Formal Model for Semantic-Driven Service Execution

Tomas Vitvar[1] and Adrian Mocan[1] and Maciej Zaremba[2]

[1] Semantic Technology Institute (STI2)
University of Innsbruck, Austria
firstname.lastname@sti2.at
[2] Digital Enterprise Research Institute (DERI)
National University of Ireland in Galway, Ireland
maciej.zaremba@deri.org

Abstract Integration of heterogeneous services is often hard-wired in service or workflow implementations. In this paper we define an execution model operating on semantic descriptions of services allowing flexible integration of services with solving data and process conflicts where necessary. We implement the model using our WSMO technology and a case scenario from the B2B domain of the SWS Challenge.

1 Introduction

Existing technologies for service invocation and interoperation usually depend on ad-hoc or hard-wired solutions for interoperability. In particular, message level interoperability is often maintained in business processes using XSLT, and process level interoperability is often achieved through manual configuration of workflows. Such rigid solutions are a drawback to services' flexibility and adaptability: changes in service descriptions require changes in service implementation or workflows. One possible approach to improve the interoperability is to use semantics in service descriptions. With help of semantics and a logical reasoning, it is possible to automate the integration process and achieve an integration that is more adaptive to changes in business requirements.

In Semantic Web Services (SWS), there are two phases in the service integration process, namely *late-binding phase* and *execution phase* [15]. Late-binding phase allows binding a user request and a set of services "on-the-fly" through semi-automation of the *service lifecycle* by applying tasks for service discovery, composition, selection, mediation, etc. Execution phase allows for the invocation and conversation of bound services. While services may have heterogeneous descriptions in terms of data and protocols, it is important to achieve their interoperability within the both phases. In this paper we elaborate on the execution phase and show how semantic services can be decoupled in the integration process and how their interoperability can be achieved through combined data and process mediation. Particular contributions of our paper are as follows:

– We define a sound conceptual model for data and process mediation for SWS exectuion extending our previous, more technical and implementation-driven work in [7].
– We built on top of existing results from the area of ontology-based data mediation [11] by providing a formal algorithm that shows how a run-time mediation can be interlaced with other type of mediation, that is, process mediation.
– We show how the formal model for service exectution can provide a solution for a real-world case scenario. For this purpose we describe the implementation using the WSMO[13], WSML[13], WSMX[3] including a solution architecture for a case scenario from the SWS Challenge[4].

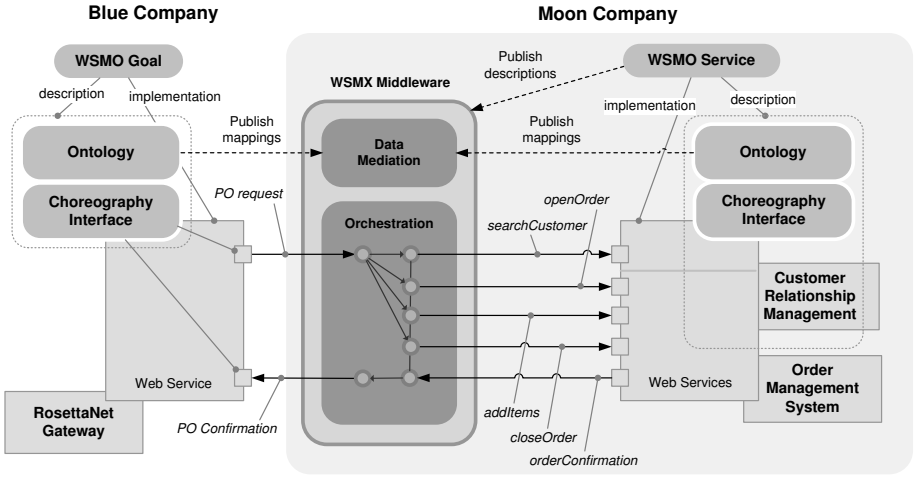

Figure 1. Case Scenario and Solution Architecture

In order to demonstrate a problem we target in our work, Figure 1 depicts a case scenario of the SWS Challenge mediation problem. In the scenario, a trading company, called Moon, uses a Customer Relationship Management system (CRM) and an Order Management System (OMS) to manage its order processing. Moon has signed agreements to exchange Purchase Order (PO) messages with a company called Blue using the RosettaNet standard PIP3A4. There are two interoperability problems in the scenario: At the *data level*, the Blue uses PIP3A4 to define the PO request and confirmation messages while Moon uses a proprietary XML Schema for its OMS and CRM systems. At the *process level*, the Blue follows PIP3A4 Partner Interface Protocol (PIP), i.e. it sends out a

[3] http://www.wsmx.org
[4] SWS Challenge, `http://www.sws-challenge.org`, defines a testbed together with a set of increasingly difficult problems on which various SWS solutions can be objectively evaluated.

PIP3A4 PO message, including all items to be ordered, and expects to receive a PIP3A4 PO confirmation message. On the other hand, various interactions with the CRM and OMS systems must be performed in Moon in order to process the order, i.e. get the internal ID for the customer from the CRM system, create the order in the OMS system, add line items into the order, close the order, and send back the PO confirmation.

In section 4 we further describe a solutuion for the scenario building on our SWS technology implementing the service execution model. In Section 2 we provide some background definitions for this model and in Section 3 we formaly define this model.

2 Definitions

A service engineer or a client (depending on the level of automation) must decide whether to bind with services according to the descriptions of *service contracts*. In SWS, we represent service contracts at the *semantic level* and *non-semantic level*. For the non-semantic level we use Web Service Description Language (WSDL) and for the semantic level we use four types of descriptions: *information*, *functional*, *non-functional* and *behavioral*. In addition, grounding defines a link between semantic and non-semantic descriptions of services.

For purposes of the execution phase, we provide the background definitions for information and behavioral semantic descriptions of services together with grounding to WSDL. In this paper we do not use the other types of semantic descriptions, please refer to e.g. [14] for more information about these descriptions. In addition, we provide definitions for the two major stages of the execution phase, that is, data mediation and process mediation.

2.1 Information Semantics

Information semantics is the formal definition of some domain knowledge used by the service in its *input* and *output* messages. We define the information semantics as an ontology

$$O = (C, R, E, I) \tag{1}$$

with a set of classes (unary predicates) C, a set of relations (binary and higher-arity predicates) R, a set of explicit instances of C and R called E (extensional definition), and a set of axioms called I (intensional definition) that describe how new instances are inferred.

2.2 Behavioral Semantics

Behavioral semantics is a description of the public and the private service behaviors. For our work we only use the public behavior, called choreography,

describing a protocol, that is, all messages sent to the service from the network and all messages sent from the service back to the network[5]. We do not use the private behavior of the service, i.e. the internal workflow, in our model. We define the choreography X using the Abstract State Machine (ASM) as

$$X = (\Sigma, L), \tag{2}$$

where $\Sigma \subseteq (\{x\} \cup C \cup R \cup E)$ is the signature of symbols, i.e., variable names $\{x\}$ or identifiers of elements from C, R, E of some ontology O; and L is a set of rules. Further, we denote by Σ_I and Σ_O the input and output symbols of the choreography (subsets of $C \cup R \cup E$), corresponding to the input data sent to the service and the returned output data. Each rule $r \in L$ is defined as $r^{cond} \rightarrow r^{\mathit{eff}}$ where r^{cond} is an expression in logic $\mathcal{L}(\Sigma)$ which must hold in a state before the rule is executed; r^{eff} is an expression in logic $\mathcal{L}(\Sigma)$ describing how the state changes when the rule is executed.

2.3 Grounding

Grounding defines a link between semantic descriptions of services and various components of WSDL. We denote the WSDL schema as S and the WSDL interface as N. Further, we denote $\{x\}_S$ as a set of all element declarations and type definitions of S, and $\{o\}_N$ as a set of all operations of N. Each operation $o \in \{o\}_N$ may have one input message element $m \in \{x\}_S$ and one output message element $n \in \{x\}_S$.

There are two types of grounding used for information and behavioral semantics. The first type of grounding specifies *references* between input/output symbols of a choreography $X = (\Sigma, L)$ and input/output messages of respective WSDL operations $\{o\}_N$ with schema S. We define this grounding as

$$\mathit{ref}(c, m) \tag{3}$$

where $m \in \{x\}_S$, $c \in \Sigma$ and *ref* is a binary relation between m and c. Further, m is the input message of operations in $\{o\}_N$ if $c \in \Sigma_I$ or m is the output message of operations in $\{o\}_N$ if $c \in \Sigma_O$.

The second type of grounding specifies *transformations* of data from schema S to ontology $O = (C, R, E, I)$ called *lifting* and vice-versa called *lowering*. We define this grounding as

$$\mathit{lower}(c_1) = m \qquad \text{and} \qquad \mathit{lift}(n) = c_2, \tag{4}$$

[5] Please note, that our notion of the choreography is different from the one used by the Web Service Choreography Description Language (WS-CDL) defining the choreoraphy as a common behavior of collaborating parties (`http://www.w3.org/TR/ws-cdl-10/`)

MEP and Rule	WSDL Operation

in-out:
if c_1 then $add(c_2)$
$c_1 \in \Sigma_I$, $ref(c_1,$ `msg1`$)$
$c_2 \in \Sigma_O$, $ref(c_2,$ `msg2`$)$
	`<operation name="oper1" pattern="w:in-out">`
	`  <input messageLabel="In" element="msg1"/>`
	`  <output messageLabel="Out" element="msg2"/>`
	`</operation>`

in-only:
if c_3 then *no action*
$c_3 \in \Sigma_I$, $ref(c_3,$ `msg3`$)$
	`<operation name="oper2" pattern="w:in-only">`
	`  <input messageLabel="In" element="msg3"/>`
	`</operation>`

out-only:
if *true* then $add(c_4)$
$c_4 \in \Sigma_O$, $ref(c_4,$ `msg4`$)$
	`<operation name="oper3" pattern="w:out-only">`
	`  <output messageLabel="Out" element="msg4"/>`
	`</operation>`

out-in:
if *true* then $add(c_5)$
if $c_5 \wedge c_6$ then *no action*
$c_5 \in \Sigma_O$, $ref(c_5,$ `msg5`$)$
$c_6 \in \Sigma_I$, $ref(c_6,$ `msg6`$)$
	`<operation name="oper4" pattern="w:out-in">`
	`  <output messageLabel="Out" element="msg5"/>`
	`  <inpput messageLabel="In" element="msg6"/>`
	`</operation>`

Table 1. MEPs, Rules and WSDL operations

where $m, n \in \{x\}_S$, $c_1, c_2 \in (C \cup R)$, *lower* is a *lowering transformation function* transforming the semantic description c_1 to the message m, and *lift* is a *lifting transformation function* transforming the message n to the semantic description c_2.

A client uses both types of grounding definitions when processing the choreography rules and performing the communication with the service while following the underlying definition of WSDL operations and their Message Exchange Patterns (MEPs). Table 1 shows basic choreography rules for four basic WSDL 2.0 MEPs[6], i.e. *in-out*, *in-only*, *out-only*, *out-in* (please note that we currently do not handle fault messages), and corresponding WSDL operations. Here, a rule $r^{cond} \rightarrow r^{eff}$ is represented as `if` r^{cond} `then` r^{eff}; the symbols `msg1`...`msg6` refer to schema elements used for input/output messages of operations; the symbols $c_1 \ldots c_6$ refer to identifiers of semantic descriptions of these messages; $ref(m, c)$ denotes a reference grounding, and `w:` is a shortening for the URI `http://www.w3.org/ns/wsdl/`.

2.4 Data Mediation

Data mediation resolves interoperability conflicts between two services that use two different ontologies. In general, the data mediation has two stages: 1) creation of alignments between *source* and *target* ontologies during *design-time* and 2) applying the alignments to resolve interoperability conflicts during *run-time*. Since the interoperability problems can greatly vary in their nature and severity, fully automatic solution for the creation of alignments are not feasible in real-world case scenarios due to the lower than 100% precision and recall of ex-

[6] `http://www.w3.org/TR/wsdl20-adjuncts/#meps`

isting methods[7]. From this reason, the design-time data mediation stage is still dependent on manual support of a service engineer.

An alignment consists of a set of mappings (rules) expressing the semantic relationships that exist between the two ontologies. In particular, a mapping can specify that classes from two ontologies are equivalent while corresponding rules use logical expressions to unambiguously define how the data encapsulated in an instance of one class can be encapsulated in instances of the second class. Formally, we define an alignment A between source and target ontologies $O_s = (C_s, R_s, E_s, I_s)$ and $O_t = (C_t, R_t, E_t, I_t)$ as

$$A_{s,t} = (O_s, O_t, \Phi_{s,t}) \tag{5}$$

where $\Phi_{s,t}$ is the set of mappings m in the form

$$m =< \varepsilon_s, \varepsilon_t, \gamma_{\varepsilon_s}, \gamma_{\varepsilon_t} > \tag{6}$$

where ε_s, ε_t represent the mapped entities from the two ontologies while γ_{ε_s}, γ_{ε_t} represent restrictions (i.e. conditions) on these entities such as $\varepsilon_s \in C_s \cup R_s$, $\varepsilon_t \in C_t \cup R_t$ while γ_{ε_s} and γ_{ε_t} are expressions in logic $\mathcal{L}(C_s \cup R_s \cup E_s)$ and $\mathcal{L}(C_t \cup R_t \cup E_t)$, respectively.

In order to execute the mappings during the execution phase, these mappings must be *grounded*[8] to rules expressed in some logical language for which a reasoning support is available (in Section 4 we use the WSML language for this grounding). We obtain the set of rules $\rho_{s,t} = \Phi^G_{s,t}$ by applying the grounding G to the set of mappings Φ. In the following definitions, $\{x\}$ stands for the set of variables used by the mapping rule and x' and x'' are two particular variables.

Every mapping rule $mr \in \rho_{s,t}$ has the following form:

$$mr : \bigwedge_{i=1..n}^{\{x\}} mr_i^{head} \rightarrow \bigwedge_{i=1..n}^{\{x\}} mr_i^{body} \tag{7}$$

where

$$\begin{aligned} mr^{head} \in\ & \{x' \textbf{ instanceOf } \varepsilon \mid \varepsilon \in C_t \wedge x' \in \{x\}\} \cup \\ & \{\varepsilon(x', x'') \mid \varepsilon \in R_t\ \wedge \varepsilon(x', x'') \in E_t \wedge x', x'' \in \{x\}\} \end{aligned} \tag{8}$$

$$\begin{aligned} mr^{body} \in\ & \{x' \textbf{ instanceOf } \varepsilon \mid \varepsilon \in C_s \wedge x' \in \{x\}\} \cup \\ & \{\varepsilon(x', x'') \mid \varepsilon \in R_s\ \wedge \varepsilon(x', x'') \in E_s \wedge x', x'' \in \{x\}\} \cup \\ & \{\gamma_s \mid \gamma_s \in \mathcal{L}(C_s \cup R_s \cup E_s \cup \{x\})\} \cup \\ & \{\gamma_t \mid \gamma_t \in \mathcal{L}(C_t \cup R_t \cup E_t \cup \{x\})\} \end{aligned} \tag{9}$$

[7] The "*Ontology Alignment Evaluation Initiative 2006*" [5] shows that the best five systems' scores vary between 61% and 81% for precision and between 65% and 71% for recall.

[8] Please note, that this grounding is different to the grounding defined in Section 2.3.

A mapping rule is formed of a head and a body. The head is a conjunction of logical expressions over the target elements and describes the result of the mediation in terms of instances of the target ontology. The body is formed of a set of logical expressions over the source entities which represent the data to be mediated, plus a set of logical expressions representing conditions over both the source and the target data.

There are situations when there is no corresponding data in the source ontology as required by the target ontology such as when mapping prices with different currency units. These issues are, however, dependant on implementation of the data mediation and the reasoning engine. In our implementation, it is possible to specify an URI for a transformation function and its parameters as placeholders for the missing target values. It is the role of the reasoning engine to fill the parameters placeholders with data from the source ontology. The data mediation engine then executes the function and gets the data for the target ontology.

2.5 Process Mediation

Process Mediation handles interoperability issues which occur in descriptions of choreographies of the two services. In [3] Cimpian defines five process mediation patterns:

a. **Stopping an unexpected message** when one service sends a message which is not expected by the other service.
b. **Inverting the order of messages** when one service sends messages in a different order than the other service expects them to receive.
c. **Splitting a message** when a service sends a message which the other service expects to receive in multiple different messages.
d. **Combining messages** when a service expects to receive a message which is sent by the other service in multiple different messages.
e. **Generating a message** when one service expects to receive a message which is not supplied by the other service.

3 Execution Phase

Figure 2 depicts the main states of the execution phase In Section 3.1 we define the algorithm for the execution phase and in Section 3.2 we discuss some relevant aspects for the data and process mediation applied within the algorithm.

3.1 Algorithm

Input:

– Service W_1 and service W_2. Each such a service W contains the ontology (information semantics) $W.O$ (Eq. 1), the choreography $W.X$ (Eq. 2) with set of rules $W.X.L$, WSDL description and grounding (Eq. 3, 4). In addition,

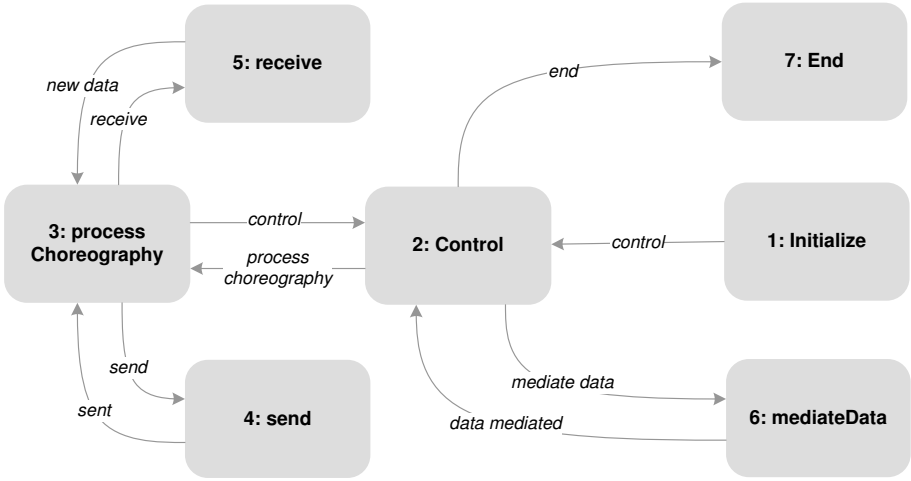

Figure 2. Control State Diagram for the Execution Phase

for a rule $r \in W.X.L$, the condition r^{cond} is a logical expression with set of semantic descriptions $\{c\}$, and the effect r^{eff} is a logical expression with set of actions $\{a\}$. For each element a we denote its action name as $a.action$ with values *delete* or *add* and a semantic description as $a.c$.
- Mappings Φ_{12} of $W_1.O$ to $W_2.O$ and mappings Φ_{21} of $W_2.O$ to $W_1.O$.

Uses:
- Symbols M_1 and M_2 corresponding to the processing memory of the choreography $W_1.X$ and $W_2.X$ respectively (a memory M is a populated ontology $W.O$ with instance data). The content of each memory M determines at some point in time a state in which a choreography $W.X$ is. In addition, each memory has methods $M.add$ and $M.remove$ allowing to add or remove data to/from M and a flag $M.modified$ indicating whether the memory was modified. The flag $M.modified$ is set to *true* whenever the method $M.add$ or $M.remove$ is used.
- Symbols D_1 and D_2 corresponding to the set of data to be added to the memory M_1 and M_2 after one or more rules of a choreography are processed. Each D has a method $D.add$ for adding new data to the set.
- A symbol A corresponding to all actions to be executed while processing the choreography. Each element of A has the same definition as the element of the rule effect r^{eff}. A has methods $A.add$ and $A.remove$ for adding and removing actions to/from the set.
- A symbol o corresponding to a WSDL operation of a service and symbols m, n corresponding to some XML data of the message (input or output) of the operation o.

States 1, 2, 7: Initialize, Control, End

1: $M_1 \leftarrow \emptyset$; $M_2 \leftarrow \emptyset$
2: **repeat**
3: $M_1.modified \leftarrow false$; $M_2.modified \leftarrow false$
4: $D_1 \leftarrow processChoreography(W_1, M_1)$
5: $D_2 \leftarrow processChoreography(W_2, M_2)$
6: **if** $D_1 \neq \emptyset$ **then**
7: $D_m \leftarrow mediateData(D_1, W_1.O, W_2.O, \Phi_{12})$
8: $M_1.add(D_1)$; $M_2.add(D_m)$
9: **end if**
10: **if** $D_2 \neq \emptyset$ **then**
11: $D_m \leftarrow mediateData(D_2, W_2.O, W_1.O, \Phi_{21})$
12: $M_1.add(D_m)$; $M_2.add(D_2)$
13: **end if**
14: **until not** $M_1.modified$ **and not** $M_2.modified$

After the initialization of the processing memory M_1 and M_2 (line 1), the execution gets to the control state when the algorithm can process choreographies (state 3), mediate the data (state 6) or end the execution (state 7). The execution ends when no modifications of the processing memories M_1 or M_2 has occurred.

State 3: $D = processChoreography(W, M)$

1: $A \leftarrow \emptyset$; $D \leftarrow \emptyset$
2: {*Performing rule's conditions and sending data*}
3: **for all** r in $W.X.L : holds(r^{cond}, M)$ **do**
4: $A.add(r^{eff})$
5: **for all** c in $r^{cond} : c \in W.X.\Sigma_I$ **do**
6: $send(c, W)$
7: **end for**
8: **end for**
9: {*Performing delete actions*}
10: **for all** a in $A : a.action = delete$ **do**
11: $M.remove(a.c)$
12: $A.remove(a)$
13: **end for**
14: {*Receiving data and performing add actions*}
15: **while** $A \neq \emptyset$ **do**
16: $c \leftarrow receive(W)$
17: **if** $c \neq null$ **then**
18: **for all** a in A: ($a.action = add$ and $a.c = c$) **do**
19: $D.add(c)$
20: $A.remove(a)$
21: **end for**
22: **end if**

23: **end while**

24: **return** D

The algorithm executes each rule of the choreography which condition holds in the memory by processing its condition and effect, i.e. the algorithm collects all data to be added to the memory or removes existing data from the memory in three major steps as follows.

- **Performing rule's conditions and sending data (lines 2-8):** the algorithm adds the effect of the rule which condition holds in the memory (line 3) to the set of effects A (line 4). Then, for each input symbol of the rule's condition (line 5), the algorithm sends the data to the service (line 6, see State 4).
- **Performing delete actions (lines 9-13):** the algorithm processes all effects with *delete* action, removes the data of the effect from the memory (line 11) as well as from A (line 12).
- **Receiving data and performing add actions (lines 14-24):** when there are effects to be processed in A and the new data is received from the service (line 16), the algorithm checks if the new data corresponds to some of the *add* effect from A. In this case, it adds the data to the set D (line 19) and removes the effect from A (line 20).

The result of the algorithm is the set D which contains all new data to be added to the memory M. The actual modification of the memory M with the new data is done in State 2. During the choreography processing, the algorithm relies on a consistent definition of the reference grounding (see Eq. 3), i.e. choreography rules are consistent with WSDL operations and their MEPs, as well as assumes no failures occur in services. In lines 14-23 the algorithm waits for every message from the service for every *add* action of the rule's effect. If the definition of the rules was not consistent with WSDL description, the algorithm would either ignore the received message which could in turn affect the correct processing of the choreography (in case of missing *add* action) or wait infinitely (in case of extra *add* action or a failure in a service). For the latter, the simplest solution would be to introduce a timeout in the loop (lines 14-23), however, we do not currently handle this situation.

State 4: $send(c, W)$

1: $m \leftarrow lower(c)$
2: **for all** o of which m is the input message **do**
3: send m to W
4: **end for**

In order to send the data c the algorithm first retrieves a corresponding message definition according to the grounding and transforms c to the message m using the lowering transformation function (line 1). Then, through each operation of which the message m is the input message, the algorithm sends the m to the service W.

State 5: $c = receive(W)$

1: **if** receive m from W **then**
2: $c \leftarrow lift(m)$
3: **return** c
4: **else**
5: **return** *null*
6: **end if**

When there is new data from the service W, the algorithm lifts the data (message m in XML) to the semantic representation using lifting transformation function associated with the message (line 2).

State 6: $D_m = mediateData(D, O_s, O_t, \Phi)$

1: $\rho \leftarrow \emptyset$; $\xi_m \leftarrow \emptyset$
2: **for all** $c \in D$ **do**
3: $\varepsilon \leftarrow getTypeOf(c)$;
4: $\varepsilon_m \leftarrow null$
5: **for all** $m =< \varepsilon_s, \varepsilon_t, \gamma_{\varepsilon_s}, \gamma_{\varepsilon_t} > \in \Phi$ **do**
6: **if** $\varepsilon = \varepsilon_s$ **then**
7: **if** $isBetterFit(\varepsilon_t, \varepsilon_m)$ **then**
8: $\varepsilon_m \leftarrow \varepsilon_t$
9: **end if**
10: $m_G \leftarrow ground(m)$; $\rho \leftarrow \rho \cup \{m_G\}$
11: **end if**
12: **end for**
13: $\xi_m \leftarrow \xi_m \cup \varepsilon_m$
14: **end for**
15: **if** $\xi_m = null$ **then**
16: **return** *null*
17: **end if**
18: $D_m \leftarrow getDataForType(\xi_m, \rho)$
19: **return** D_m

The algorithm performs two steps during data mediation. Firstly, the algorithm processes mappings in order to determine the most suitable target concepts to mediate the source data to, and secondly, the algorithm transforms the mappings into an executable form and executes the mappings. Since current reasoning engines does not scale well in terms of processing time, keeping these steps separate enable high-performance in processing of alignments independent on the logical language and reasoning engine used. In other words, this approach minimizes the use of the reasoning during the data mediation.

– **Step 1**: The algorithm first determines a concept for an instance data to be mediated (line 3). After that, the algorithm traverses through a set of mappings in order to determine the type of the target data (mediated data) (lines 5-12). Since there could be more mappings from a given source entity to the several other target entities, the algorithm determines the most suitable

concept (lines 7-9). In particular, if a concept ε_s is mapped to two target concepts ε_t^1 and ε_t^2, then ε_t^1 is more suitable if ε_t^1 is a sub-concept of ε_t^2 (the most specific) or if ε_t^2 can be reached via binary relationships (i.e. attributes) starting from ε_t^1 (maximal coverage).
- **Step 2:** While traversing the set of mappings, the algorithm grounds each mapping to a logical language by transforming them to a set of logical mapping rules (line 10). Finally, by using a reasoner engine, the algorithm queries and retrieves all the data of the selected target type according to the source data and the set of mapping rules (line 18).

3.2 Discussion

Data mediation ensures that all new data coming from one service is translated to the other's service ontology. Thus, no matter from where the data originates the data is always ready to use for the both services. From the process mediation point view, the data mediation also handles the splitting of messages (pattern c) and combining messages (pattern d). Since the mediated data is always added to the both memories (see State 2, lines 8, 12 and the next paragraph for additional discussion) the patterns (a) and (b) are handled automatically through processing of the choreography rules. In particular, the fact that a message will be stopped (pattern a) means that the message will never be used by the choreography because no rule will use it. In addition, the order of messages will be inverted (pattern b) as defined by the choreography rules and the order of ASM states in which conditions of rules hold. This means that the algorithm automatically handles the process mediation with help of data mediation through rich description of choreographies when no central workflow is necessary for that purpose. In order to fulfill the pattern (e), the algorithm might need a third-party data for which an integration workflow might be neccessary. Although some of the third-party data can be gathered through transformation functions of the data mediation which can in turn facilitate some cases of pattern (e), we do not provide a general solution for this pattern. A special case of pattern (e) could be "generating an acknowledgement message" for which the algorithm should distinguish types of interactions. For example, if the algorithm is able to understand control interactions (such as acknowledgements) among all the interactions between services, it could generate an acknowledgment message (evaluation of successfull reception of the message by the other service is, however, another issue).

4 Implementation

In this section we describe the implementation of the execution model using the WSMO, WSML and WSMX technology on the use case from the SWS Challenge as Figure 1 depicts. We use WSMO to model ontologies, services, mediators and goals according to the scenario. WSMO uses WSML family of ontology languages to define concrete semantics of these elements. In addition, WSMX is the execution environment for WSMO allowing to run the execution of WSMO services. In the core of the solution, the WSMX middleware is located between

Blue and Moon systems. WSMX functionality can be customized to conform to particular integration needs through choosing appropriate components and their configuration. For our solution, we use the *orchestration* which executes the conversation and the *data mediation* which resolves the heterogeneity issues, both operating according to the execution model. In addition, WSMX contains the base components such as *reasoning* which performs logical reasoning over semantic descriptions and *communication* or *persistence*. For brevity, we do not show them in the figure.

We use WSMO ontology to model the information semantics of services, and the *choreography interface* definition of WSMO Service/Goal to model the behavioral semantics of services. In addition, we need to create grounding to underlying WSDL and XML Schema (The SWS Challenge provides all services in WSDL together with endpoints accessible via SOAP over HTTP) and mapping rules between ontologies. Firstly, we create ontologies in WSML language as semantic representations of the PIP3A4, CMR and OMS XML schemata. Secondly, we create lifting and lowering transformations in XSLT between underlying XML and the ontologies. Finaly, we define mappings between the both ontologies. Although we could define one overarching ontology for the both XML schema together with lifting/lowering transformations to this ontology, we want to demonstrate the use of mappings and data mediation. Hence we define two heterogeneous ontologies for the two heterogeneous XML schema.

```
axiom aaMappingRule23
    definedBy
        mediated(?X21, SearchCustomerReq)[searchString hasValue ?Y22] memberOf o1#
            SearchCustomerReq
:- ?X21[businessName hasValue ?Y22] memberOf o2#BusinessDescription.
```

Listing 1.1. Mapping Rules in WSML

Listing 1.1 shows a sample mapping rule between the *SearchCustomerReq* concept of the CMR ontology (denoted using $o1$ prefix) and *BusinessDescription* concept of the PIP3A4 ontology (denoted using $o2$ prefix). The construct $mediated(X, C)$ represents the identifier of the newly created target instance, where X is the source instance that is transformed, and C is the target concept we map to.

In line with Eq. 2, the WSMO service choreography contains the definition of the input, output and shared symbols (called state signature or vocabulary) and a set of rules. Using these rules we model the choreography of the both PIP3A4 and CRM/OMS services separately and for each define the order in which the operations should be correctly invoked. Listing 1.2 shows a fragment of the choreography for the CRM/OMS service. There are two rules defined. The first rule (lines 17-22) defines that the *SearchCustomerReq* will be sent to the service and on result the *SearchCustomerResp* will be expected as the output message. The *SearchCustomerReq* message must be available in the memory (in our case the data for the message is provided by the Blue after the data mediation). The second rule (lines 24-30) defines that the *SearchCustomerResp* must be available in the memory while its *customerId* will be used for the *customerId* of the *CreateNewOrderReq* which will be sent to the service. On result, the *CreateNewOrderResp* will be expected to be received back. The data for the

CreateNewOrderReq will be again supplied by the Blue after the data mediation. All the messages used in the choreography as the input or output symbols refer to the definition of concepts in the ontology imported in line 3 while at the same time the grounding of these concepts to the underlying WSDL messages is defined in lines 5-14.

```
1  choreography MoonWSChoreography
2   stateSignature _"http://example.com/ontologies/MoonWS#statesignature"
3    importsOntology {_"http://example.com/wsml/Moon" }
4    // input symbols
5    in moon#SearchCustomerReq
6         withGrounding { _"http://example.com/MoonCRM#wsdl.interfaceMessageReference(search/in0)" }
7        moon#CreateNewOrderReq
8         withGrounding { _"http://example.com/MoonOMS#wsdl.interfaceMessageReference(openorder/
                in0)" }
9
10      // output symbols
11      out moon#SearchCustomerResp
12          withGrounding { _"http://example.com/MoonCRM#wsdl.interfaceMessageReference(search/out0)
                "}
13         moon#CreateNewOrderResp
14          withGrounding { _"http://example.com/MoonOMS#wsdl.interfaceMessageReference(openorder/
                out0)" }
15      ...
16   transitionRules _"http://example.com/ontologies/MoonWS#transitionRules"
17    // rule 1: search the customer in CRM
18    forall {?customerReq} with (
19     ?customerReq memberOf moon#SearchCustomerReq
20    ) do
21     add(_# memberOf moon#SearchCustomerResp)
22    endForall
23
24    // rule 2: open the order in OMS
25    forall {?orderReq, ?customerResp} with (
26     ?customerResp[customerId hasValue ?id] memberOf moon#SearchCustomerResp and
27     ?orderReq[customerId hasValue ?id] memberOf moon#CreateNewOrderReq
28    ) do
29     add(_# memberOf moon#CreateNewOrderResp)
30    endForall
```

Listing 1.2. Moon CRM/OMS Choreography

5 Related Work

The most relevant related work is among other submissions addressing the SWS-Challenge mediation scenario, namely WebML [10] and dynamic process binding for BPEL[8]. They are based on software engineering methods focusing on modelling of integration process as a central point of integration. They do not use logical languages in their data model. In addition, Preist et al [12] presented a solution covering all phases of a B2B integration life-cycle, starting from discovering potential partners to performing integrations including mediations. Their solutions is rather conceptual with missing details about the actual components and algorithms used. More general SWS related work include IRS-III[4] which is an execution environment also based on WSMO. In addition, there are related works that apply ASM for various stages of service integration process. Altenhofen et al [1] also address Process Mediation of services modelled using ASM, however they focus only on Process Mediation aspect while we provide a

complete conceptual model and implementation addressing both Data and Process Mediation. In [6] a composition algorithm based on Web service process ASMs is described where formal, mathematical model of Web services and orchestrations is provided. Their model of ASMs varies from our ontologized ASMs model for example with respect to modelling incoming and outgoing messages. In our model it is implicitly inferred from concept grounding in choreography in/out states whether ASM Knowledge Base modifications entail communication with the service as opposed to ASM used in [6] explicitly models communication constructs. The purpose of describing Web service public processes in [6] is different – it is primarily focused on composition of ASM services, while we utilize ASMs for achieving Process Mediation between communicating Web services. Lerner [9] focuses on analysis and verification of parameterized State Machines applied to reusable processes specified in Little-JIL language. Provided algorithm is able to detect deadlock and other anomalies of analyzed processes. Underlying language is based on State Machines similarly like in our case. We might consider in the future to focus more on the ASM process analysis and verification using similar methods as proposed by Lerner. Benatallah et al [2] present a conceptual model for Web service protocol specifications. They provide framework supporting analysis of commonalities and differences between protocols supported by different Web services. Similarly, like in case of Lemcke [6] most of the focus was given to public process analysis (called Web service protocol by Benatallah) while in our work we presented models, mediation algorithms and working implementation addressing both data and public process heterogeneity issues.

6 Conclusion and Future Work

One of the main advantages of our approach is the strong partner de-coupling. This means that when changes occur in back-end systems of one partner, consequent changes in service descriptions does not affect changes in the integration. The integration automatically adapts to the changes in service descriptions as there is no central integration workflow. On the other hand, changes in back-end system still require manual effort in making changes in semantic descriptions such as ontologies and mapping rules. Although our SWS technology allows for semi-automated approaches in modelling and mapping definitions, it is still a human user who must adjust and approve the results. It is important to note, however, that this type of integration where no central workflow is necessary is only usable in situations when two public processes (ASM choreographies) are compatible, that is, they may have different order/structure of messages but by adjusting the order/structure the integration is possible. In general, there could be cases where third-party data need to be obtained (e.g. from external databases) for some interactions. Although some of the third-party data can be gathered through the transformation functions of the mapping rules, in some cases, an external workflow could be required to accommodate the integration process. It is our open research work to further investigate such cases in detail.

Acknowledgments

This work is supported by the Science Foundation Ireland Grant No. SFI/02/CE1/I131, and the EU projects SUPER (FP6-026850), and SemanticGov (FP-027517).

References

1. M. Altenhofen, E. Börger, and J. Lemcke. An abstract model for process mediation. In *ICFEM*, pp. 81–95. 2005.
2. B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data Knowl. Eng.*, 58(3):327–357, 2006.
3. E. Cimpian and A. Mocan. Wsmx process mediation based on choreographies. In *Business Process Management Workshops*, pp. 130–143. 2005.
4. J. Domingue, *et al.* Irs-iii: A broker-based approach to semantic web services. *J. Web Sem.*, 6(2):109–132, 2008.
5. J. Euzenat, *et al.* Results of the Ontology Alignment Evaluation Initiative 2006. In *Proceeding of International Workshop on Ontology Matching (OM-2006)*, vol. 225, pp. 73–95. CEUR Workshop Proceedings, Athens, Georgia, USA, November 2006.
6. A. Friesen and J. Lemcke. Composing web-service-like abstract state machines (asm). In *Autonomous and Adaptive Web Services*. 2007.
7. T. Haselwanter, *et al.* WSMX: A Semantic Service Oriented Middleware for B2B Integration. In *ICSOC*, pp. 477–483. 2006.
8. U. Kuster and B. Konig-Ries. Dynamic binding for bpel processes - a lightweight approach to integrate semantics into web services. In *Second International Workshop on Engineering Service-Oriented Applications: Design and Composition (WESOA06) at 4th International Conference on Service Oriented Computing (ICSOC06)*, pp. 00–00. Chicago, Illinois, USA, December 2006.
9. B. S. Lerner. Verifying process models built using parameterized state machines. In *ISSTA*, pp. 274–284. 2004.
10. T. Margaria, *et al.* The sws mediator with webml/webratio and jabc/jeti: A comparison. In *ICEIS (4)*, pp. 422–429. 2007.
11. A. Mocan and E. Cimpian. An Ontology-Based Data Mediation Framework for Semantic Environments. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 3(2), April - June 2007.
12. C. Preist, *et al.* Automated Business-to-Business Integration of a Logistics Supply Chain using Semantic Web Services Technology. In *Proc. of 4th Int. Semantic Web Conference*. 2005.
13. D. Roman, *et al.* Web Service Modeling Ontology. *Applied Ontologies*, 1(1):77 – 106, 2005.
14. T. Vitvar, J. Kopecky, and D. Fensel. WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web. In *ECOWS*. 2007.
15. T. Vitvar, *et al.* Semantically-enabled service oriented architecture : concepts, technology and application. *Service Oriented Computing and Applications*, 2(2):129–154, 2007.