情報処理学会論文誌 Vol.54 No.2 982–991 (Feb. 2013)

自律型Webサービス：原理と実装

大谷 真[1,a)]

受付日 2012年5月9日, 採録日 2012年11月2日

概要：インターネットでの電子商取引などのビジネスメッセージ交換について考えるものとする．従来のWebサービスでは，関連サイトにまたがる全体的なビジネスプロセスモデル（BPM）が事前定義されていることが前提である．各サイトはそれにそって動作するように作られていなければならない．自律型Webサービス（AWS；Autonomous Web Services）は，全体的なBPMは不要で個々のサイトが独立にBPMを持つものとし，各サイトのBPMを動的に協調させることで，自由に作られたサイト間でも柔軟にビジネスプロセスが実行できることを目的としている．2004年にAWSの核である動的モデル協調の初期的アルゴリズムを提案して以来，我々は，AWSの基本原理を確立しアルゴリズムを改良するとともに，ソフトウェア基盤であるAWSミドルウェアの制御方式とそのアプリケーションインタフェースを提案した．また，BPMの外部表現と内部表現，ビジネスプロセスインスタンスに対するスレッド制御方法などの実装方式を考案してきた．さらにそれらに基づき，AWSミドルウェアを実装しその評価を行った．この結果，AWSの基本原理とアルゴリズムの妥当性だけでなく，ミドルウェア制御方式や実装方式の妥当性，有効性，実装可能性，および実用性が検証された．本論文の目的は，AWSの原理，実装，および成果を体系的に述べることである．

キーワード：Webサービス，自律システム，電子商取引，ビジネスプロセスモデル，ミドルウェア

Autonomous Web Services: Principle and Implementation

Makoto Oya[1,a)]

Received: May 9, 2012, Accepted: November 2, 2012

Abstract: Let us consider business message exchanges over the Internet like in electric commerce. The current Web Services requires a predefined entire BPM (Business Process Model) across related sites. Each site must be built in accordance with the entire BPM. The AWS (Autonomous Web Services) does not requires such a predefined entire BPM and permits each site has its own BPM. It aims to enable a flexible execution of business processes between freely built sites by dynamically harmonizing BPMs. Since having proposed in 2004 an initial algorithm for the dynamic model harmonization, we established the basic principle of the AWS with improvement of the algorithm then proposed the new control mechanism and application interface for the AWS middleware. We also contrived implementation methods such as an external representation of BPM and the thread control mechanism for business process instances. Based on them, we succeeded in implementation of the AWS middleware and completed its evaluation. These show not only the properness of the AWS's basic principle and algorithm, but also the appropriateness, effectiveness, implementation possibility and practicality of the proposed mechanism of the AWS middleware. This paper systematically states the AWS's principle, its implementation, and the results.

Keywords: Web services, autonomous system, electronic commerce, business process model, middleware

1. はじめに

SOAを含むWebサービスは，2000年にBoxらによってSOAPプロトコルが提案 [1] されて以来急速な進歩をと

[1] 湘南工科大学
Shonan Institute of Technology, Fujisawa, Kanagawa 251–8511, Japan
[a)] oya@info.shonan-it.ac.jp

げ，今やインターネットでのビジネスメッセージ交換の基盤技術になった．当初は同期型の RPC に限定されていた SOAP は 2007 年の SOAP1.2 [2] において基本プロトコル SOAP/Framework と RPC に分離され，同時に SOAP/Framework の上位に非同期のメッセージングプロトコルも標準化された [3]．Web サービスはインタフェースプログラミング [4] を基礎としており，実行時プロトコルだけでなくプロトコル記述言語をも定めているところに特徴がある．SOAP の記述言語 WSDL は 2001 年に提案され，2007 年の WSDL2.0 [5] においてメッセージパターンなどの基本機能を含む仕様として確立された．もう 1 つの重要な記述技術はビジネスプロセスの流れを記述するための言語であり，BPEL [6] や BPMN [7] が規定されている．これらの技術を中心にいわゆる SOA [8] が実現された．2000 年半ばから，市場に“アプリケーションサーバ”や“ソフトウェアバス”と呼ばれる SOA ミドルウェア製品が登場し，現在ではインターネットでの情報システムの連携基盤として広く普及している．

しかし，従来の Web サービスには，複数のサイトにまたがる全体的なビジネスプロセスの流れ（ビジネスプロセスモデル；BPM）を前もって取り決めておかなければならないという問題がある．すなわち，電子商取引に代表されるビジネスメッセージ交換を考えた場合，関連するシステムを横断する一連のメッセージ交換の手順があらかじめ詳細に定義されていることが必須前提である（**図 1** (a)）．そのうえで，個々のシステムは全体的 BPM に合わせて自システムの役割部分を正確に実行するように作られていなければならない．これを無視して自由に作られたシステムはその取引には参加できない．大企業や業界団体または政府が主導して取引手順を決められる場合，あるいは定型的な商取引の場合などは別として，一般には取引手順の詳細な標準化は困難なことが多く，現実的には適用できないことが多い．これに対して，我々が研究を進めてきた自律型 Web サービス（AWS；Autonomous Web Services）は，自由に作られたシステム間でも，柔軟にビジネスプロセスが実行できることを目標としている．AWS では，システムを横断した全体的 BPM は不要で，個々のシステムはそれぞれ独自に BPM を持つことができる．そのうえで，インターネット内でシステムが遭遇した時点で，個々の BPM を動的に協調させ（整合化させ），協調済 BPM に従って一連のビジネスメッセージ交換を実行する（図 1 (b)）．

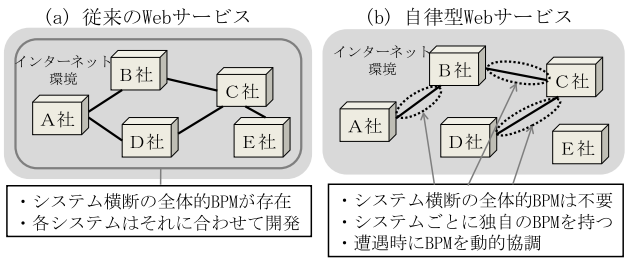


図 1 従来の Web サービスと自律型 Web サービス（AWS）
Fig. 1 Conventional Web Services and Autonomous Web Services (AWS).

AWS の中核技術は動的モデル協調（DMH；Dynamic Model Harmonization）である．DMH の基本原理とアルゴリズム（DMH アルゴリズム）は MDA [9] を背景に，文献 [10], [11], [12] で提案され，文献 [13], [14] で体系化された．AWS を実現するための基盤ソフトウェアが AWS ミドルウェアである．AWS ミドルウェアは DMH 実装や通信下位層などの AWS 実装詳細をアプリケーションから隠蔽し，抽象度の高い API でアプリケーションを開発可能とすることを目的としている．2008 年に AWS ミドルウェアの基本方式が提案され [14]，その後，プロトタイプの開発が行われ [15], [16], [17], [18], [19], [20], [21]，実装方法が確立された [22]．これに基づいて 2011 年に 3 層構造を持つソフトウェアとして完成した [23], [24]．また実運用を想定した実験をとおして AWS の有効性も検証された [25], [26]．

AWS 研究成果の一部は学術雑誌論文や海外学術会議論文として公開されているが，かなりの部分は参考文献のとおり多数の学会大会発表論文で示されている．本論文の目的はこれまでの成果を最新の結果とともに体系だって説明することである．本論文の構成は次のとおりである．2 章では DMH について，基本原理，BPM の形式定義，および DMH アルゴリズムについて述べる．3 章では AWS ミドルウェアの目的と概要，BPM の外部表現，AWS 実現のために考案された制御方式であるモデル駆動型 AP 実行，そしてアプリケーションプログラムインタフェースを記す．4 章では AWS ミドルウェアを構成する 3 つのソフトウェア層に分けて実装方式詳細を述べる．5 章では実運用を想定したテストを含めた AWS の評価結果を述べる．6 章に結論と今後の課題を簡潔にまとめる．

2. 動的モデル協調（DMH）

2.1 ビジネスプロセスモデル（BPM）

DMH では，インターネット内のシステムは，それぞれ独自に自システムのビジネスプロセスの流れを記述しておく．たとえば「見積りを依頼し，見積り結果を受け取る．その後，発注する，または，取引を中止する」といった記述である．これをビジネスプロセスモデル（BPM）と呼ぶ．BPM は外部に公開（外部からアクセス可能に）しておく．従来の Web サービスとは異なり，システムを横断した全体的な BPM の記述は不要である．なおここでいうビジネスプロセスとは，第 1 義的には電子商取引を想定しているが，必ずしもそれに限らず，個人・組織のシステム間での何らかの意味を持つ一連のメッセージ交換ならどんなものでもよいことに留意願いたい．

BPM M は，$M = (O, B)$ と形式定義される．O はそのシ

ステムが持っているオペレーション $op = (pattern, format)$ の集合である．*pattern* はそのオペレーションが受信（'input'）であるか送信（'output'）であるかを示す．*format* は受け渡されるメッセージ形式であり，ここではその名前を指定する[*1]．B は振舞い（オペレーション実行の流れ）であり，一般には O をアルファベット集合とする有限状態機械として定義される．すなわち，$B = (S, \lambda, F, \Phi)$．ここに，$S$ は状態の集合，λ（$\in S$）は開始状態，F（$\subset S$）は終了状態の集合，Φ は遷移関数である．ただしこの論文では，B は非決定性オートマトンの範囲に限定する[*2]．つまり，Φ を $S \times O \to S$ に制約する．

研究上必要な範囲に単純化しているが，BPM は原理的に WSDL [5] の自然な拡張形になるように工夫されている．O は WSDL の interface または portType に相当し，*pattern* は MEP（message exchange pattern）に対応している．XML による表現も可能である．

2.2 DMH の基本原理

DMH は次の 4 ステップで構成される．**図 2** にその概観を示す．

1) システム Z_1 と Z_2 は，それぞれの BPM $M_1 = (O_1, B_1)$ と $M_2 = (O_2, B_2)$ を公開しておく．
2) Z_1 と Z_2 が遭遇した（ビジネスプロセスの実行を希望した）時点で，動的に，Z_1 と Z_2 は互いの BPM を交換する．
3) それぞれのシステムは，相手システムから受け取った BPM と自システムの BPM を突き合わせて，両者が協調動作するように，自システムの BPM を変形する（変形のためのアルゴリズムを DMH アルゴリズムと呼び，変形した BPM のことを協調済 BPM と呼ぶ）．
4) 協調済 BPM が空でなければ（すなわち変形が成功したら），協調済 BPM に従ってアプリケーション（AP）を駆動し，ビジネスプロセスに必要な一連のメッセージ交換を行う．

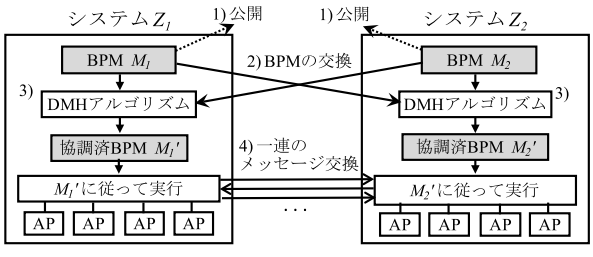


図 2 動的モデル協調（DMH）の基本原理
Fig. 2 Principle of Dynamic Model Harmonization (DMH).

[*1] 商取引を例にとると format は見積書様式や注文書様式などの送受信メッセージに対する様式定義に相当する．format にはその名前（いわば様式名称）を指定する．本論文では，format 名に対してその形式（または様式）の意味と表現形式が外部環境において何らかの方法で定義されているものと仮定している．
[*2] アルゴリズムの決定性確保のためにこれまでの研究では B を非決定性オートマトンの範囲に制約している．

2.3 オペレーションマッチング

DMH アルゴリズムの中では，振舞い B の変形に先立って互いの BPM 間でオペレーションマッチング（どのオペレーションの出力をどのオペレーションで入力できるか）を決定する．このために，format f と g に対して以下の 3 値関数 tMatch(f, g) の実装が外部環境の一部として与えられていることを仮定する．

$$\mathrm{tMatch}(f,g) = \begin{cases} \text{true（}f\text{ のすべてのインスタンスが } g\text{ を充足すると判定できるとき）} \\ \text{false（}f\text{ のあるインスタンスが } g\text{ を充足しないと判定できるとき）} \\ \text{undefined（true とも false とも判定できないとき）} \end{cases}$$

これを用いてオペレーションマッチング関数 o_match(o, p) を以下のとおり定義する（ただし，o と p はオペレーション，f_o と f_p はそれぞれの format とする）．

$$\mathrm{o\_match}(o,p) = \begin{cases} \mathrm{tMatch}(f_o, f_p)\text{（}o\text{ が output で } p\text{ が input のとき）} \\ \mathrm{tMatch}(f_p, f_o)\text{（}o\text{ が input で } p\text{ が output のとき）} \\ \text{false}\quad (o\text{ と }p\text{ がともに output または input のとき）} \end{cases}$$

オペレーションマッチングの判定には悲観的判定（o_match() = true のときのみマッチしたと見なす）と楽観的判定（o_match() ≠ false のときマッチしたと見なす）の 2 つがある [10]．可能性があるオペレーションの組合せをメッセージ送受信開始前に除外してしまうことをできるだけ避けるため，本論文ではこれまでの AWS 研究と同様に楽観的判定を適用している．ただし本論文で述べていることは悲観的判定を適用したときにも同様に成立することに留意いただきたい．

tMatch() は様々な方法で実現できる．最も単純な方法の 1 つは，format を名前空間と名前の組で表現しておき，名前空間が一致して $f = g$ なら true，$f \neq g$ なら false，名前空間が一致しないときは undefined を返すものである（これを“自明な”tMatch と呼ぶ）．DMH 検証のシミュレーションや AWS ミドルウェアのテストでは，この自明な tMatch() がしばしば使用されている（文献 [10], [14], [26] など）．このことは簡単な tMatch() 実装であっても DMH は十分な効果をもたらすことを間接的に示すものである．他の有力な実現方法はセマンティク Web やオントロジを応用することである [11], [27]．また WSDL マッチングに関する研究成果（文献 [28], [29] など）を応用することも考えられる．なお本論文では，tMatch() の存在を前提に

BPM 協調を論議し，tMatch() の実装方法は範囲外としている．

2.4 DMH アルゴリズム

当初提案 [10] や改良 [12], [15], [22] を洗練させた DMH アルゴリズムを以下に示す．

自システム Z_1 の BPM を $M_1 = (O_1, B_1)$ とし，相手システム Z_2 から受け取った BPM を $M_2 = (O_2, B_2)$ とする．$B_1 = (S_1, \lambda_1, F_1, \Phi_1)$，$B_2 = (S_2, \lambda_2, F_2, \Phi_2)$ とする．システム Z_1 内での DMH アルゴリズムは以下のとおりである．

まず，M_1 と M_2 を連携させた BPM $N = (P, (T, \mu, G, \Gamma))$ を次の手順で作成する．

- $P = \{(o_1, o_2) \,|\, o_1 \in O_1 \,\&\, o_2 \in O_2 \,\&\, \mathrm{o\_match}(o_1, o_2) \neq \mathrm{false}\}$
- $T = S_1 \times S_2$, $\mu = (\lambda_1, \lambda_2)$, $G = F_1 \times F_2$
- $\Gamma((s_1, s_2), (o_1, o_2)) = \Phi_1(s_1, o_1) \times \Phi_2(s_2, o_2)$ （ただし，$(s_1, s_2) \in T$ & $(o_1, o_2) \in P$）
- μ から到達できないまたは G のどの要素にも到達できない状態 τ を，T からすべて取り除く．さらに，τ からの遷移および τ への遷移を，Γ からすべて取り除く．

次に，システム Z_1 の協調済 BPM $M_1' = (O_h, (S_h, \lambda_h, F_h, \Phi_h))$ を，以下のとおり N の “射影” をとって，作成する．

- $O_h = \{o_1 \,|\, (o_1, o_2) \in P\}$, $S_h = \{s_1 \,|\, (s_1, s_2) \in T\}$, $\lambda_h = \lambda_1$, $F_h = \{s_1 \,|\, (s_1, s_2) \in G\}$
- $\Phi_h(s_1, o_1) = \{t_1 \,|\, (t_1, t_2) \in \Gamma((s_1, s_2), (o_1, o_2)) \,\&\, (s_1, s_2) \in T \,\&\, (o_1, o_2) \in P\}$

以上に述べたのは 2 つのシステム間での DMH アルゴリズムである．これを拡張し 3 つ以上の BPM を動的協調させるためのアルゴリズムも [30] で提案されている．

2.5 DMH の例

図 3 に DMH による BPM の変形例を示す．システム Z_1 が図 3 (a) の BPM を持つとする．丸印は状態を表し矢印は状態の遷移を表す．状態 0 は開始状態を表し二重丸は終了状態を表す．矢印の下にはオペレーションと括弧内に入出力区分（in または out）と format を示す．見積依頼し（AskQuotation），見積書を受け取り（GetQuotation），発注し（SendOrder），注文請書を受信し（GetOrderConf），支払通知する（SendPayment），必要なら途中で中断連絡する（Quit）．見積りを省略していきなり発注（SendOrder）から始めることもできる．各々の format（の名前）は f1, f2, f3, f4, f5, f6 であり[*3]，すべて同一名前空間に定義されているものとする．

[*3] これらは見積依頼書（QuotationRequest），見積書（Quotation），注文書（Order），注文請書（OrderConfirmation），支払通知書（PaymentNotification），中断連絡（QuitNotofication）に対応するが，紙面の都合で短い名前にした．

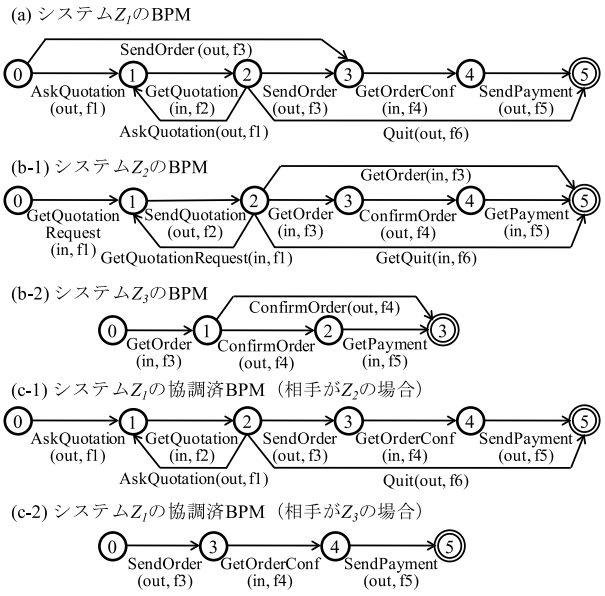


図 3 DMH の例
Fig. 3 Examples of DMH.

この BPM を持つシステム Z_1 がシステム Z_2 または Z_3 と遭遇し，それぞれの BPM が図 3 の (b-1) と (b-2) だったとする．これらの BPM には，システム Z_1 の BPM に対応して，見積受付（GetQuotationRequest），見積回答（SendQuotation），注文受付（GetOrder），受注連絡（ConfirmOrder），支払確認（GetPayment），中断受付（GetQuit）が定義されているが，図のとおりオペレーションの流れは Z_1 の BPM とは一致していない．

オペレーションマッチングには 2.3 節に述べた自明な tMatch を用いるものとして，DMH を実行すると，Z_2 または Z_3 に対して Z_1 の BPM は図 3 (c) のように変形される．

3. AWS ミドルウェア

AWS ミドルウェアは，2008 年から開発開始し [15]，基本方式の設計 [17]，メッセージングの改良 [18], [21]，DMH の改良 [30]，ビジネスプロセスの並行実行化 [20] などを経て，2011 年度に完成した．この章では，AWS ミドルウェアの目的と概要，実現方式および実現仕様について述べる．プログラム実装面については 4 章で後述する．

3.1 AWS ミドルウェアの目的と概要

AWS ミドルウェアの目的は，BPM の変形，アプリケーションフローの動的な変更，あるいは HTTP と SOAP を使ったメッセージングプロトコルなどの実装詳細をアプリケーションプログラム（AP）から隠蔽することである．AWS ミドルウェアを使えば，AP 開発者はこれらの詳細を知ることなく，自システムの BPM を記述してビジネスロジックをプログラミングするだけで，AWS を適用したシステムを容易に作成できる．

図 4 に示すように AWS ミドルウェアは大きく 3 つの

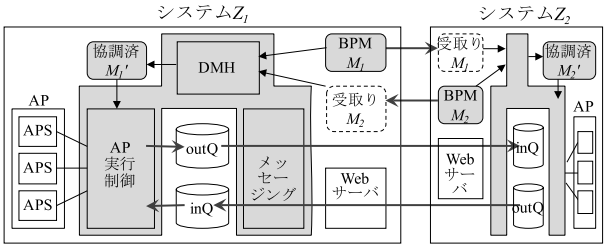


図 4 AWS ミドルウェアの概観
Fig. 4 Outline of the AWS middleware.

機能を持つ．相手システムから BPM を受け取って DMH アルゴリズムにより協調済 BPM を生成する機能，協調済 BPM に基づいて AP を実行制御する機能，および，AP 実行にともなうメッセージ交換を行うためのメッセージング機能である．

AWS では，AP 本体は，各オペレーションに対応するプログラム断片（AP セグメント（APS）と呼ぶ）の集合として記述される．AP 実行制御では協調済 BPM の状態変化に従って APS を実行する．メッセージング機能層はリライアブルな対等型非同期メッセージングのプロトコルエンジンであり，入出力キュー（outQ と inQ）を使って Web サーバ経由で相手システムとメッセージテキストをやりとりする．

3.2 BPM の外部表現

2.1 節では BPM を数学的に形式定義したが，AWS ミドルウェアでは BPM は 2 つの表現形態をとる．他システムとの交換などのための外部的な表現と，協調済 BPM をミドルウェア内で受け渡すための内部的な表現（4.1 節で後述）である．外部表現は Web サービスの慣行に従って XML で表現することにした．具体的には利用者の便宜を考慮して，状態遷移表型と正規表現型の 2 種類の XML 表現形態を準備している [19]．**図 5** に図 3 (a) の BPM を状態遷移表型で記述した例を示す．

3.3 モデル駆動型 AP 実行

DMH の結果 BPM が変形される．すなわち可能なビジネスプロセスの流れが変化する．それに合わせて AP 内の処理の流れも変化しなければならない．これをどう制御するかが 1 つの課題であった．AWS ミドルウェアではモデル駆動型 AP 実行と名付けた新たな方式によって解決した [14], [22]．以下にその概略を述べる．

前述のとおり，アプリケーション開発者は AP 本体を APS の集合として記述しておく．実際には APS は Java メソッドの形をとる．これとは別にアプリケーション開発者は AWS ミドルウェアの config ファイル内でマッピング：$o_i \rightarrow (meth_i, Con_i)$ を定義しておく[*4]（ただし $o_i \in O$）．

[*4] config ファイル内でのマッピング記述方法が XML 形式で定められている [22].

```
<BPModel>
 <operations>
  <operation name="AskQuotation" format="f1">
   <pattern>output</pattern></operation>
  <operation name="GetQuotation" format="f2">
   <pattern>input</pattern></operation>
  <operation name="SendOrder" format="f3">
   <pattern>output</pattern></operation>
  <operation name="GetOrderConf" format="f4">
   <pattern>input</pattern></operation>
  <operation name="SendPayment" format="f5">
   <pattern>output</pattern></operation>
  <operation name="Quit" format="f6">
   <pattern>output</pattern></operation>
 </operations>
 <behavior>
  <states>
   <state no="0">
    <next operation="AskQuotation">1</next>
    <next operation="SendOrder">3</next></state>
   <state no="1">
    <next operation="GetQuotation">2</next></state>
   <state no="2">
    <next operation="SendOrder">3</next>
    <next operation="AskQuotation">1</next>
    <next operation="Quit">5</next></state>
   <state no="3">
    <next operation="GetOrderConf">4</next></state>
   <state no="4">
    <next operation="SendPayment">5</next></state>
   <state no="5"></state>
  </states>
  <first>0</first>
  <last>5</last>
 </behavior>
</BPModel>
```

図 5 BPM の外部表現（記述例）
Fig. 5 External representation of BPM (example).

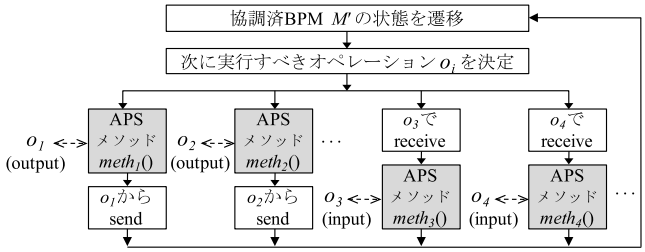


図 6 モデル駆動型 AP 実行
Fig. 6 Model driven AP execution.

なお $meth_i$ と Con_i はそれぞれ o_i に対応する APS メソッドの名前とコンテナクラス名である．**図 6** のように，AP 実行制御部は AP 実行にともない協調済 BPM M' の状態を逐次遷移させ次に実行すべきオペレーション o_i を決定して，イベント駆動型ループのように，モデルの状態に応じて対応する APS メソッド $meth_i$ を起動する．APS メソッドは，メッセージ内のデータをコンテナオブジェクトにカプセル化された形で，AP 実行制御部から受け取る（または渡す）．メッセージング層を介してのメッセージテキストの送信（send）と受信（receive）は APS の外側で AP 実行制御部が実行する．なお，o_i が input の場合は $meth_i$ 起動前に o_i に対応する receive が実行され，o_i が output のときは $meth_i$ からリターン後に send が実行されることに注意されたい．

3.4 アプリケーションプログラムインタフェース

委譲パターンを適用（4.2 節参照）し AP 開発者からみ

```
public class App {
  public void apAskQuotation(Con1 out) {
    out.item = "TV-45001"; out.qty = 6;   }
  public void apGetQuotation(Con2 in) {
    /* check whether in.price and
       in.deliveryDate is acceptable */   }
  public void apSendOrder(Con3 out) {
    out.item = "TV-45001"; out.qty = 6;
    out.price = ...;                      }
  public void apGetOrderConf(Con4 in) {
    /* check whether the response
       in in.result is acceptable */      }
  public void apSendPayment(Con5 out) {
    out.paymentDate = ...; ...;           }
  public void apQuit(Con6 out) {
    out.quitReason = "Estimation only";   }
}
```

図 7 AP 本体クラスのプログラム例
Fig. 7 Program example of AP body class.

て極力単純な API となるようにした．AP 開発者は各オペレーションに対応するビジネスロジックを記述したメソッドからなるクラスを AP 本体として作成すればよい．**図 7** は図 3 (a) と図 5 の BPM に対応する AP 本体の例である．config ファイルによって，BPM 内の 6 個のオペレーションに APS メソッド apAskQuotation()，apGetQuotation()，apSendOrder()，apGetOrderConf()，apSendPayment()，apQuit() と，コンテナクラス Con1，Con2，Con3，Con4，Con5，Con6 が対応付けられているものとしている．APS メソッドの中ではコンテナクラスに用意してあるプロパティを使って入出力データにアクセスしている．AP 開発者はコンテナオブジェクトのアクセスのためのプロパティやメソッドのほかに，2 つのコールバックメソッド generateMessageData() と parseMessageData() も実装しておく必要がある．前者は send すべきメッセージテキストを生成し，後者は receive したメッセージテキストを解析するメソッドであり，必要に応じて AP 実行制御部が呼び出す．

4. AWS ミドルウェアの実装

AWS ミドルウェアは Java を開発言語，Tomcat をサーバ実行環境として実装されている．3.1 節で述べた 3 つの機能に対応して 3 つのソフトウェア層から構成されている．

- 動的モデル協調層：DMH を実行し協調済 BPM を生成する．
- フレームワーク層：モデル駆動型で AP を実行制御する．
- メッセージング層：対等型非同期メッセージングのプロトコルエンジン．

4.1 動的モデル協調層

動的モデル協調層はクラス DMH として実装されている．**図 8** に示すように，AP は DMH オブジェクトを生成した後，自システムの BPM の URL（図では URL1）と相手システムの URL（図では URL2）を，DMH オブジェクトに連絡しておく（詳しくは AWSD という別ファイルを介して連絡）．一般に URL1 はローカルファイルで URL2 はインターネット上の URL だが，必ずしもそれに限らない．たとえば以前に受け取ってキャッシュしてある BPM を使う場合は URL2 がローカルファイルになる．これにより様々な BPM 公開形態に対応できる．

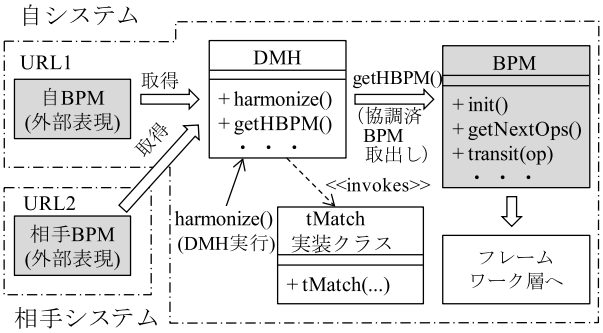


図 8 動的モデル協調層の構成
Fig. 8 Structure of the dynamic model harmonization layer.

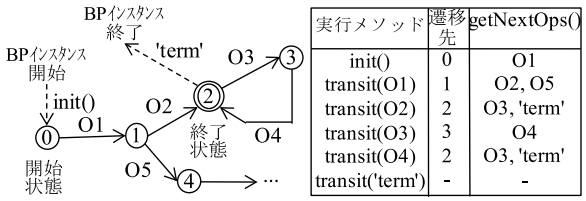


図 9 BPM クラスのメソッド実行例
Fig. 9 Execution example of the BPM class methods.

tMatch() はインタフェースだけが定義されており，AP は適当な実装クラスを前もって DMH オブジェクトに登録しておく．その後 AP が harmonize() で DMH の実行を指示すると，DMH オブジェクトは HTTP プロトコルまたはファイルアクセスを用いて 2 つの BPM を取得し，その後 DMH アルゴリズムを実行する．動的協調に成功すれば協調済 BPM オブジェクトが作成される．協調済 BPM オブジェクトは BPM クラスのインスタンスであり，フレームワーク層に渡される．

BPM クラスの主なメソッドは以下である：

- init()： 現在の状態を開始状態にセットする．
- getNextOps()： 現在の状態から次に実行可能なオペレーションの集合を返す．
- transit(*op*)： 現在の状態からオペレーション *op* を実行して次の状態に遷移させる．

図 9 はこれらのメソッドを使った状態遷移の例を示したものである．なお図中の 'term' はこのビジネスプロセスインスタンス（4.2 節参照）を終了させることを意味する．

4.2 フレームワーク層

図 10 はフレームワーク層の構成を示したものである．

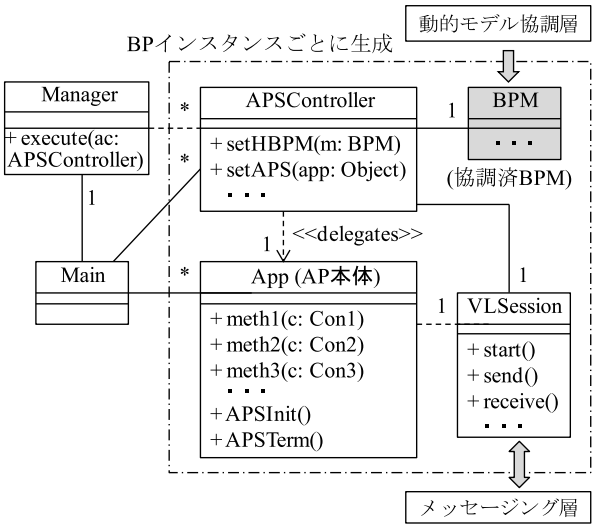


図 10 フレームワーク層の構成
Fig. 10 Structure of the framework layer.

```
. . .
dmh = new DMH();                          //(0)
dmhにURLなどを登録;
dmh.harmonize();
. . .
m = new Manager();                        //(1)
while(true) {
  waitFor(BPインスタンス開始指示);
  ac = new APSController();               //(2)
  app = new App(...);
  ac.setHBPM(dmh.getHBPM());              //(3)
  ac.setAPS(app);                         //(4)
  m.execute(ac);                          //(5)
}
```

(0) DMHオブジェクトを生成し，動的モデル協調を実行.
(1) Managerオブジェクトを生成.
(2) APSControllerとAP本体をインスタンス化.
(3) acに協調済BPMのコピーを登録.
(4) acの委譲先としてappを登録.
(5) BPインスタンスの実行を開始.

図 11 Main クラスの典型的な処理概略
Fig. 11 Typical processing in a Main class.

利用者が開発するのは，Main クラスとアプリケーション本体のクラス（図中の App）だけである．他は AWS ミドルウェアの一部である．

この層の中心は AP の実行を制御する APSController クラスである．App は APSController の委譲先（デリゲーション）[*5]として登録され，これによって App 内の APS メソッドが APSController から実行できるようにしている．

あるアプリケーションが，ある相手システムと協調済 BPM を使って，開始状態から開始して終了状態で停止するまでに行う一連の処理をビジネスプロセスインスタンス（BP インスタンス）と呼ぶ．おおざっぱにいえば，ビジネスプロセスが取引手順を意味するのに対して，BP インスタンスは毎回の取引を意味する．図 10 の一点鎖線内は，BP インスタンス発生時に毎回インスタンス化され，その BP インスタンスが終了すると削除される．複数の BP インスタンスを同時にインスタンス化することも可能であり，そのときはインスタンスごとに実行スレッドが割り当てられ並行実行される．

上記は利用者アプリケーションの一部である Main の中で制御する．Main の処理内容は利用者業務依存だが，典型的な処理概略は**図 11** のとおりである：(0) あらかじめ DMH オブジェクトを生成し動的モデル協調（DMH）を実行しておく．(1) まず Manager オブジェクトを生成する．(2) BP インスタンスの開始時に（すなわち新たな取引を開始するときに），APSController オブジェクト ac と App オブジェクト app を生成する．(3) ac に協調済 BPM オブジェクトを登録する．(4) ac の委譲先として app を登録する．(5) Manager オブジェクトに execute() を実行することで ac の実行開始を指示する．

実行が指示されると，APSController オブジェクト ac は，4.1 節に述べた BPM クラスのメソッドを使って協調済 BPM の状態を変化させながら適宜 app 内の対応メソッドを起動する．またそれにともなって発生するメッセージの送信と受信をメッセージング層の代理オブジェクトである VLSession オブジェクトを介してメッセージング層に依頼する．なお APSInit() と APSTerm() は BP インスタンスの開始時と終了時に起動される特別なメソッドである．

BP インスタンスは並行して発生することが珍しくないため，AWS ミドルウェアでは各 ac（およびその委譲先である app）にそれぞれ異なるスレッドを割り当てて並行実行させている．一方，多数の BP インスタンスが同時発生する（つまり多数の取引が同時並行進行する）ことも多いため，そのままではスレッド数が過多になりシステムに悪影響が及ぶ．そこで，Manager はスレッドプールを使って ac にスレッドを割り当てるように実装されている [20]．execute() 実行時に十分なスレッドがプールされていない場合，ac はスレッド待ちキューに入れられ，スレッドが空くまで実行保留にする．また，実行時に receive() が長期間待ちになったとき（たとえば相手システムからの見積り結果連絡に時間がかかっているとき）は，当該 ac からいったんスレッドを取り去り，一定時間待ったのち，再度スレッド待ちキューに入れる．これにより長期間応答待ちが発生しても全体がブロックされないようにしている．

4.3 メッセージング層

メッセージング層のメカニズムは上位層から完全に隠蔽される形で実装されている．アプリケーションの端点間での対等型，非同期，store-and-forward かつリライアブルなメッセージ交換機能を提供する．システムの自律性を確保するために，従来のメッセージング基盤（MQ や JVM など）で一般的な事前定義されたメッセージングチャネルをシステム間で共有する方式ではなく，ビジネスプロセス

[*5] 当初は継承を用いていたが，上位クラスが使用できないため，後に委譲パターンに変更された [31].

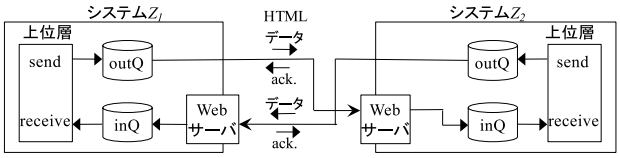


図 12 メッセージング層の下位システム構成（対称型）
Fig. 12 System structure under the messaging layer (symmetrical).

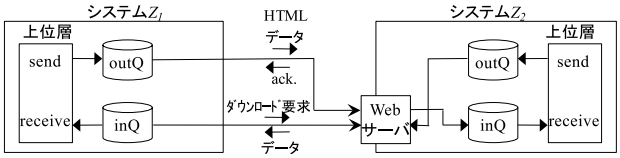


図 13 メッセージング層の下位システム構成（非対称型）
Fig. 13 System structure under the messaging layer (asymmetrical).

を開始する時点で VLSession と呼ばれる長期間持続可能なセッションを動的生成する方式としている．プロトコルは，HTTP と SOAP/Framework [2] の上の ebXML メッセージングサービス [3] のプロトコルを簡略化したものを使用している．

下位システム構成としては，両システムに Web サーバが存在する構成（対称型構成）（**図 12**）と一方にしか Web サーバが存在しない構成（非対称型構成）（**図 13**）の 2 つをサポートしている [18]．前者は一般の企業間のときを，後者は一方が小規模オフィスやモバイルなどのときを想定している．対称型では HTTPrequest でキュー内のデータを伝送し，HTTPresponse は単に結果を acknowledge するだけである．非対称型では Web サーバが存在しない側（図 13 のシステム Z_1）から Web サーバが存在する側（システム Z_2）への伝送は対称型と同じだが，システム Z_2 からシステム Z_1 へのメッセージ伝送は，システム Z_1 が適宜ダウンロード要求の HTTPrequest を出し，システム Z_2 の出力キューに送信待ちデータが存在していたら HTTPresponse でデータを伝送する．

他の考慮点としては DBMS を用いた入出力キューの実装があげられる [14], [21]．BP インスタンスの開始から終了までが長期間かかることを考慮して，耐久性面などから DBMS（PostgreSQL）を用いて実装した．リライアブルメッセージングに必要な再送制御やメッセージ順序保障制御も DBMS 内に複数のテーブルを定義して実現した．

5. 評価実験

DMH アルゴリズムについては，提案時に机上シミュレーションテストを用いてその妥当性と有効性が理論面から評価されている [10], [12]．AWS ミドルウェアについては，実装開発の一環として各種の人工的テストを行い，ソフトウェアとしての妥当性検証を行った．しかし，本論文でこれまで述べた AWS の全体方式，基本原理，ミドルウェアの基本方式とインタフェース，およびミドルウェアの各層の実装方式の妥当性と有効性を実用面を含めて評価するにはこれらのテストだけでは不十分である．そこで，開発した AWS ミドルウェアを使い，実運用を想定した疑似システムを作成し，総合的な評価実験を行った [26]．以下その内容概略と結果を述べる．

表 1 実運用を想定したシステムによる実験結果
Table 1 Experiment results assuming practical application.

	S1	S2	S3	S4	S5	S6	S7	S8	S9
B1	○	○	○	○	○	○	○	○	○
B2	○	○	○	○	○	○	○	○	○
B3	○	○	○	○	○	○	○	○	○
B4	○	○	○	○	○	○	○	○	○
B5	○	○	○	○	○	□	□	□	□
B6	○	○	○	○	□	○	○	○	○
B7	○	○	○	○	□	○	○	○	○
B8	○	○	○	○	□	○	○	○	○
B9	○	○	○	○	□	○	○	○	○

○：正常に取引完了　□：DMHで取引実行不能と事前検出

実験では典型的な見積り・発注・受注の業務を想定した．購入側は，5 つのオペレーション（見積依頼，見積受信，発注，請書受信，中断連絡）を想定し，これらを網羅的に組み合わせて，実運用上考えられる BPM を 9 個（B1〜B9）作成した（付録 A.1）．販売側は，対応する 5 つのオペレーション（見積受付，見積回答，注文受付，受注連絡，中断受付）を想定し，同様に，実運用上考えられる 9 個の BPM（S1〜S9）を作成した（付録 A.2）．さらにこれらの BPM に対応した 18 個の疑似取引システムを作成し，9 個 × 9 個 = 81 通りのテストを行った．結果を**表 1** に示す．73 通り（図中 ◯）で取引（BP インスタンス）が最後まで完了した．残りの 8 通り（□）は BPM の整合化が不可能であることが DMH の段階で判明したため実際の取引開始前に処理が中止された．この実験をとおして実運用を想定したシステムでも，DMH が妥当かつ効果的に動作することが確認できたとともに，AWS ミドルウェアの基本方式と実装方式が妥当であり，AWS が実用的なソフトウェアとして実装可能であることも検証できた．またアプリケーション内でユーザ入力待ちになっている時間を除けば，1 回の BP インスタンスは 2〜3 秒で終了しており，DMH アルゴリズム実行時間を含めて性能上の大きな問題がないことも検証できた．

なお補足として**表 2** に，もし AWS を適用せずに（したがって AWS ミドルウェアも使用せずに）従来の Web サービスをそのまま用いて同様のテストを実行したらどうなるかを机上シミュレーションで求めた結果を示す．表 1 では正常に取引完了（◯）であった 41 通りで，表 2 では事前の BPM 協調を行わないため，許容されないオペレーションを受け取り実行時に誤動作を起こす可能性がある（△）となっている．さらに表 1 では取引実行不能と事前検出（□）できる 8 通りについても，表 2 では取引実行中に誤

表 2 AWS を適用しない場合のシミュレーション結果
Table 2 Simulation results when not applying AWS.

	S1	S2	S3	S4	S5	S6	S7	S8	S9
B1	○	△	△	△	△	△	△	△	△
B2	○	○	△	△	△	△	△	△	△
B3	○	△	○	△	△	△	△	△	△
B4	○	○	○	○	△	△	△	△	△
B5	○	○	○	○	○	×	×	×	×
B6	○	△	△	△	×	○	△	△	△
B7	○	○	△	△	×	○	○	△	△
B8	○	△	○	△	×	○	△	○	△
B9	○	○	○	○	×	○	○	○	○

○：正常に取引完了 ×：取引実行中に誤動作を起こす
△：取引実行中に誤動作を起こす可能性あり

動作を起こす（×）．AWS の有効性を示す一事例である．

6. おわりに

AWS は次世代の Web サービスのための技術で，複数システムをまたがる全体的な BPM を前提とせず，個々のサイトが独立に持っている BPM を動的協調させることで，電子商取引などのビジネスメッセージ交換を可能とすることを狙いとしている．そのために我々は，AWS の基本原理である動的モデル協調（DMH）を考案し，形式定義された BPM を協調変形するアルゴリズムを提案した．AWS ミドルウェアの目的は，AWS 詳細メカニズムをアプリケーションから隠蔽することである．AWS ミドルウェア実現のために，モデル駆動型 AP 実行と呼ばれる基本方式を考案し，新たなアプリケーションインタフェースを提案した．XML を使って BPM の外部表現を定めた．それらを基に AWS ミドルウェアを 3 層構造のソフトウェアとして実装した．動的モデル協調層では DMH アルゴリズムの実装に加えて URL で示された BPM を自動的に取り込むなどの新規の工夫を加えた．フレームワーク層については，委譲パターンを適用し AP 開発量が軽減するインタフェースを提案した．さらに，BP インスタンスという新概念を導入し，動的なスレッド割付けを実現し，そのための API を考案した．メッセージング層は，対等非同期型リライアブルメッセージング機能を，両側に Web サーバが存在する対称型構成と片側にしか存在しない非対称型構成の両方で実装した．さらに，理論的評価や通常ソフトウェアテストに加え．実運用を想定した疑似システムを実際に作成して，トータルな実証評価を実施した．これにより，提案した基本原理とアルゴリズム，AWS ミドルウェアの基本方式やインタフェース，実装メカニズムについて，妥当性，有効性および実用性が評価できた．また，プログラムインタフェースなど利用者視点からの使いやすさも評価できた．

今後の課題は，理論面からは BPM の記述能力向上と tMatch() 実装方法があげられ，実装面からは長期にわたる BP インスタンスの途中アーカイブとスレッド制御，および必須フローの保証があげられる．また，現状 Web サービスへの AWS の部分適用，インターネット環境の他にモバイル環境やユビキタス環境への応用も課題である．

謝辞 実装開発を中心に以下の湘南工科大学学生の協力を得た（敬称略）：伊東正起，澤口宗和，木村泰輔，高木良輔，塚本修也，吉川恭平，大友浩照，平本真道，安齋太一朗，二宮良太．また，本研究は科研費（21500110）の助成を受けたものである．

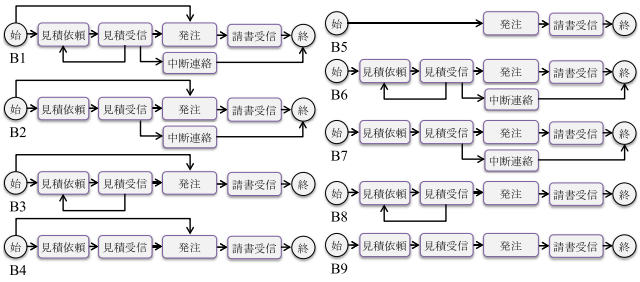
参考文献

[1] Box, D., Ehnebuske, D., Kakivaya, G., et al.: SOAP 1.1, W3C Note (2000).
[2] Gudgin, M., Hadley, M., Mendelsohn, N., et al.: SOAP 1.2, W3C Recommendation (2007).
[3] OASIS: ebXML Messaging Services Version 3.0, OASIS Standard (2007).
[4] 大谷 真：コンポーネント化と分散オブジェクト技術—インタフェースプログラミングの視点から，ビジネスオブジェクト入門，今城哲二（監修），pp.291–334, SRC (2000).
[5] Chinnici, R., Moreau, J., Ryman, A., et al.: WSDL 2.0, W3C Recommendation (2007).
[6] OASIS: W.S. Business Process Execution Language 2.0, OASIS Standard (2007).
[7] OMG: Business Process Model and Notation (BPMN) 1.2, omg/formal/2009-01-03 (2009).
[8] Erl, T., Karmarkar, A., Walmsley, P., et al.: *Web Service Contract Design and Versioning for SOA*, Prentice Hall (2009).
[9] Miller, J., et al.: MDA Guide Version 1.0.1, OMG doc. omg/2003-06-01 (2003).
[10] 大谷 真，木下正博，嘉数侑昇：自律的 Web サービスにおけるビジネスプロトコルの動的生成について，電子情報通信学論文誌，Vol.J87-D-I, No.8, pp.824–832 (2004).
[11] Oya, M. and Ito, M.: Dynamic Model Harmonization between Unknown eBusiness Systems, *I3E 2005 IFIP*, pp.389–403, Springer (2005).
[12] Oya, M., Kinoshita, M. and Kakazu, Y.: On Dynamic Generation of Business Protocols in Autonomous Web Services, *Systems and Computers in Japan*, Vol.37, No.2, pp.37–45, Wiley (2006).
[13] 大谷 真：モデル動的協調による自律対等型 Web サービスのアーキテクチャ，情報処理学会第 70 回全国大会講演論文集，pp.1-457–458 (2008).
[14] Oya, M.: Autonomous Web Services Based on Dynamic Harmonization, *I3E 2008 IFIP*, pp.139–150, Springer (2008).
[15] 伊東正起, 塚本修也, 高木良輔, 木村泰輔, 大谷 真：AWS ミドルウェアの研究—アプローチと構成，動的モデル協調層，アプリケーションフレームワーク層，自律型メッセージング層，情報処理学会第 71 回全国大会講演論文集，pp.1-509–516 (2009).
[16] 大谷 真, 伊東正起, 塚本修也, 高木良輔, 木村泰輔：AWS（自律型 Web サービス）とそのミドルウェア，情報処理学会第 71 回全国大会講演論文集，pp.1-503–504 (2009).
[17] 伊東正起，平本真道，大友浩照，大谷 真：動的協調ミドルウェアの実装方法の研究，情報処理学会第 72 回全国大会講演論文集，pp.1-787–788 (2010).
[18] 木村泰輔，吉川恭平，伊東正起，大谷 真：AWS メッセージング基盤改良の検討/非対称構成型メッセージング機能の実現，情報処理学会第 72 回全国大会講演論文集，pp.1-793–796 (2010).
[19] 大友浩照，伊東正起，吉川恭平，大谷 真：AWS におけるビジネスプロセスモデル，情報処理学会第 72 回全国大会講演論文集，pp.1-789–790 (2010).

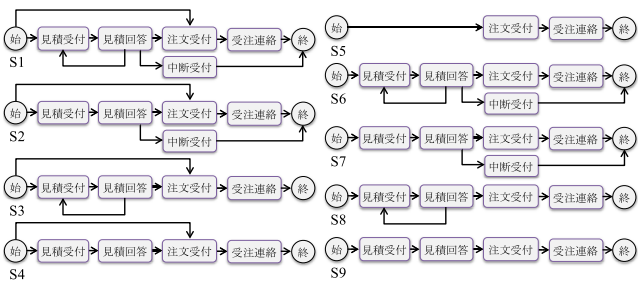
[20] 二宮良太，平本真道，安齋太一朗，大谷 真：AWS（自律型 WEB サービス）ミドルウェアフレームワーク制御，情報処理学会第 73 回全国大会講演論文集，pp.1-707–708 (2011).
[21] 木村泰輔，二宮良太，平本真道，大谷 真：自律型 Web サービスメッセージング基盤の開発，情報処理学会第 73 回全国大会講演論文集，pp.1-709–710 (2011).
[22] Oya, M., Ito, M. and Kimura, T.: Middleware for the Autonomous Web Services (AWS), *I3E 2010, IFIP AICT341*, pp.5–16, Springer (2010).
[23] 大谷 真：自律型 Web サービス（AWS）の原理と実装，情報処理学会第 74 回全国大会講演論文集，pp.1-37–38 (2012).
[24] 大友浩照，二宮良太，大谷 真：AWS ミドルウェアにおける動的モデル協調層の開発，情報処理学会第 74 回全国大会講演論文集，pp.1-583–584 (2012).
[25] 平本真道，木村泰輔，大友浩照，大谷 真：実応用を想定した AWS ミドルウェアの評価，情報処理学会第 73 回全国大会講演論文集，pp.1-703–704 (2011).
[26] 平本真道，安齋太一郎，大谷 真：統一的なビジネスプロトコルを前提としない AWS 電子商取引の研究，情報処理学会第 74 回全国大会講演論文集，pp.4-775–776 (2012).
[27] 安齋太一朗，平本真道，大谷 真：AWS における電子帳票の互換性決定の実装検討，情報処理学会第 74 回全国大会講演論文集，pp.2-441–442 (2012).
[28] Martino, B.: An Ontology Matching Approach to Semantic Web Services Discovery, Lecture Notes in Computer Science, pp.550–558, Springer (2006).
[29] Chabeb, Y., Tata, S. and Belaid, D.: Toward an integrated ontology for web services, *Proc. ICIW 2009*, pp.462–467 (2009).
[30] 大友浩照，大谷 真：自律型 Web サービスにおける複数システム間での動的モデル協調，FIT2011 講演論文集第 4 分冊，RO-012, pp.165–168 (2011).
[31] 二宮良太，大友浩照，大谷 真：委譲型 AWS フレームワークの実現方法，情報処理学会第 74 回全国大会講演論文集，pp.1-83–84 (2012).

付　　録

付録 A.1　実運用を想定した実験の BPM（購入側，9 種類）



付録 A.2　実運用を想定した実験の BPM（販売側，9 種類）



大谷 真　（正会員）

1972 年早稲田大学理工学部数学科卒業．同年（株）日立製作所入社．2003 年北海道大学大学院情報科学研究科教授．2005 年 10 月湘南工科大学工学部情報工学科教授．博士（工学）．OS/ミドルウェア，Web サービス，自律システム，モデリングを研究．電子情報通信学科，ACM，観光情報学会各会員．