Middleware for the Autonomous Web Services (AWS)

Makoto Oya, Masaki Ito, Taisuke Kimura

Shonan Institute of Technology, 1-1-25, Tsujido Nishi-Kaigan, Fujisawa, 251-8511, Japan

Abstract. The purpose of the Autonomous Web Services (AWS) is to enable business transaction exchange in the Internet between systems having different business process models, by dynamically harmonizing them when the systems encounter. Based on the principles and the basic methods proposed in the previous researches such as [3], we succeeded in development of the experimental implementation of the AWS middleware. The AWS middleware consists of three software layers - the dynamic model harmonization layer, the application framework layer, and the messaging layer. This paper mentions the development principles, operation concepts, proposed specifications, detailed algorithms and test results of the AWS middleware that we developed. This success of implementation demonstrates the AWS's theoretical properness and its availability to real world applications, as well as the applicability of the improved model harmonization algorithm proposed in this paper.

Keywords: Web Services, autonomy, business process model, e-commerce

1 Introduction

The Web services is a well matured technology and widely used as the infrastructure for business transaction exchange between systems over the Internet. In the present Web services, it is prerequisite that a global business process model across relevant systems, like BPEL [6], is precisely defined in advance, and each system is built conforming to the global business process model (Fig.1 (a)). Therefore, a voluntarily and freely built system cannot participate in the concerning service. This prerequisite is very strong and not appropriate for ordinary business transactions except large scale or fixed form transactions. The Autonomous Web Services (AWS), whose concept was defined in [3], does not assume the existence of a predefined global business process model. It dynamically harmonizes (i.e. adjusts each other) business process models in individual systems when the systems encountered in the Internet (Fig.1 (b)). The AWS aims to realize that systems independently built and having different business models can freely exchange business transactions.

The AWS's theoretical backbone is the Dynamic Model Harmonization (DMH), proposed in [1][2], and systematized in [3]. In addition, the AWS's core technologies include the application framework based on model driven execution and the messaging mechanism as their infrastructure. Based on these preceding research results, we succeeded in development of the experimental implementation of the

AWS middleware. The purpose of the AWS middleware is to execute the AWS's complex mechanism hiding the implementation details from applications. It enables application developers easy to develop AWS applications only by coding business logics corresponding to input/output operations.

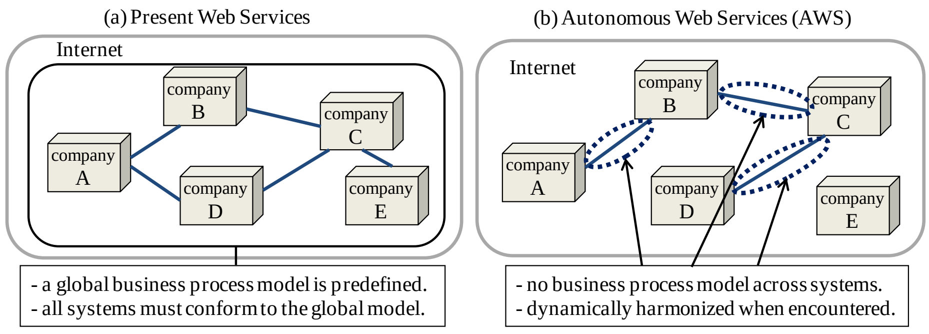

(a) Present Web Services
Internet
company B
company C
company A
company D
company E
- a global business process model is predefined.
- all systems must conform to the global model.
(b) Autonomous Web Services (AWS)
Internet
company B
company C
company A
company D
company E
- no business process model across systems.
- dynamically harmonized when encountered.

Fig. 1. Present Web Services and the Autonomous Web Services (AWS)

The structure of this paper is: Section 2 surveys the core technologies of the AWS in the previous works. It also contains a newly improved DMH algorithm. Section 3 is the main part of this paper and explains the development principle, operation concepts, specifications and detailed algorithm of the AWS middleware that we developed. Section 4 provides brief discussions and conclusions.

2 Essences of the Autonomous Web Services (AWS)

This section surveys the theoretical background and principles of the AWS. It also proposes the improved DMH algorithm.

2.1 Dynamic Model Harmonization (DMH)

The core principle of the AWS is the DMH (Dynamic Model Harmonization) (Fig.2). Systems export own business process models (BPMs). A BPM is a description of possible flow of input and output message sequences. An example is "ask estimation, and receive the estimation results, then, order, and receive its acceptance". The BPMs of both systems are exchanged when the systems encounter over the Internet. By the DMH algorithm, the original BPM is modified adjusting with the opponent's BPM. Then, the business transaction exchange starts using the modified BPM (called the harmonized BPM). In this way, business conversation between systems having different BPMs is executed with best efforts.

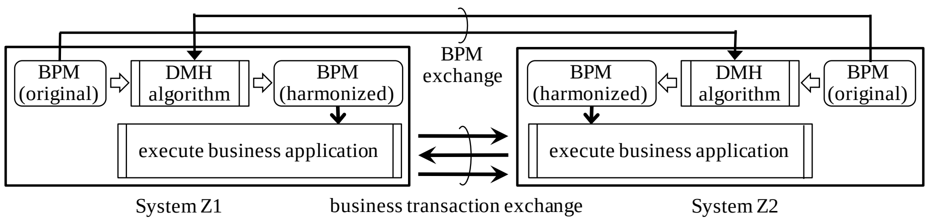


Fig. 2. Overview of the Dynamic Model Harmonization

BPM is formally defined as BPM = (*O*, *B*), where:

- *O* is a set of *operations op*, where *op* = (*pattern*, *format*), a *pattern* specifies 'output' or 'input', and a *format* specifies a format of message, and
- A *behavior B* is, in general, represented as a finite state machine, namely: $B = (S, \lambda, F, \Phi)$, where *S* is a set of states, λ ($\in S$) is a starting status, F ($\subset S$) is a set of final states, and Φ is a transition function.

Note that *O* corresponds to "interface" or "portType" in WSDL [8], and *pattern* corresponds to the MEP (message exchange pattern) though restricted 'input only' and 'output only'.

As discussed in [3], we assume a three-valued matching function t_match(*f*, *g*) is given from the outer environment, where *f* is an output and *g* is an input format, and

t_match(*f*, *g*) =	true	(if all instances of *f* match to *g*), or
	false	(if some instance of *f* does not match to *g*), or
	undefined	(if cannot determine either true or false).

Variety of implementation of t_match() is possible. A trivial one is: true if $f = g$ and false if $f \neq g$ when both *f* and *g* are in a same name space, and undefined when *f* and *g* are in different name spaces. This trivial t_match() is used at testing the middleware in this research. Implementation of t_match() is out of scope of this paper but another interesting research theme. Solutions applying ontology or semantic web have been proposed [2][5][4]. At the same time, it is valuable to report that the DMH properly harmonizes BPMs even in cases when such a simple t_match() is applied.

In this paper, we limit a behavior *B* is a non-deterministic automaton, as the previous studies did. Under this limitation, Φ is restricted as $S \times O \rightarrow S$. We introduced the following DMH algorithm improved from the previous researches [1][3].

Let an original BPM of the own system be *M* and a BPM received from the opponent system be *M'*, where $M = (O, (S, \lambda, F, \Phi))$ and $M' = (O', (S', \lambda', F', \Phi'))$. First, create $N = (P, (T, \mu, G, \Gamma))$ in the following steps:

- $P = \{ (o, o') \mid o \in O$ and $o' \in O'$ and $\text{o\_match}(o, o') \neq \text{false}\}$, where o_match(*o*, *o'*) is t_match(*fo*, *fo'*) if *o* is output and *o'* is input, and t_match(*fo'*, *fo*) if *o* is input and *o'* is output, and false if both *o* and *o'* are input or output. (*fo* and *fo'* are formats of *o* and *o'* respectively.)
- $T = S \times S'$, $\mu = (\lambda, \lambda')$, $G = F \times F'$.

- $\Gamma((s, s'), (o, o')) = \Phi(s, o) \times \Phi'(s', o')$, where $(s, s') \in T$ and $(o, o') \in P$.
- Remove all $\tau \in T$ and its relating paths from Γ that are not reachable from μ or do not reach to G. Remove elements of T and G that do not appear in the resulting Γ.

Then, create a harmonized BPM $Mh = (Oh, (Sh, \lambda h, Fh, \Phi h))$ as a "projection" of N:

- $Oh = \{ o \mid (o, o') \in P \}$, $Sh = (s \mid (s, s') \in T \}$, $\lambda h = \lambda$, $Fh = \{ s \mid (s, s') \in G\}$ and
- $\Phi h(s, o) = \cup^X \cup^Y \Gamma((s, s'), (o, o'))$ [where X: for all s' satisfying $(s, s') \in T$, and Y: for all o' satisfying $(o, o') \in P$.]

2.2 Model Driven Application Execution

As a consequence of BPM modification, the control flow of the application program must also be modified. The second problem is how to handle this situation. The following solution has been proposed in [3]:

- A user develops an application program as a set of process units (called AP segments) corresponding to each input/output operation.
- The AWS middleware successively transits a status of the harmonized BPM, and invokes an AP segment corresponding to an input/output operation at the current status. (see Fig.3)

2.3 Messaging Mechanism with VL Session

The third problem is on infrastructure to perform message exchange over the Internet/Web. It is well known that peer-to-peer asynchronous messaging is most appropriate for business message exchange. Many studies have been done and the protocol for Web services has already been standardized [9]. In addition to these well-known technologies, [3] has pointed out the necessity to manage very long sessions (VL sessions) and endurable queues. Fig.3 illustrates information flow from the DMH to peer-to-peer message sending/receiving through a VL session.

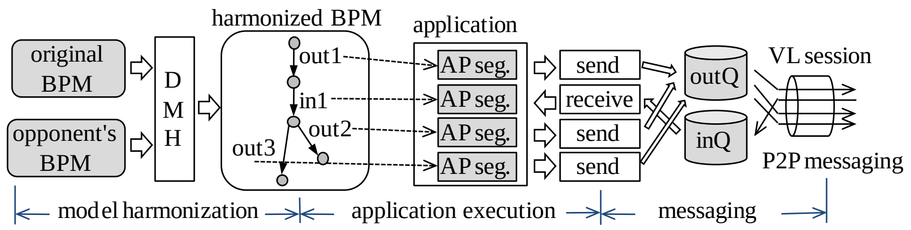


Fig. 3. DMH, Application Execution, and Messaging

3 AWS Middleware

This is the main part of this paper, explaining the development principles, operation concepts, specifications and detailed algorithms of the AWS middleware that we developed.

The purpose of the AWS middleware is to execute the AWS's complex mechanisms such as BPM modification, application flow modification and message sending/receiving protocol, and to hide such AWS's implementation details from application programs. Thus, an application developer can easily develop an AWS application that performs a series of transaction exchange in the AWS's way, only by describing a BPM and coding business logics corresponding to input/output operations. The AWS middleware consists of three software layers, the model harmonization layer, the application framework layer, and the messaging layer, which respectively correspond to the three phases in Fig.3.

3.1 Model Harmonization Layer

As Fig.3 shows, the role of the model harmonization layer is to get an original BPM description and an opponent's BPM description, execute the DMH algorithm, and generate a harmonized BPM. The harmonized BPM is passed to the application framework layer as a Java object, not as an external description. The issues on implementing the model harmonization layer were (a) a format of BPM description, (b) implementation of the DMH algorithm, and (c) a specification of a BPM object passed to the application framework layer. (b) was realized by implementing the algorithm in 2.1. The solutions to (a) and (c) are mentioned below in this section,.

3.1.1 BPM Description

We adopted XML encoding to describe a BPM, considering consistency with other Web technologies. BPM is formally defined as (O, B). O is a set of operations, which corresponds to a portType or an interface in WSDL. We introduce BPM description syntax for O simplifying the WSDL syntax. As for a behavior B, two types of description are considered - (i) in a state transition table form, (ii) in a regular expression form. (i) is a simple way and has a merit relationship between AP segments and an application control flow is simplified, but has a demerit a description tends to be long. On the other hand, (ii) realizes a short description and easy to understand, but has a problem timings of AP segment invocation are not explicit and may have difficulty in programming AP segment codes. Therefore, we adopted both types of description. Users can choose either convenient description type depending on the application. Fig.4 shows an example of BPM description in two types.

(a) State transition table type description

```
<BPModel>
  ..operations description, same as (b)..
<behavior>
<states>
  <state no="0">
   <next operation=" AskEstim">1</next>
  </state>
  <state no="1">
   <next operation ="RecEstim">2</next>
  </state>
  <state no="2">
   <next operation= ="Order">3</next>
   <next operation="Cancel">4</next>
  </state>
  <state no="3">
   <next operation= ="RecAccept">4</next>
  </state>
  <state no="4"></state>
 </states>
 <first>0</first>
 <last>4</last>
</behavior>
</BPModel>
```

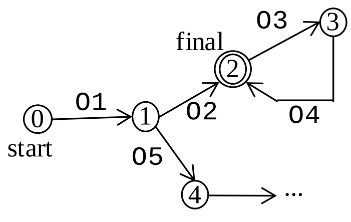
(b) Regular expression type description

```
<BPModel>
<operations>
  <operation name="AskEstim" format="Est01">
    <pattern>output</pattern> </operation>
  <operation name="RecEstim" format="Rep01">
   <pattern>input</pattern></operation>
   .....
</operations>
<behavior>
  <atom operation="AskEstim"/>
  <atom operation="RecEstim"/>
  <choice>
    <sequence>
      <atom operation="Order"/>
      <atom operation="RecAccept"/>
    </sequence>
    <atom operation="Cancel"/>
 </choice>
</behavior>
</BPModel>
```

Fig. 4. Two Types of BPM Description

3.1.2 BPM Object

A BPMS is internally represented by a Java object. Several methods are prepared to a BPM object for access in the framework layer. Important are init() to reset the current status to the initial status, getNextOps() to return an array of possible next operations when transits from the current status, and transit(*op*) to transit to the next status from the current status after executing an operation *op*. getNextOps() simply returns a set of next possible operations and does not causes state transition. When more than two operations are next possible, a desired operation can be selected by setNextOperations() by the AP segment. Note that a special operation 'term' is returned when the current status is final. Fig.5 shows an example of status transitions by transit() and operation sequences returned by getNextOps().



transition	state	NextOps
init()	0	01
transit(01)	1	02, 05
transit(02)	2	03, 'term'
transit(03)	3	04
transit(04)	2	03, 'term'
transit('term')	-	-

Fig. 5. Example of State Transition and Next Possible Operations

3.2 Application Framework Layer

The application framework layer controls the application execution flow driven by the harmonized BPM object, and performs the message input/output and the format translation in place of the application program. At the same time, it hides the AWS's underlining complex mechanism from application programs. This section mentions what functions are prepared for application developers and how these functions are realized inside the framework.

3.2.1 Functions prepared for Application Development

The body of the application framework layer is a class AWSFramework. A developer creates an application class inheriting AWSFramework. Section 2.2 explains an application is created as a set of AP segments. We implemented an AP segment as a Java method (called an AP method). To develop an application, a developer codes Java methods in an application class corresponding to each operation in the original BPM. Fig.6 is an example of application. This class contains three AP methods, apOrder(), apResponse() and apPay(). Only by writing such a simple program, the AWSFramework invokes an appropriate AP method along with state transition of the harmonized BPM, and behind performs complicated tasks including sending/receiving messages.

```
public class Ap extends AWSFramework {
   public void apOrder(OrderData out) {
      out.item = 'TV-45001'; out.qty = 6; }
   public void apResponse(ResResult in) {
      /* process delivery date (in.date) and price(in.price) */ }
   public void apPay(PaymentData out) {
      out.payDate = payment date; }
   publc Boolean dMatch(String data, Format format) {
      /* check whether data matches with format */ }
}
```

Fig. 6. Example of Application Program

We introduced a XML format "config" file to specify a mapping between operations and AP methods, considering the separation of protocol and programming [3]. Fig.7 is an example of config file corresponding to the program in Fig.6.

```
<operation name="Order"> <method>apOrder</method>
  <parameter>OrderData</parameter> </operation>
<operation name="Response"> <method>apResponse</method>
  <parameter>ResResult</parameter> </operation>
<operation name="Payment"> <method>apPay</method>
  <parameter>PaymentData</parameter> </operation>
```

Fig. 7. Example of Config File

A config file also includes classes to encapsulate details of message data format encoding, called a container class. In Fig.6 for example, AP methods (apOrder, apResponse and apPay) receives container classes (OrderData, ResResult and PaymentData) instead of message texts themselves. Container classes, coded by an application developer, must implement generateMessage() that generates a message text from a container object, and purseMessage() that creates contents of a container object from a message text. AWSFramework invokes them when performs message input/output. Two other methods, initialize() and terminate(), can be included in an application class. They are invoked right after the VL session starts and right before the VL session terminates. In addition, a special method dMatch() must be included which determines whether a message text matches with a given format.

3.2.2 Execution Mechanism in the Framework

The model harmonization layer passes a harmonized BPM objet to the AWSFramework. Successively transiting a status of the BPM object, the AWSFramework selects an appropriate AP method and invokes it with its container object cObj as a parameter. Fig.7 is a pseudo code outlining the process inside the AWSFramework.

```
(vlSession = new VLSession()).start();

bpm.init();
nextOps = bpm.getNextOps().clone();
invoke('initialize');

while(nextOps[0]!='term') {
   if(nextOps.length==1 and getPattern(nextOps[0])=='output')
{
      op=nextOps[0];
      bpm.transit(op);
      nextOps = bpm.getNextOps().clone();
      cObj = genContainerObj(op);
      invoke(getMethod(op),cObj);
      mess = cObj.generateMessage();
      vlSession.send(mess);
   }
   else if(getPattern(op)=='input' for(op:nextOps)) {
      mess = vlSession.receive();
      for(op:nextOps) if(dMatch(getFormat(op),mess)) break;
      bpm.transit(op);
      nextOps = bpm.getNextOps().clone();
      cObj = genContainerObj(op);
      cObj.purseMessage(mess);
      invoke(getMethod(op),cObj);
   }
}
invoke('terminate');
vlSession.terminate();

public getNextOperation() {return nextOps;}
public setNextOperation(ops) {nextOps = ops.clone();}
```

Fig. 8. Processing inside the AWSFramework

After generating and starting a VL session object, setting the status of bpm to the initial status, and invoking initialize(), it enters the main loop transiting the status using bpm.transit(). nextOps in Fig.7 is an array containing next operations. 'term' is set in nextOps if the current status is final. The default value of nextOps is bpm.getNextOps(), a sequence of operations that may occur from the current status. When the length of nextOps is 1 (that is, when next operation is deterministic), or when the length is >1 but all operation patterns are 'input', the next operation(s) are executable. Otherwise, the AP method can restrict possible operations using setNextOperation(). The first "if" clause in the main loop in Fig.7 is the process for 'output' operation and the second "else-if" clause is for 'input'. In the case of 'output', after generating a container object and invoking the AP method, it sends a message text created by generateMessage() by the send() method provided by the messaging layer. In the case of 'input', it first receives a message and determines a corresponding operation op using dMatch(), then injects data into a container object using purseMessage() and invokes the concerning AP method. Note that getMethod(), getContainerObj(), getPattern() and getFormat() are the methods to access the config information, which respectively return the corresponding AP method object, the container class object, the pattern ('input' or 'output') and the format name.

3.3 Messaging Layer

3.3.1 Peer-to Peer Asynchronous Messaging with VL Session

The mechanism of the messaging layer is completely hidden from applications. It provides the peer-to-peer, asynchronous, reliable and stored-forward-type messaging feature between application endpoints to the upper layer. A long term lasting session called a VL session is established between applications from the beginning to the end of a series of transactions (see Fig.3). The VL session is the concept introduced in [3]. The interface from the application framework layer is designed simple - just to use concise methods to a VL session object, e.g., start(), send, receive() and terminate().

3.3.2 Underlying System Configuration and Protocol

The messaging layer supports two types of underlying configuration of systems - a configuration (called a symmetric configuration) where both sides of system have Web servers (Fig. 9), and a configuration (called an asymmetric configuration) where only one side has a Web server (Fig. 10). The former is for usual business transaction exchange between ordinal enterprises, and the latter is for smaller configurations such as for a SME (small and medium enterprise) or a mobile system. In the symmetric configuration, a message data retrieved from the output queue (outQ in the figures) is sent by a HTTP request, and the HTTP response simply acknowledges the results. In the asymmetric configuration, the way of sending a message data from the side without a Web server (system A in Fig.10) to the side with a Web server (system B in Fig.10) is same as the symmetric configuration, but different when sending a message from system B to system A. System A, time to time, sends to system B a HTTP request asking download, and receives message data associated with the HTTP response if outQ in system B has data to be sent.

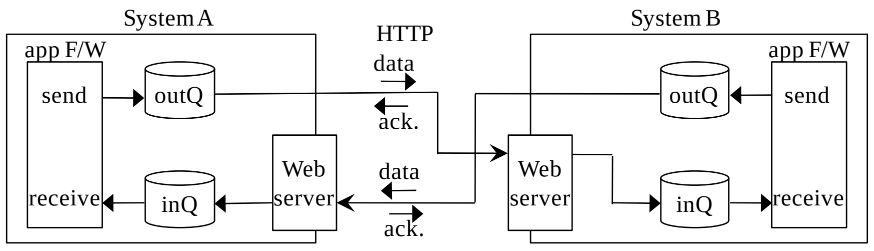


Fig. 9. Symmetric System Configuration

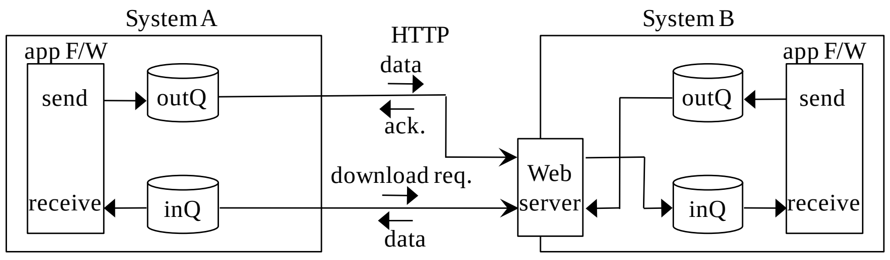


Fig. 10. Asymmetric System Configuration

The messaging protocol was implemented over HTTP and SOAP/Framework, as a simplified form of the ebXML Messaging standard [9]. The reason this standard is not directly applied is that the standard specification is too big and having unnecessary functions for this experimental implementation, and we needed some modification to permit plural number of downloading to improve performance of reliable messaging.

In addition, the followings were considered at implementing the messaging layer.

- Implementation of queues: Taking account of durability, queues were implemented upon DBMS (PostgreSQL). Eight tables are defined, enabling the retry control and the messaging sequence assurance required in reliable messaging.
- Asynchronization and performance: Multi process/thread structure is applied.
- VL session management: It is a policy of the AWS that no global server manages a session across systems. Therefore, a VL session is managed in individual systems cooperating with each other. A session ID, for example, is generated in conjunction with an opponent system's algorithm.

4 Discussion and Conclusions

4.1 Discussion

The implementation was done using Java. After completing basic software tests, we evaluate the developed AWS middleware from two viewpoints, artificial tests and benchmark tests. In the former viewpoint, theoretically comprehensive BPMs are created and the appropriateness and functionality of the the AWS middleware are verified and evaluated. On the other hand, benchmark tests (Table.1 shows some of them) are created simulating real trading applications and evaluate its availability. A portion of artificial and benchmark tests has been completed by now, and it was confirmed that the implementation of each layer is appropriate and properly works, and the introduced specifications are adequate and enough encapsulate the implementation details.

Table 1. Benchmark Tests (Samples)

Simple01	Basic transaction
Simple02	Multiple step transaction
Fork01	Transaction with a simple branching
Fork02	Multiple branching transaction
Loop01	Simple looping business process
Loop02	Composite looping business process

Let us evaluate the topics of the each layer this paper mentions. We proposed and developed two types of BPM description (see 3.1.1). This provides to application developers free choice of convenient way of description. The proposed specification of the application framework layer (3.2.1) is evaluated through testing as appropriate, and makes an application program codes simpler (such as in Fig.6). AP methods realized as Java methods are automatically invoked by the framework, and input/output data passed to/from applications are encapsulated. The executions mechanism in the framework (mentioned in 3.2.2) was verified through this experimental implementation and testing. At the same time, it was found that more study is necessary in cases an application consists of many processes and/or threads. Several considerations were done at implementing the messaging layer (as mentioned in 3.3), the properness of the solutions was verified through this development.

4.2 Conclusions

Basic ideas and theories of the AWS are the DMH, the application framework based on model driven execution, and the messaging mechanism as their infrastructure. In this study, we implemented the middleware for the AWS. The success of this experimental development proves properness and availability of these ideas, theories and relating technologies. The developed AWS middleware provides the full basic functionality of AWS, and hides the complex implementation details from application programs. The ultimate goal of the AWS is to provide the next generation Web services infrastructure enabling free and flexible business transaction exchanges among independently built autonomously managed systems. Future issues include the dynamic model harmonization among more than three systems, enhancement of capability of a state machine in a behavior, implementation of improved t_match() applying ontology for example, implementation of parallel business processes execution, and security.

Acknowledgement
This implementation was done with cooperation of our students: Hiroteru Otomo, Masamichi Hiramoto, and Kyohei Yoshikawa. This work was supported by the Grant-in-Aid for scientific Research, KAKENHI (21500110).

References

[1] M. Oya et al, "On Dynamic Generation of Business Protocols in Autonomous Web Services", IEICE transaction on Information and Systems, vol. J87-D-I, no. 8, pp. 824-832, 2004 (in Japanese); Systems and Computers in Japan, Wiley, Vol.37, No.2, pp. 37-45, 2006.

[2] M. Oya, and M. Ito: "Dynamic Model Harmonization between Unknown eBusiness Systems", IFIP I3E, Springer ISBN:0-387-28753-1, pp. 389-403, 2005.

[3] M. Oya: "Autonomous Web Services Based on Dynamic Harmonization", IFIP I3E, Springer, ISBN:978-0-387-8590-2, pp. 139-150, September, 2008.

[4] K. Sycara, M. Paolucci, A. Ankolekar, N. Srinivasan, "Automatic Discovery, Interaction, and Composition of Semantic Web Services", Journal of Web Semantics, vol. 1, Issue 1, pp. 27-46, 2003.

[5] B. Martino1, "An Ontology Matching Approach to Semantic Web Services Discovery", Lecture Notes in Computer Science, Springer, pp. 550-558, 2006.

[6] Y. Chabeb, S. Tata,D. Belaid, "Toward an integrated ontology for web services", Proc. of ICIW 2009 , art. no. 5072561, pp. 462-467, 2009.

[7] M. Oya, M. Ito, S. Tsukamoto, R. Takagi, T. Kimura, "AWS (Autonomous Web Services and its Middleware", IPSJ 71st Conference Proceedings, pp. 1-503-504, 2009. (in Japanese)

[8] 8. R. Chinnici et al, Web Services Description Language (WSDL) Version 2.0, W3C Recommendation, 2007.

[9] OASIS, "ebXML Messaging Services Version 3.0", OASIS Standard, 2007.

[10] OASIS, "Web Services Business Process Execution Language Version 2.0", OASIS Standard, 2007.

[11]. J. Miller et al, MDA Guide Version 1.0.1, OMG doc. omg/2003-06-01, 2003.