Autonomous Web Services Based on Dynamic Model Harmonization

Makoto Oya

Information Science, Shonan Institute of Technology,

1-1-25, Tsujido Nishi-Kaigan, Fujisawa, 251-8511, Japan

Abstract. Web Services has become the infrastructure to connect business applications over the Internet. Under the current Web Services, however, stakeholder systems must follow the predefined rules for a particular business service including those about business protocols to send/receive messages and about system operation. Only the systems built strictly to follow the predefined rules can participate in the concerning service. This is insufficient when considering future eBusiness systems. More flexible mechanism is desired where freely built and autonomously running systems can exchange business messages without pre-agreed strict rules. We call it Autonomous Web Services (AWS) and proposed the framework called Dynamic Model Harmonization (DMH) with its algorithm, which dynamically adjusts different business process models between systems [1]. In this paper, we propose middleware technology that realizes AWS based on the DMH. The proposal includes the mechanism and specification to drive an application by a dynamically harmonized business process model, as well as the DMH algorithm extended in line with the proposed mechanism. In addition, the way to control very long transactions often appear in AWS environment is proposed, and messaging infrastructure for AWS is discussed based on the prototype implementation.

1 Introduction

As Web Services became enough matured and widely used, adjustment of business process among eBusiness/eCommerce systems has become a next important issue.

Please use the following format when citing this chapter:

Oya, M., 2008, in IFIP International Federation for Information Processing, Volume 286, Towards Sustainable Society on Ubiquitous Networks, eds. Oya, M., Uda, R., Yasunobu, C., (Boston: Springer), pp. 139 - 150.

Web Services is the technology to exchange data between applications in different systems over the Internet using HTTP/XML. There are two types of message exchange protocol, the RPC protocol that directly uses HTTP request-response protocol and the messaging protocol that asynchronously exchanges messages using additional protocol layer over HTTP. RPC is enough for simple information delivery. For full-scale commercial transactions, the messaging protocol is suitable in general because actual work often occurs after receiving HTTP-request and immediate HTTP-response with the result data is difficult to make. These protocols have been standardized and well established, such as SOAP/Framework [3] for the basic protocol, SOAP/RPC [4] for RPC, ebXML Messaging [5] and WS-Reliability [6] for the messaging protocol. WS-Reliability supports the reliable message delivery features, the duplicate elimination and the guaranteed message ordering, which are indispensable for the commercial message exchange.

Under the current Web Services technology, it is assumed that systems share an agreed model of business process to execute consistent exchange of business messages. The stakeholder systems must strictly follow the predefined rules for a particular service. The rules precisely define the business process including message formats and possible sequences of message delivery and may direct a specific operation rule of each system. Defining such common rules and forcing systems to follow them is possible in a particular enterprise or enterprise group, a particular domain group, or a community sharing a well-defined purpose. As a matter of fact, many business transaction systems are working under the current Web Services using business process description language such as BPMN and BPEL [7].

When considering eBusiness/eCommerce systems scattered over the world in the Internet, however, predefining or standardizing precise business process rules for various business transactions and forcing systems to be developed strictly according to them is not realistic and does not agree with the vision of the Internet. Improved flexible Web Services technology is needed where freely built and autonomously running systems can exchange business messages without predefined rules in order to pursue free and flexible network society for the future. We call this technology as *Autonomous Web Services* (*AWS*). Unknown systems, when they encounter in the Internet, adjust their business process models using their best effort and execute business transaction. They are not necessary to be developed in the way to follow specific rules or to subordinate to specific management servers. Loose common *environment* that is independent of particular services may exist, however, systems that are freely developed and autonomously running under such environment can execute business transaction on demand. This is the ultimate goal of AWS.

Two big challenges are addressed to realize AWS. The major challenge is how to dynamically adjust business process models and drive the application program as the adjusted model instructs. The other is, derived from systems' autonomy, how to control and maintain very long transactions that often occur between systems not sharing a unified operation rule. In order to solve the former challenge, we provided the method that dynamically generates business protocols [2], and then proposed the generalized framework called *Dynamic Model Harmonization* (*DMH*) and showed the detailed algorithm [1]. In DMH, systems expose their business process model descriptions including operation signatures and behavior. Two systems exchange their models when they encounter in the Internet. Each system harmonizes (i.e., dynamically

modifies) its model adjusting to the partner's model, and then starts exchanging a series of business messages.

This paper proposes technology to enable middleware that realizes AWS based on DMH. We call this middleware as *AWS middleware*. The DMH algorithm proposed in [1] is extended to apply to middleware implementation. Mechanism to control applications according to the harmonized model is proposed. An application is broken into segments corresponding to each send/receive operation and the segments are driven by a harmonized model generated by DMH algorithm. A solution to the second challenge is also proposed, introducing two concepts, VL session and VL process.

The structure of this paper is: Section 2 explains the principle of DMH and proposes the extended DMH algorithm applicable to middleware technology. Section 3 mentions the overview and major problems of AWS middleware. Section 4 explains the proposed way of controlling application under DMH and provides details of model harmonization, control mechanism and programming model using an example. Section 5 mentions about VL transaction and messaging infrastructure, and discusses on the prototype implementation. Section 6 gives a short conclusion.

2 Dynamic Model Harmonization

The core of Autonomous Web Services (AWS) is Dynamic Model Harmonization (DMH). This section defines the basic principle of DMH, and proposes the extended DMH algorithm applicable to middleware control.

2.1 Principle of DMH

In the framework of DMH, each system exposes its external specification as a *model* M, which is represented by $M = (O, B)$, where:

- O is a set of *operations op*, where $op = (pattern, format)$.
- *pattern* indicates whether the operation is output (send) or input (receive).
- *format* specifies a type of output or input message.
- *behavior* B is, in general, represented as a finite state machine, namely
  $B = (S, F, \Phi)$, where
  S ($\subset O$) is a set of starting operations,
  F ($\subset O$) is a set of final operations, and
  Φ is a transition function.

DMH consists of the following four steps (**Fig.1** outlines these steps):

1. Two systems Z_1 and Z_2 expose each model $M_1 = (O_1, B_1)$ and $M_2 = (O_2, B_2)$.
2. Z_1 and Z_2 exchange the models when they encounter.
3. In each system, matching between the operations is examined and the model is modified (harmonized) to co-operate with the model of the opposite system. This process is called *DMH algorithm*.

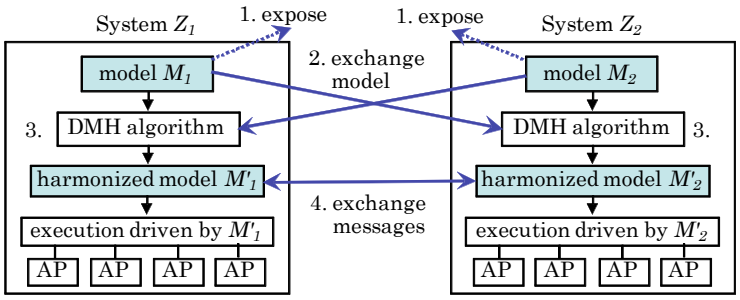

Fig. 1. Outline of Dynamic Model Harmonization

4. If resulting harmonized models M'_1 and M'_2 are not empty and operation mapping is determined, business transaction, i.e. a series of message exchanges, starts according to the harmonized model. Otherwise, business transaction is abandoned.

It is assumed that a type matching function t_match() is provided from the environment (see Section 1). When f is an output format and g is an input format, t_match(f, g) examines whether instances of f matches to g. As shown in [1], t_match() is appropriate to be defined as the following three value function:

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).

Various kinds of t_match() exist. Simplest one is when type names in the same name space are specified as *format* and they are stored in a specific portion of message texts. Value of t_match() is easily determined by examining type names equality. However, in more complex cases such as type is given by a complex XML schema, determination of t_match() may be limited in a certain range. DMH algorithm determines possible operation mapping by examining t_match() ≠ false taking this into consideration. Application of Web ontology is another potential approach for t_match() implementation. See more discussion in [1] and [2].

Note that the first element O in M corresponds to "interface" in WSDL2.0 [8]. The *pattern* corresponds to the MEP (message exchange pattern) though this paper restricts its value either 'input' or 'output' to simplify the discussion.

2.2 DMH Algorithm

In [1], a behavior was represented by a regular expression and a precise DMH algorithm was shown and verified. Here, a behavior is represented as a nondeterministic automaton, and an extended DMH algorithm in line with middleware control is proposed bellow. When a behavior B is restricted as a nondeterministic automaton, a transition function Φ is given by $\Phi(o)$ = (a subset of O) where $o \in O$. The proposed DMH algorithm is outlined as follows.

- First, for $M_1 = (O_1, B_1)$, B1 = (S_1, F_1, Φ_1), $M_2 = (O_2, B_2)$ and $B_2 = (S_2, F_2, \Phi_2)$, a product model M is built, such that $M = (O, B)$ where $O = O_1 \times O_2$, $B = (S, F, \Phi)$, $S = S_1 \times S_2$, $F = F_1 \times F_2$, $\Phi(o_1, o_2) = \Phi_1(o_1) \times \Phi_2(o_2)$.
- An operation matching function o_match(p, q) is defined as follows (where p, $q \in$ O, fp and fq are formats of p and q respectively):
  - o_match(p, q) = false if both p and q are input or output,
  - o_match(p, q) = t_match(fp, fq) if p is output and q is input, and
  - o_match(p, q) = t_match(fq, fp) if p is input and q is output.

- Next, M is modified in the following three steps:

1. remove from O all pairs (o_1, o_2) satisfying o_match(o_1, o_2) = false.
2. reduce S, F and values of Φ to be subsets of O.
3. remove unnecessary nodes and paths from B (= (S, F, Φ)). Namely, remove from O all pairs (o_1, o_2) that are not reachable from S and all pairs (o_1, o_2) that do not reach to F. They are also removed from the domain and values of Φ.

- The resulted M (denoted M') is called a *harmonized co-operation model* (or simply a *harmonized model* when the context is clear). Z_1 part of M' (denoted M'_1) and Z_2 part of M' (denoted M'_2) are called harmonized models of Z_1 and Z_2.

At some status of the harmonized co-operation model and for some output operation of the system, if a destination input operation of the opposite system is not uniquely determined, we say "operation mapping is not determined". Precisely, it is the case when, for some $(o_1, o_2) \in$ O, $\Phi(o_1, o_2)$ includes both (a, b_1) and (a, b_2) or both (b_1, a) and (b_2, a), where a is an output operation, and b_1 and b_2 are distinct input operations. Otherwise, we say "operation mapping is determined". If the harmonized co-operation model is not empty and operation mapping is determined, then it decides Z_1 and Z_2 are co-operative and business transaction is started. After starting, a series of message delivery is controlled according to the harmonized co-operation model.

3 AWS Middleware

This section proposes a middleware, called *AWS middleware*, realizing AWS based on the DMH framework and algorithm mentioned in the previous section.

3.1 Overview

AWS middleware constitutes a store-and-forward type asynchronous messaging system. Fig.2 shows the overview when systems Z_1 and Z_2 exchange messages over AWS middleware. Note that the internal structure of Z_2 is same as Z_1 but simplified in the figure. End points of messaging are called UA (user agent). UA has a unique name and plays as a primary entity of sending and receiving messages. A message

sent by UA through AP (application program) is stored in the output queue (outQ), then, asynchronously delivered to the input queue (inQ) in the opposite system. The message in inQ is asynchronously taken out and passed to UA through AP. Reliable messaging, exactly once semantics and message ordering guarantee, is supported since AWS middleware handles business transaction.

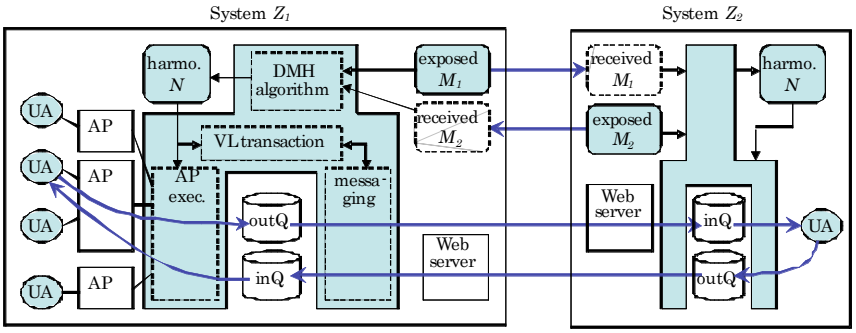

Fig. 2. AWS middleware overview

Messaging protocol of AWS middleware is built over HTTP and SOAP/Framework. Like ebXML Messaging [5], one one-way message delivery is processed by one HTTP request-response. A message taken out from outQ is sent to the Web server of the recipient system by HTTP POST request. If successfully stored in inQ, HTTP response is returned to the original system and the message is removed from outQ. If not successful, retry occurs. Asynchronous reliable messaging is implemented in this way. The protocol is natural extension of the current Web Services protocols and any additional existing Web Services technology is applicable. Security mechanism such as PKI, for instance, is applicable exploiting UA has a unique name.

3.2 Problems

Major problems to be solved are following three:

1. DMH driven application execution
   In system Z_1, AWS middleware generates a harmonized co-operation model N from its own model (M_1 in Fig.2) and a received model (M_2) from Z_2. Then it needs to control AP execution flow according to the contents of N. It is difficult if send/receive operations are embedded in AP. In addition, as a premise of such DMH driven execution control, it needs to separate operation mapping from AP, establishing independence of a model and program codes.
2. Control of very long transaction
   Comparing to short transaction processing such as online banking system, business transaction in the Internet does not require strict ACID but each transaction often continues during long period such as from price inquiry to delivery of goods. In AWS, each system has its own policy of system operation and administration. Life of transaction tends to be longer because of long waiting time to receive an expecting message. Even a message sending operation may be blocked because of a long

period of retry, several hours, several days, or more. Thus, a business transaction in AWS is much longer than that in the current well-managed Web Services business transaction system. AWS middleware needs to control such a very long transaction (VL transaction).

3. Messaging infrastructure
   Messaging infrastructure itself does not have big issues since messaging is enough mature technology, except that retry algorithm and message queue implementation need special consideration in AWS middleware. Sending operation may accompany very long retry cycle as mentioned in 2 but AWS does not assume systems to share a single messaging policy. AWS middleware needs to decide times and interval of retry based on runtime negotiation and dynamic control. In addition, reliable and dependable implementation of inQ and outQ is required because message queues keep large number of messages during long period, even beyond system shutdown time.

We focus on the problem 1 in this paper and propose our solution in the next section. The problems 2 and 3 are discussed in Section 5.

4 DMH Driven Application Execution

Differently from usual client-server communication, under the framework of DMH, output operations are exposed as well as input operations. An application sends a message to an output operation of its own system instead of an input operation of the opposite system. Operation mapping is automatically determined by AWS middleware using DMH algorithm as mentioned in2.2. Application does not need to know operations of opposite systems and independence and autonomy of a system are improved. Moreover, to separate program codes and a model, AWS middleware supports a *program configuration file* to specify methods and message format handling classes corresponding to each operation in the model.

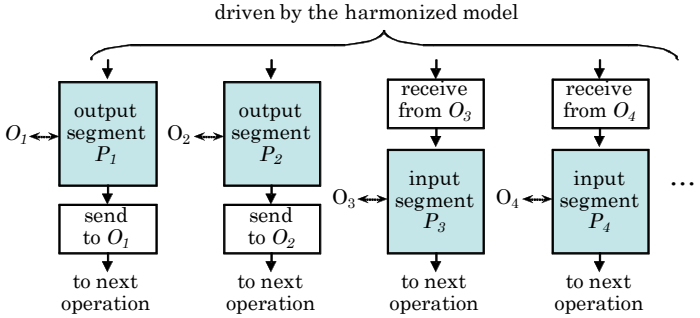

Fig. 3. Program execution control using harmonized model

AP is defined as a set of segments (called *message I/O segments*) to realize application execution control driven by the harmonized model. Each message I/O segment corresponds to an output or input operation. According to an execution status of the harmonized co-operation model, AWS middleware invokes an appropriate message

I/O segment like an event driven loop. A segment forms a method. Input and output are not instructed directly from program codes. The message is sent to the corresponding output operation right after the method returns in case of an output segment, and the received message is passed to the method when it is invoked in case of an input segment. Fig.3 illustrates the proposed programming model. The details are mentioned in the following subsections using an example.

4.1 Model Description and DMH

Suppose a system Z_1 exposes a model M_1 shown in Fig.4. Here, a model is described by YAML [9] and a behavior is denoted by an activity diagram for easy understanding. Note that YAML is convertible to XML and an activity diagram can be transformed into XML using such as UML/XMI [10]. Name spaces are omitted for simplicity. Type names of format are assumed in the same name space and t_match() is calculated simply using names identity (see 2.1).

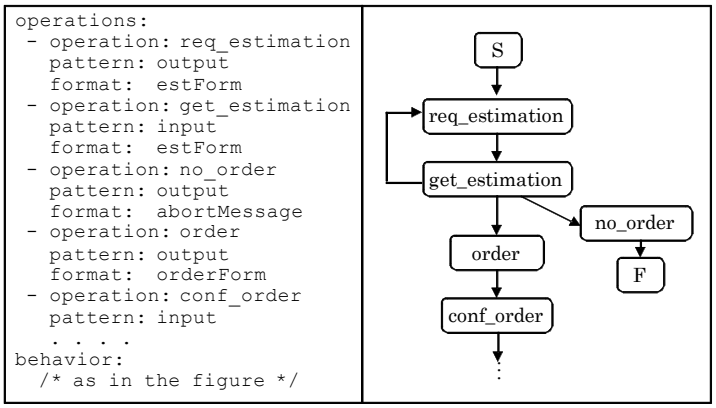

Fig. 4. Exposed model M_1

M_1 has four operations in the scope of Fig.4. It sends an estimation condition (req_estimation) and receives the answer to it (get_estimation), orders (order) if the answer is acceptable, and sends another estimation condition (req_estimation) or aborts the transaction (no_order) if the received answer seems not appropriate.

Suppose Z_1 encounters another system Z_2 exposing a model M_2 in Fig.5. M_2 receives estimation condition (Estimate) and returns the answer (Answer). It can return several different answers to one estimation request but cannot receive estimation request more than one time. DMH algorithm generates a harmonized co-operation model N from M_1 and M_2 as shown in Fig. 6. Then AWS middleware drives AP using the model N.

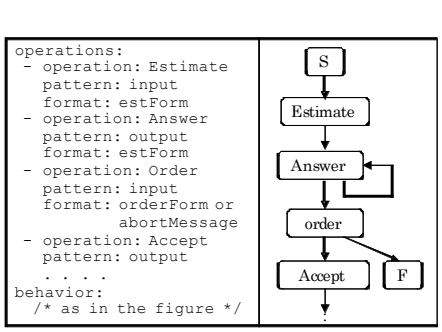

Fig. 5. Fig.5 Exposed model M_2

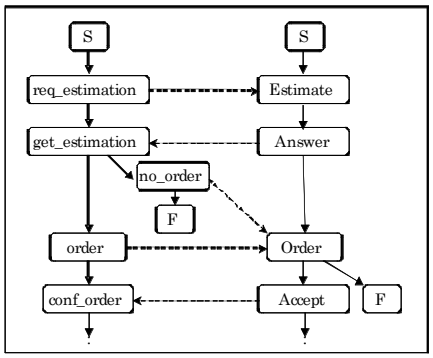

Fig.6. Harmonized co-operation model *N*

4.2 Program Configuration File

AP in Z1 consists of a set of methods implementing message I/O segments and *message coding classes* to encode output messages and decode input messages. Each method receives by its parameter a reference to an instance of the corresponding message coding class. AP configuration is specified in a program configuration file. An example is shown in Fig. 7.

```
programCofiguration:
 - operation: req_estimation
   method:    sendEst
   codingClass: EstForm
 - operation: get_estimation
   method:    recieveEst
   codingClass: EstResult
 - operation: order
   method:    order
   codingClass: OrderForm
 . . . .
```

Fig. 7. Program configuration file

In this example, it specifies sendEst() is a method corresponding to the output operation req_estimation and stores output message contents in a class EstForm object. receiveEst() is a method corresponding the input operation get_estimation and receives the contents of received message by a class EstResult object. EstForm and EstResult need to be created as classes to access estForm type format specified in the M_1 in Fig. 4. Besides operations appear in a model, special methods invoked at starting and ending time of transaction may be prepared.

4.3 Application Codes

Fig.8 is an example of application code that implements the methods specified in the program configuration file in Fig. 7. As explained using Fig. 3, message I/O instructions are not placed inside of the methods. Since sendEst() and order() corresponds to output operations, when they return, the objects passed by their parameters are encoded into message texts by the message coding classes EstForm and OrderForm, and routed to req_estimation and order. receiveEst() is invoked when get_estimation has received a message text. The message text is decoded by the message coding class EstResult into an object and passed to receiveEst() by its parameter. An operation next to the operation get_estimation is non-deterministic in M_1. Possible operations are order, no_order and req_estimation. As seen in the example (Fig. 6), the model is harmonized, a part of possible operations may become not applicable. Two methods, setNextOperation() and getNextOperation, are prepared to handle this situation. In this example of receiveEst(), possible next operations can be examined by getNextOperation() and an intended next operation can be specified by setNextOperation('order') or setNextOperation('no_order') in receiveEst() before returning.

```
class SampApp extends AWSApp {
  public sendEst(EstForm: ef)
  {
    /* set estimation conditions in ef */
    return;
  }
  public receiveEst(EstResult: er)
  {
    /* process estimation results in er */
    getNextOperation(...);
    setNextOperation(...);
    return;
  }
  public order(OrderForm: or)
  {
    set order contents in or */
    return;
  }
  . . . .
}
```

Fig. 8. Application code

5 VL Transaction and Messaging

(1) VL Transaction

We introduce two concepts, *VL session* (very long session) and *VL process* (very long process) to control VL transaction. A VL session is a non-volatile session. It starts from *S* in the model and kept until *F*. After the DMH algorithm decided co-operation

is possible, a common session identifier (VL session id) is generated and a transaction starts from starting operations. Hereafter, VL session is kept until the transaction reaches final operations even one or both system stops and restarts on the way. Model exchange at VL session creation is performed by messaging communication recursively using AWS middleware. AWS middleware has a special well-known VL session for this purpose. All communications between AWS middleware are done through this special VL session.

VL session enters into waiting status in two reasons: the opposite system does not yet send a message though input operation is invoked, or a message is not delivered to the queue of the opposite system though the message is already sent from output operation. Waiting period may be very long, a few hours, a few days, a few weeks or more as discussed in 3.2. It is not suitable from system resource limitation to keep VL sessions active during long term of waiting. AWS middleware automatically archives VL sessions to external storage during such period. The control mechanism for this feature is VL process. VL processes behave as ordinal processes (or tasks) when they are active. One VL process has one or more VL sessions. When a VL session falls into long waiting status during message sending or receiving, the VL process is archived to external storage with the contexts of its VL sessions. When waiting status is released, an actual process is assigned and the VL process is reactivated. VL processes are also archived at system shutdown time, realizing non-volatile VL sessions. Archive of VL process occurs only during sending message or receiving message. It does not occur when AP is running, which avoids to archive large data including AP's local resources.

(2) Messaging Infrastructure

We developed prototype implementation supporting messaging infrastructure with VL session feature [11]. Solvability of the problems mentioned in 3 of 3.2 was examined through this prototyping. VL sessions were implemented as objects. Blocking and non-blocking send() and receive() were supported as low level APIs. When a VL session object is created, its ID is generated by negotiation between systems and the VL session is maintained hereinafter. Protocol was built over SOAP/Framework. Receiving process works as CGI behind a Web server. A “retry reason level” is included HTTP error responses and a retry interval is dynamically calculated using the levels. In order to make queues (inQ and outQ) durable for long time, they are implemented over DBMS (PostgreSQL). Queue access synchronization was simplified and became safer. Through this prototyping, it was shown the existing messaging technology is well applicable for implementation of AWS messaging infrastructure with some considerations mentioned above.

6 Conclusion

In this paper, we proposed the middleware technology to realize Autonomous Web Services as well as the extended DMH algorithm. Application is segmented corresponding to input and output operations and driven by the middleware. Application

execution flow dynamically changes in line with a harmonized model. It realizes systems having different business process models dynamically execute business transaction without predefined strict rules. We also showed solutions to the secondary problem on very long transactions introducing the concepts of VL session and VL process. VL sessions are archived when input/output messaging falls into a long waiting status, effectively providing non-volatile long session. Finally, implementation method of messaging infrastructure for AWS is examined through prototyping. Materialization of the ultimate goal of Autonomous Web Services is a long running theme. Future issues include type matching method, harmonization of more than two systems and VL transaction management.

Acknowledgments. This work was supported by KAKENHI (19500095).

References

1. M. Oya and M. Ito, Dynamic Model Harmonization between Unknown eBusiness Systems, IFIP I3E, Challenges of Expanding Internet: E-Commerce, E-Business, and E-Government, Springer, pp. 389-403, 2005.
2. 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.
3. M. Gudgin et al, SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), W3C Recommendation, 2007.
4. M. Gudgin et al, SOAP Version 1.2 Part 2: Adjuncts (Second Edition), W3C Recommendation, 2007.
5. OASIS, ebXML Messaging Services Ver. 3.0: Part 1. Core Features, OASIS Standard, 2007.
6. OASIS, WS-Reliability 1.1, OASIS Standard, 2004.
7. T. Andrews et al, Business Process Execution Language for Web Services, 2003.
8. R. Chinnici et al, Web Services Description Language (WSDL) Version 2.0, W3C Recommendation, 2007.
9. O. Ben-Kiki et al, YAML Ain't Markup Language (YAML™) Version 1.1, yaml.org, 2005.
10. OMG, MOF 2.0/XMI Mapping, Version 2.1.1, OMG doc. formal/2007-12-01, 2007
11. M. Ito, M. Sawaguchi, H. Matsubara and M. Oya, Asynchronous P2P Communication Middleware, IPSJ 70th Conference, pp.I-565-568, 2008. (in Japanese)
12. J. Miller et al, MDA Guide Version 1.0.1, OMG doc. omg/2003-06-01, 2003.