SUMMARY

Web service can give valuable benefits to the collaborative workflow management systems; a barrier-less interoperability among heterogeneous system, high modularity, and portability. The manufacturing processes of each collaborative system can be posted as a globally-understandable service by using Web service technology and used by other systems. To make different systems to understand each other's process, the service-oriented process model has been proposed. The service-oriented process model provides the fundamental foundation of globally-acceptable management model for distributed process as service. In this model, the OWL, which is a standardized language for Web ontology, has been used to represent the process and describe execution status of process in this model. The Web service-oriented process model has been successfully deployed in the MIDAS framework. The authoring environment of MIDAS enables users to create and advertise the OWL process definition and deploys Web service generated based on the OWL definition. The execution environment of MIDAS enables users to search, select and use the posted service. The MIDAS has three execution styles using Web service: invoking tool application via Web service, assigning a logical task via Web service, and using Web service as public production library.

BIBLIOGRAPHY

AberdeenGroup. Beating the Competition with Collaborative Product Commerce, Jun 2000, AberdeenGroup, Inc.

Alsop, S. The Dawn of E-Service. Fortune, Nov 9, 1998, pp. 243-244

Baldwin, R. and Chung, M.J. Design Methodology Management: A Formal Approach, IEEE Computer, February 1995, pp. 54-63

Bloomberg, Jason. Web services and a New Approach to Software Development. Rational Software. 2002.
http://www.therationaledge.com/content/apr_02/f_webServices_jb.jsp

Chung, M.J., and Kwon, P. A Web-based Framework for Design and Manufacturing a Mechanical System. DETC, Atlanta, Georgia. Sep. 1998.

Chung, M.J., Kwon, P. and Pentland, B. Design and Manufacturing Process Management in a Supply Chain Environment (2003) Scalable Enterprise Systems Research, edited by Vittal Prabhu and Sounder Kumara, Chapter 2; pp. 33-64, Kluwer Academic Publishers,Boston, MA. 2003

Chung, M.J., Kwon, P. and Pentland, B. Making Process Visible: A Grammartical Approach to Managing Design Processes. (2002) ASME Transaction, Journal of Mechanical Design. vol. 124, 364-374

Cohen, Frank. Understanding Web service interoperability. IBM. 2002.
http://www-106.ibm.com/developerworks/webservices/library/ws-inter.html

Ding, Y., Fensel, D., Klein, M., and Omelayenko, B., "The semantic web: yet another hip?" Data & Knowledge Engineering, Vol. 41, No. 2, pp. 205-228, 2002.

Fensel, D., Horrocks, I., Harmelen, F., McGuinness, D. L., and Patel-Schneider, P. F., "The semantic web - oil: an ontology infrastructure for the semantic web", IEEE Intelligent Systems & Their Applications, Vol. 16, No. 2, pp. pp. 38-45, 2001.

Hendler, J., "The Semantic Web - Agents and the Semantic Web," IEEE Intelligent Systems & Their Applications. Vol. 16, No. 2, pp. 30-37, 2001.

IBM, "Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications", Rational Whitepaper, 2003.

Lavana, H., Khetawat, A., Brglez, F., and Kozminski, K., "Executable Workflows: A Paradigm or Collaborative Design on the Internet", Proceedings of the 34th ACM/IEEE Design Automation Conference, June 1997.

McMillan, R., "IDC: Web Service to Enable $4.3B Hardware Market by 2007", Computerworld, 23 May 2003

Paolucci, M., Srinivasan, N., Sycara, K., Solanki, M., Lassila, O., McGuinness, D., Denker, G., Martin, D., Parsia, B., Sirin, E., Payne, T., McIlraith, S., Hobbs, J., Sabou, M., and McDermott, D., "OWL-S", http://www.daml.org/services/owl-s/1.0/owl-s.pdf

Peltz, C., "Web Services Orchestration and Choreography", IEEE Computer (October), pp. 46-52, 2003.

Qin, Y. (2002). Manufacturing Infrastructure and Design Automation System (MIDAS) with XML representation. Computer Science and Engineering. East Lansing, Michigan State University.

Schey, J. A., (1987), Introduction to Manufacturing Processes, 2nd edition, McGraw-Hill, New York, NY.

Shapiro, R., "A Comparison of XPDL, BPML, and BPEL4WS." xml.coverpages.org/Shapiro-XPDL.pdf, 2002.

Sun Microsystems, Inc. (2002) Web Services Made Easier.

UDDI.Org. UDDI specification version 2.04 http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htmMark Klein, Abramham Berstein. Searching for services on semantic web using process Ontologies. International semantic web working symposium, 2001

W3C Web Service Choreography Working Group Charter. 2002

W3C Web service descript group. Web Services Description Requirements (W3C Working Draft 28 October 2002) http://www.w3.org/TR/ws-desc-reqs/

W3C. Web Service Description Language(WSDL) 1.1

Weerawarana, S. and Francisco, C., "Business Process with BPEL4WS: Understanding BPEL4WS, Part1", http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol1/

WSCI. http://www.w3.org/TR/2002/NOTE-wsci-20020808/

Xerces2 Java Parser 2.6.2. http://xml.apache.org/xerces2-j/index.html