Using UML to model web services for automatic composition

El Gammal, A.; El-Sharkawi, M.

Published in:
International Journal of Software Engineering

Publication date:
2010

Document Version
Early version, also known as pre-print

Using UML to Model Web Services for Automatic Composition

Amal Elgammal(1) and Mohamed El-Sharkawi (2)

(1) Department of Information Systems, Faculty of Computers and Information. Cairo University (Egypt)
E-mail: a.f.s.a.elgammal@uvt.nl

(2) Department of Information Systems, Faculty of Computers and Information. Cairo University (Egypt)
E-mail: m.elsharkawi@fci-cu.edu.eg

ABSTRACT

There is a great interest paid to the web services paradigm nowadays. One of the most important problems related to the web service paradigm is the automatic composition of web services. Several frameworks have been proposed to achieve this novel goal. The most recent and richest framework (model) is the Colombo model. However, even for experienced developers, working with Colombo formalisms is low-level, very complex and time-consuming. We propose to use UML (Unified Modeling Language) to model services and service composition in Colombo. By using UML, the web service developer will deal with the high level graphical models of UML avoiding the difficulties of working with the low-level and complex details of Colombo. To be able to use Colombo automatic composition algorithm, we propose to represent Colombo by a set of related XML document types that can be a base for a Colombo language. Moreover, we propose the transformation rules between UML and Colombo proposed XML documents. Next Colombo automatic composition algorithm can be applied to build a composite service that satisfies a given user request. A prototypical implementation of the proposed approach is developed using Visual Paradigm for UML.

Keywords: Automatic web services composition, Colombo composition model, Web services Composition, Web service description, Unified Modeling Language.

1- INTRODUCTION

There is a great interest and support paid to the web services paradigm nowadays. The web services paradigm allows rich, flexible, and dynamic interoperation of highly distributed and heterogeneous web-hosted services. A web service is simply a software that describes a collection of operations via an interface that are network-accessible through standardized XML protocols [2]. The three main standards that represent the backbone of web services are: SOAP (Simple Object Access Protocol, the XML standard for message exchange), WSDL (Web Service Description Language, the XML standard for describing web services) [3], [4] and UDDI (Universal Description, Discovery and Integration of web Services, the registry where services description can be published) [5].

The web service paradigm is a promising paradigm; however, it has a mine of problems that need to be solved. The automatic web service composition problem has gained a great support from both the industry and academia. Web service composition is the process of combining different web services to provide a value-added service. In other words, web services composition represents the situation when the client request can not be satisfied by an individual available web service, however combining and coordinating a set of available web services can fulfill her needs.

Composition involves two different issues [1] which are: *Composition Synthesis* and *Composition Orchestration*. Composition synthesis concerns itself with synthesizing a specification of how component services can cooperate and coordinate with each other in order to fulfill user’s request. While composition orchestration concerns itself with how to actually achieve the coordination among services, by executing the specification produced by the composition synthesis and by supervising and monitoring both the control flow and data flow among the participating services. Orchestration has been intensively studied by other research areas especially the research on workflows. Hence, the main focus of this research is on the composition Synthesis.

Performing the composition process manually is a difficult task even for very simple compositions. Automation of this process is considered one of the most important challenges that face the web services paradigm. To achieve this novel goal, available web services should be described following a rich framework. WSDL is not sufficient to allow the automation of the web services composition, because it just provides information about the input/output signature (the message types that the web service can send and receive). Richer descriptions are needed, such as semantic information and the behavioral descriptions of web services. Four main models have been proposed to solve this problem, which are: (i) OWL-S model [6], (ii) Roman model[7], [8], (iii) Mealy/conversation model [9], and (iii) Colombo model [1], [10]. Colombo Model is the most recent and richest model, as it combines the important aspects of the other three models and unifies them in a single framework.

We can summarize the contribution of this paper as follows:

1. Colombo model adapts formalisms that are complex and time consuming even for experienced developers and modelers. We propose to use UML to model web services in Colombo. Subsequently, the web service developer will deal with the high level, graphical, and easy models of UML which will significantly facilitate her work.
2. Colombo is a conceptual model, which means that it does not have an associated supporting language. We propose a set of related XML document types that can be a base of a Colombo language.

3. Furthermore, we propose the transformation rules between UML and XML, which allows the utilization of the Colombo composition algorithm. Weassume that the composition algorithm can accept and produce XML documents.
4. An implementation of this model is developed by creating a prototype of a CASE tool that would be used to facilitate the design of composite web services using UML. Figure 1 presents a block diagram of our proposal

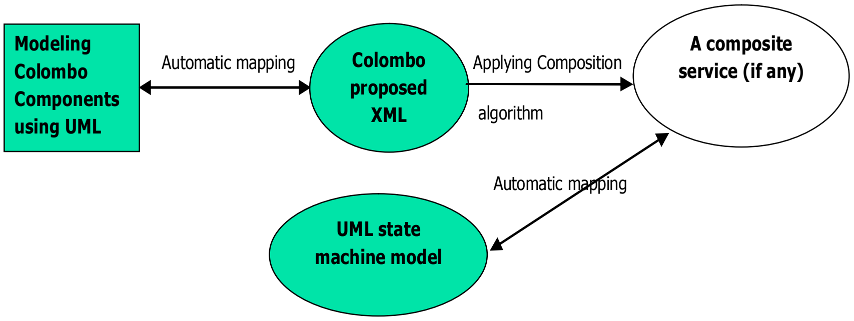


Figure 1 Proposal overview

Figure 1 presents the proposed modeling architecture. The process starts with the web service developer modeling Colombo components using UML. Next, an automatic mapping is performed to transform UML diagrams to a set of XML documents.Then, these XML documents are sent as input to the Colombo composition algorithm proposed in [1], assuming that Colombo composition algorithm can accept and produce XML documents. Finally, the composition algorithm will produce a composite service as its output. The resulting composite service is then automatically transformed to a UML diagram to facilitate the investigation of the resultant composite web service by the web service developer.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 briefly illustrates how web services are characterized in Colombo. Section 4 presents our proposal to model Colombo web services using UML. Section 5 covers the proposed XML document types to represent Colombo. Section 6 presents a prototypical implementation of the proposed approach. Finally, Section 6 concludes the paper.

.

2- RELATED WORK

It is interesting here to mention studies that use UML in various stages of the web services development as a motivation for our work. The study in [11] used UML for modeling and development of Service Oriented Architecture (SOA). They used Model-Driven Architecture (MDA) as it conceives models as first class elements during system design and implementation and establishes a separation of development process into three abstraction levels, which are: CIM (Computational Independed Model), PIM (Platform Independent Model), and PSM (Platform Specific Model). They maintained the mappings between the various levels, which in turn makes it possible to automate the entire web services development process. They have proposed a PIM level UML profile, besides its corresponding meta-model for the modeling and development of SOA. The PIM has been chosen because it does not reflect any constraints about any specific platform or implementation technology.

Authors in [12] have used UML to define the WSDL descriptions of web services. They advocate the usage of WSDL-independent models versus WSDL-dependent models. WSDL-independent models means using pure UML constructs without introducing any WSDL-specific stereotypes. They is basically done because pure UML can improve the understanding of web services, especially when modeling complex web services. The conversation rules between UML and WSDL have also been introduced and embedded in UML transformation tool (UMT) [13]. Similar to this work is the work in [14], which presents a platform-independent service and workflow modeling. However the transformation rules to a specific platform has not been defined.

On the other hand, several studies have proposed WSDL-dependent UML profiles. The study in [15] has proposed UML profile for WSDL through the introduction of WSDL-specific stereotypes. In [16] a Hypermodel tool has been propoed such that XML schema that is a part of the WSDL document can be imported into UML. But the resulting UML model will have XML schema specific stereotypes. We have to point out that web service compositions based on WSDL descriptions can't be automated because WSDL lacks any semantic or behavioral descriptions.

The study in [17] has proposed to model web services composition with (agent) UML 2.0, which provides some agent-speciifc extensions to UML. They have considered WSDL web services. WSDL descriptions lacks any semantic or behavioral descriptions, hence web services composition can not be automated. They started with a high-level and global description of web services interactions in an implementation independent way, which is known as *web Service Choreography.* Next, these high level descriptions have to be mapped to a low-level SOAP messages between different interacting web services, which is known as *web service orchestration*.Business Process Execution Language (BPEL) [18] is the defacto standard for orchestrating web services and is considered in this work. UML has been used to model WSDL and BPEL descripttions. Furthermore, a mapping has been introduced between the proposed UML notations and WSDL , as well as between UML and BPEL. This study considers the manual composition of web services that opposes with the main focus of this paper, where automatic composition is its main novel goal.

The study in [19] proposed a model-driven methodology for semantically described automatic web services composition. A UML profile has been introduced to represent OWL-S web services. The transformation rules between UML and OWL-S have also been proposed to show that the proposed UML profile is expressive enough to support one of the leading semantic web service languages. In [20] a graphical tool has been developed as an OWL-S editor, such that OWL-S documents can be imported and exported. In [21], a semantic web-independent graphical language has been presented. The proprietary ODE SWS graphical language has been used for modeling tasks that can be associated with inputs, outputs, pre-conditions and post-conditions. However, the transformation rules have not been introduced.

The study in [22] has presented an approach where a user can annotate operations and its parameters with pre-conditions, post-conditions and semantic types in a tree view browser. This tool supports both OWL-S and WSDL-S (extends WSDL 2.0 with semantic descriptions) which provided a semantic language-independent approach.

To the best of our knowledge, no work has been done for modeling the Roman model [8] or Mealy/Conversation [9] model in UML. In this paper, we have considered Colombo model as unifies OWL-S, Roman and Mealy/Conversation models in a single rich framework. Consequently, we can claim that our work is more generic than others. In the following Section we will briefly illustrate how web services are modeled in Colombo to enable their automatic composition.

3- COLOMBO MODEL

In Colombo [1], a web service is characterized in terms of the following four components:

a. *A world state*: representing the real world, viewed as a database instance over a relational database schema, referred to as world schema.

b. *Atomic processes*: represent operations or functionalities that can be performed by different web services. Atomic processes can access and modify the world state, and can include conditional effects. Atomic processes represent a common understanding of an agreed upon reference alphabet/semantics. Atomic processes represent the community ontology, such that web services, clients or any other participant of this community should share this ontology.

c. *Message passing behavior*: these are messages (message types) that can be sent or received by a web service. Here we are concerned with the message types (classes) instead of the message contents. For example, a message named ‘requestPurchase’ may hold two variables of type String, which are ‘code’ and ‘PayBy’, which represent respectively the code of the item the client would like to purchase, and the payment method.

d. *The behavior of the web service*: the behavior may include multiple atomic processes and message passing activities. It is specified in Colombo using guarded automata. A guarded automaton is a finite state machine (FSM), such that conditions can be specified between transitions. A transition from one state to the next will take place only when the transition condition is evaluated to true.

Colombo classifies web services into four types, each of them can be modeled in the same way, using guarded automata:

a. *Non-Client Web Services*: these are real atomic web services that perform different functionalities and their descriptions are published in the UDDI [5] registry.

b. *Client Web Services*: A client web service represents client behavior. Since the client interacts with a web service by repeatedly sending and receiving messages, her behavior can be modeled the same way as non-client web service. But since client's behavior is non-deterministic in terms of the actions she makes and the choices she selects, her guarded automata will contain only two states, which are: ReadyToTransmit and ReadyToRecieve. The client will toggle between these two states until she terminates.

c. *Goal Service*: This represents the desired behavior to be achieved. It is also specified as a guarded automaton in terms of Alphabet of atomic processes *A.*

d. *Mediator Service*: Colombo adapts a mediated topological approach for composition. In the mediated approach a virtual service named as the 'mediator' is responsible for controlling data flow and control flow among participating services. The behavior of the mediator service should simulate the behavior of the goal service. The mediator service represents the composition synthesis specification which should be orchestrated to fulfill client request, it represents the expected output from the Colombo automatic composition algorithm. For an overview of the different web services composition topological approaches refer to [2].

It is assumed that each non-client and mediator web service instance has:

a. *A local store (LStore)*: can be implemented as a relational database table, which is used to store parameter values of incoming messages and output values of atomic processes, and to populate parameters of outgoing messages and input parameters to atomic processes. The conditional branching of web services behavior at any time is based on the values stored in its local store at this time.

b. *A port* for each incoming or outgoing message to allow communication between web services.

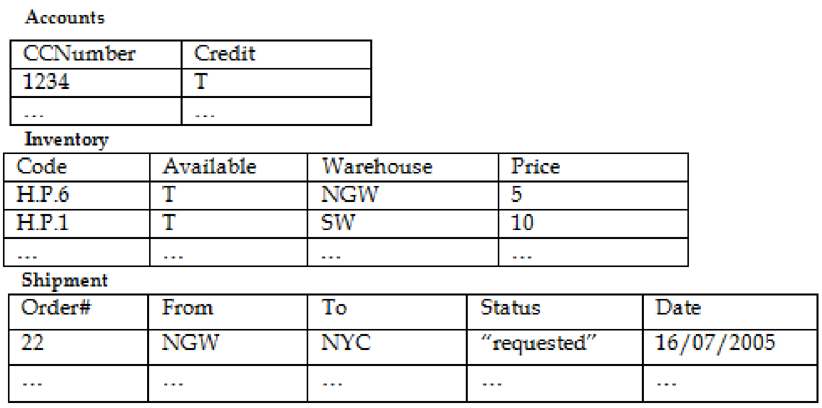
c. *A queue store* (*QStore*): for each incoming message. *QStore* is of bounded length to be equal to one

The client web service does not have either a local store or queue store. A unary relation is maintained for the client service includes the parameter values of the messages the client has received so far. The work in [1] has introduced a complete and sound automatic composition algorithm, and also has determined the algorithm decidability and complexity under some restrictions.

3-1 RUNNING EXAMPLE

The running example used throughout the paper is borrowed from [1]. Assuming that an application developer needs to locate a web service to perform the following functionalities: (i) order an item, (ii) such that payment can be made by credit card, (iii) request shipment, and (iv) check shipment status at any time. Figure 2 presents the world state I relevant to this example. In Figure 2, 'Accounts' table checks the validity of a given credit card number, 'Inventory' table contains the codes of the items, whether it is available or not, warehouse name where the item is available and its price. The 'Shipment' table includes order numbers, from and to the item will be shipped, the status of the shipment and the date of the order.

Figure 3 presents the alphabet A of atomic processes. The f symbol represents the access function. E.g. $f_1^{Accounts}(C)$ function accesses the Accounts relation in the world state I and returns the value of the 1st attribute after the key of the tuple having a primary key equal to the parameter value 'C'. The null value is represented using 'ω' symbol. And '-'sign indicates that values remain unchanged after modifications.



Accounts

CCNumber	Credit
1234	T
...	...

Inventory

Code	Available	Warehouse	Price
H.P.6	T	NGW	5
H.P.1	T	SW	10
...	...	...	...

Shipment

Order#	From	To	Status	Date
22	NGW	NYC	"requested"	16/07/2005
...	...	...	...	...

Figure 2 World Schema Instance I [1]

Figure 4 presents the available web services represented as guarded automata. The functionalities of these web services are: (i) Bank: Checks the validity of a credit card number, (ii) Storefront: Given the code of an item returns its price and the warehouse in which the item is available, and (iii) Standard Warehouse (SW): Deals only with orders by credit card, and allows for shipping the ordered item if the card is valid and the client can check shipment status at any time. For the transitions between states, if it starts with '?' means the receipt of a message, if it starts with '!' means the send of a message, otherwise means an atomic process invocation.

```
**CCCheck**
        I: C: Dom=; % CC card number
        O: app: Bool; %CC approval
        effects:
                if $f_1^{Accounts}(C)$ then
                    either modify Accounts(C;T) or
                    modify Accounts(C;F) and approved:= T
            else
                approved: = F
**CheckItem**
        I: C: Dom=; % item code
        O: avail : Bool; wh: Dom=; p:Dom< % resp. item availability, selling warehouse and price
        effects:
                if $f_1^{Inventory}(c)$ then
                    avail:=T and wh:= $f_2^{Inventory}(c)$ and p:= $f_3^{Inventory}(c)$
                    and either no-op on Inventory or modify Inventory (c;F, -, -)
                if $\neg f_1^{Inventory}(c)$ or $f_1^{Inventory}(c) = \omega$ then
                    avial:=F
**requestShip**
        I: wh: Dom=; addr: Dom=; %resp. source warehouse and target address
        O: oid: Dom=; d: Dom<; s: Dom=; %resp. order id, shipping date and status
        effects:
        $\exists$ d, o oid: =new(o) and insert Shipment(Oid, wh, addr, "requested, d") and d =
        and s = "requested"
**CheckShipStatus**
        I: oid: Dom=; % order id
        O: s: Dom=; d: Dom<; % resp. shipping date & status
        effects:
                if $f_1^{Shipment}(oid) = \omega$ then no-op and s, d uninit
                else s: $f_3^{Shipment}(oid)$ and d: = $f_4^{Shipment}(oid)$
```

Figure 3 Alphabet of atomic processes [1]

Figure 5 presents the guarded automata of the goal service specified in terms of alphabet of atomic processes A. After applying the automatic composition algorithm proposed in [1], a mediator service will be generated. For more details about the guarded automata of the mediator service of this example and how it is produced refer to [1]. In the next Section we will show how Colombo web services can be modeled using UML by introducing some extensions to UML.

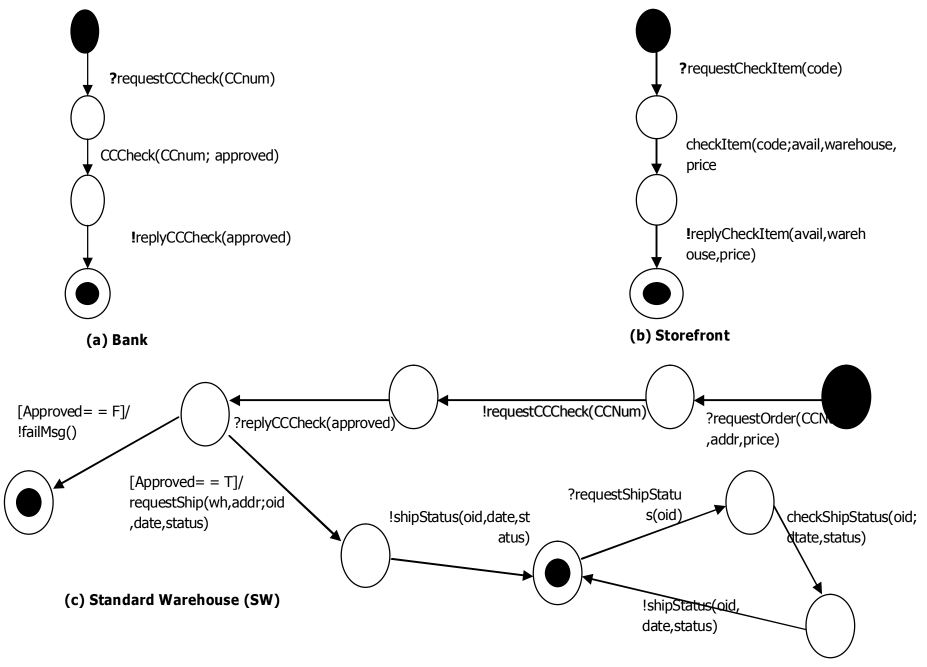

?requestCCCheck(CCnum)
CCCheck(CCnum; approved)
!replyCCCheck(approved)
(a) Bank
?requestCheckItem(code)
checkItem(code;avail,warehouse, price
!replyCheckItem(avail,warehouse,price)
(b) Storefront
[Approved= = F]/ !failMsg()
?replyCCCheck(approved)
!requestCCCheck(CCNum)
?requestOrder(CCN ,addr,price)
[Approved= = T]/ requestShip(wh,addr;oid ,date,status)
!shipStatus(oid,date,status)
?requestShipStatus(oid)
checkShipStatus(oid; dtate,status)
!shipStatus(oid, date,status)
(c) Standard Warehouse (SW)

Figure 4 Guarded automata of the available web services [1]

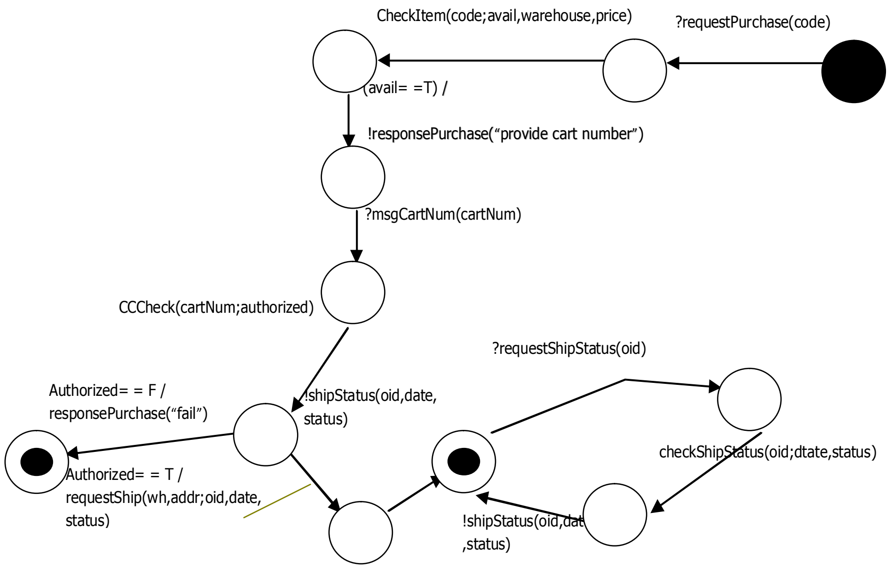

CheckItem(code;avail,warehouse,price)
?requestPurchase(code)
(avail= =T) /
!responsePurchase("provide cart number")
?msgCartNum(cartNum)
CCCheck(cartNum;authorized)
?requestShipStatus(oid)
Authorized= = F / responsePurchase("fail")
!shipStatus(oid,date, status)
checkShipStatus(oid;dtate,status)
Authorized= = T / requestShip(wh,addr;oid,date, status)
!shipStatus(oid,dat ,status)

Figure 5 Guarded automata of the goal service [1]

4- MODELING WEB SERVICES IN UML

There are three main mechanisms in UML that can be used when extending UML's base elements [23]: (i) *stereotypes:* introduce new elements by extending a base one, (ii) *tagged values:* introduce new properties for the new element, and (iii) *constraints:* are restrictions on the newly introduced stereotypes with respect to its base elements. A grouping of a set of introduced stereotypes is named as a *UML profile*. We have extended Standard UML 2.0 base elements and introduced a UML profile that can be used to model web services in Colombo. Figure 6 presents the meta-model of the proposed UML profile. Table 1 presents the tagged values of the introduced stereotypes.

WebService stereotype represents the core element of the proposed UML profile. *WebService* stereotype extends standard UML 2.0 Class construct. The attributes of the *WebService* stereotype are the message classes that the web service can send and receive. While in the operation counterpart, the operations (atomic processes) that the web service can invoke are listed. *WebService* stereotype captures the static side of a web service which describes web services properties and relationships.

Each message class is described by *MessageClass* stereotype. In Figure 6 *MessageClass* stereotype extends Standard UML 2.0 Class construct, where the message class contents can be specified in the *MessageClass* stereotype attribute counterpart.*AtomicProcess* stereotype represents the atomic processes that web services can invoke during their executions (from the alphabet of atomic processes *A*). *AtomicProcess* stereotype extends Standard UML2.0 Action construct, such that each atomic process is represented in UML as an activity diagram. In UML, each action can be triggered by input events through its *InputPin*, and after its execution, it can produce some output events from its *OutputPin*. *Input* stereotype extends Standard UML 2.0 InputPin, and *output* stereotype extends Standard UML 2.0 onputPin. The inputs and outputs of interest are messages. That is why there is a 1:1 relationship between *Input* and *MessageClass* stereotypes, and the same relation holds between *Output* and *MessageClass* Stereotype.

Input and output messages represented by *Input* and *Output* stereotypes respectively can be syntactically typed based on XML schema types, or it can be semantically typed by utilizing the UML Ontology Profile package [24]. UML Ontology Profile is a pre-existing package that can automatically import ontology concepts inside a UML environment. *OntClass* in UML Ontology Profile extends Standard UML 2.0 Class construct to represent an ontology concept.

In order to represent the effect of an atomic process, we have introduced the new stereotype *Effect* that extends *Standard UML 2.0 Constraint* base element. The relationship between *AtomicProcess* stereotype and *effect* stereotype is 1:M. The two other pieces of semantic information that might be useful for describing atomic processes, nevethless they are not included in Colombo are pre-conditions and post-conditions. In UML there already exist two built-in stereotypes that can be used to represent pre-conditions and post-conditions, which are: *Standard UML2.0 <<pre-condition>>* and *Standard UML2.0 <<post-condition>>.* The relationship between *AtomicProcess* stereotype and <<*pre-condition*>> is 1:M and the same relationship holds between *AtomicProcess* stereotype and <<*post-condition*>>*.* The logical statements that can express *Effect,* <<*pre-condition*>*,* and <<*post-condition*>> can be specified by using *Object Constraint Language (OCL).* OCL is a formal language used to express constraints. These are typically invariant conditions that must hold for the system being modeled. OCL is a part of Standard UML 2.0.

The built-in sterotypes <*pre-condition*>> and <<*post-condition*>> should be linked to an Activity (an *AtomicProcess* stereotype in our case). However it may be more readable if <<*pre-condition*>> and <<*post-condition*>> are linked directly to the related inputs and outputs. In Table 1, we have introduced a tagged value to the <<*pre-condition*>> sterotype named as 'Input' that is used to specify the input messages that this precondition affects. The same is done for <<*post-condition*>> sterotype*,* we have introduced a tagged value named as 'Output' to specify the output messages that this post-condition affects. If the <<*pre-condition*>> does not include any inputs in its logical statement, then Input tagged value is set to null. If the <<*post-condition*>> does not include any outputs in its logical statement, then Output tagged value is set to null.

The category of the *AtomicProcess* can be specified using the *Category* stereotype [19]. The *Category* stereotype extends *Standard UML2.0 Comment* base element. It has four tagged values as shown in Table 1, which are: Taxonomy, TaxonomyURI, Value, and Code that identify a category concept defined within an ontology. In Colombo, each web service can invoke more than one atomic process. Each atomic process can be invoked by more than one web service. That is why we have introduced the *WebServiceProcesses* stereotype that extends Standard UML2.0 Class construct to break down this M:N relationship (this M:N relationship is not shown in Figure 6 to avoid a cluttered diagram). *WebServiceProcesses* stereotype has two tagged values as shown in Table 1, which are: (i) *WebService* typed as *WebService*, and (ii) *AtomicProcess* typed as *AtomicProcess*.

The *WebService* stereotype, which represents the static part of web services, serves as a supertype for the various types of web services. Three stereotypes are introduced which extend *WebService* stereotype, which are: *NonClientService, GoalService,* and *ClientService,* respectively. The *MediatorService* stereotype has the same properties as *NonClientService. MediatorService* can have one or more interfaces (the same as *NonClientService)* and its instance is also characterized to have a local store and a queue store. That is why *MediatorService* stereotype extends *NonClientService* stereotype.

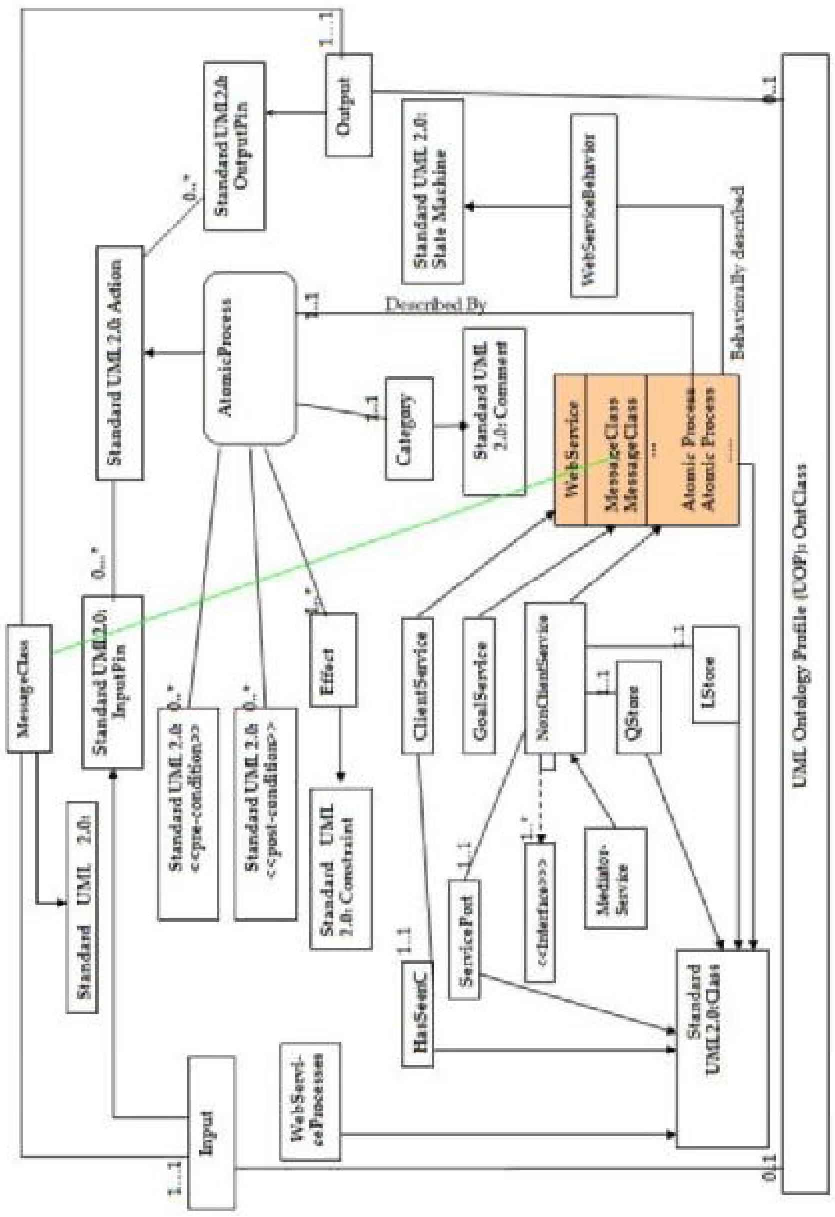


Figure 6 Meta-model of the proposed UML profile

Table 1 Proposed tagged values

Stereotype	Tagged values	Base Element
1. Atomicprocess		Standard UML2.0:Activity
2. Input		Standard UML2.0:Activity:InputPin
3. Output		Standard UML2.0:Activity:OutputPin
4. MessageClass		Standard UML2.0:Class
5. Category	Taxonomy : string TaxonomyURI : string Value: string Code: Integer	Standard UML2.0:Comment
6. Effect		Standard UML2.0: Constraint
7. <<Pre-condition>>	Input [0..*] : ***Input***	Standard UML2.0
8. <<Post-condition>>	Output [0..*] : ***Output***	Standard UML2.0
9. WebService		Standard UML2.0:Class
10. NonClientService		***WebService*** stereotype
11. GoalService		***WebService*** stereotype
12. ClientService		***WebService*** stereotype
13. MediatorService		***WebService*** stereotype
14. WebServiceBehavior		Standard UML2.0:State Machine
15. NonClientBehavior		***WebServiceBehavior*** stereotype
16. GoalBehavior		***WebServiceBehavior*** stereotype
17. ClientBehavior		***WebServiceBehavior*** stereotype
18. MediatorBehavior		***WebServiceBehavior*** stereotype
19. State		Standard UML2.0
20. NonClientState	TriggerType (send/ receive/ invoke) :String	Standard UML2.0:State
21. GoalState	TriggerType (send/ receive/ invoke) :String	Standard UML2.0:State
22. MediatorState	TriggerType (send / receive) :String To : ***WebService*** From: ***WebService***	Standard UML2.0:State
23. ClientState	Name(ReadyToTransmit/ ReadyToReacieve): String.	Standard UML 2.0 state
24. <<Interface>>		Standard UML2.0
25. WebServiceProcesses	Web Service [1..*]: ***WebService*** Atomic Process[1..*]: ***AtomicProcess***	Standard UML2.0: Class
26. LStore	Parameter[0..*]: String Value[0..*] : String InOut[0..*] (In/Out/P/flag): String	Standard UML2.0: Class
27. QStore	Parameter[0..*] : String Value[0..*] : String	Standard UML2.0: Class
28. ServicePort	Message[0..*]: message type Direction [0..*] (In/Out): String	Standard UML2.0: Class
29. HasSeenC	ConstantsC[0..*]: String Parameter[0..*]: string Value[0..*]: string	Standard UML2.0: Class

LStore and *QStore* stereotypes extend Standard UML2.0 Class construct to simulate local store and queue store of Colombo model. *LStore* has three tagged values as shown in Table 1, which are: 'Parameter', 'Value', and 'InOut'. The 'parameter' can be a message type, a parameter to/from an atomic process, or distinguished variable 'π' that indicate if there is a message in the relevant queue. 'InOut' can take either In/Out/P/Flag, such that, 'In' and 'Out' represent the direction of the message, 'P' if it is a parameter to or from an *Atomic Process*, while 'Flag' if it is the distinguished variable 'π'. *Qstore* has two tagged values, which are: 'Parameter' and 'Value'. 'Parameter' and 'Value' represent the input message type and its value.

NonClientService stereotype may have one or more interfacea (corresponding to the operations the NonClientService can perform). That is why a 1:M relationship is introduced between *NonClientService* stereotype and *Standard UML2.0<<interface>>* . *ClientService* stereotype has a 1:1 relationship with *HasSeenC* Stereotype. HasSeenC stereotype extends *Standard UML2.0 Class* and corresponds to the HasSeenC unary relation of the Colombo framework. *HasSeenC* has three tagged values, which are, which are: ‘ConstantsC’, ‘Parameter’, and ‘Value’. ‘ConstrantsC’ represents all the constants appearing in the service specification, ‘Parameter’ and ‘Value’ corresponds to the parameters and values that have been transmitted to the client. *HasSeenC* has a 1:M relationship with *Output* since all what will be transmitted to the client is output messages from the perspective of web services. Futhermore, to simulate service ports, ServicePort stereotype has been introduced that extends *Standard UML 2.0 Class* construct. *ServicePort* stereotype has two tagged values, which are: ‘Message’ and ‘Direction’. ‘Message’ is typed as *MessageClass* stereotype, and ‘Direction’ can take either the values 'In' or 'Out' representing the direction of the message with respect to the *WebService. ServicePort* has a 1:1 relationship with *NonClientService*.

The behavioral descripton of web services can be represented in UML as proposed in Figure 7 by using *WebServiceBehavior* stereotype that extends Standard UML2.0 State machine. Figure 7 presents a detailed view of the dynamic representation of web services behavior. In Standard UML2.0, each state machine diagram can have one or more Standard UML 2.0 State construct. That is why a 1:M relationship is maintained between *Standard UML 2.0 State Machine* and *Standard UML 2.0 State* construct. We have introduced four new stereotypes that extend *Standard UML 2.0 State* to represent the states of the different types of web services which are: *NonClientState*, *GoalState*, *ClientState* and *MediatorState* stereotypes. The *NonClienState* stereotype has one tagged value, which is ‘TriggerType’ that can assign the values ‘send’, ‘receive’, or ‘invoke’ representing the cause of the transition from one state to the next.

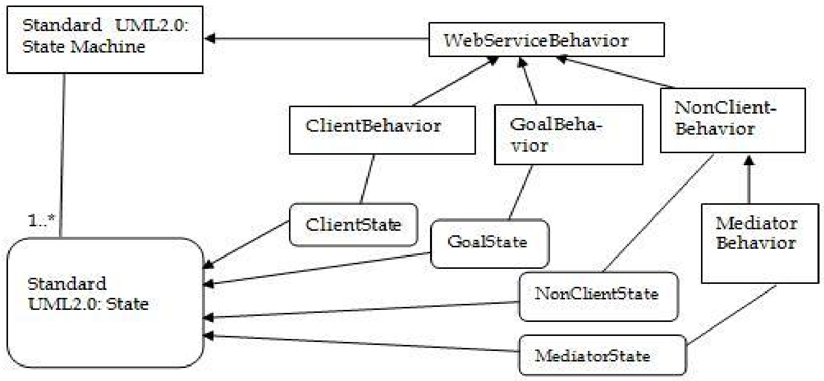


Figure 7 Behavioral description of web services

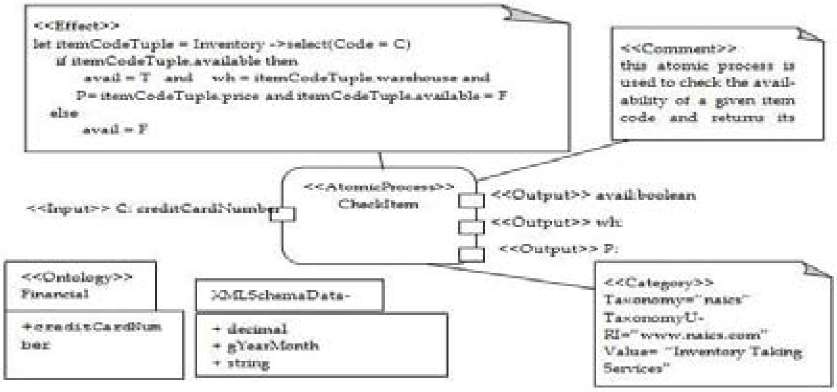

<<Effect>>
let itemCodeTuple = Inventory ->select(Code = C)
if itemCodeTuple.available then
avail = T and wh = itemCodeTuple.warehouse and
P= itemCodeTuple.price and itemCodeTuple.available = F
else
avail = F
<<Comment>>
this atomic process is used to check the availability of a given item code and returns its
<<AtomicProcess>>
CheckItem
<<Input>> C: creditCardNumber
<<Output>> avail:boolean
<<Output>> wh:
<<Output>> P:
<<Ontology>>
Financial
+creditCardNumber
XMLSchemaData-
+ decimal
+ gYearMonth
+ string
<<Category>>
Taxonomy="naics"
TaxonomyURI="www.naics.com"
Value= "Inventory Taking Services"

Figure 8 Representing 'CheckItem' atomic process in UML

The *ClientState* stereotype has one tagged value, which is 'Name'. 'Name' represents the name of the state and it can take either 'ReadyToTransmit' or 'ReadyToRecieve' values. 'TriggerType' tagged value is not introduced for the client because the cause of the transition from one state to the next can be implicitly understood from its Name. If the state name is ReadyToTransmit and the client changes its state to (ReadyToRecieve), this means that the trigger type is 'send', and vice versa.

GoalState stereotype has one tagged value, which is 'TriggerType'. It represents the cause of the transition, which can take the values 'send', 'receive' or 'invoke'. 'send' and 'receive' are restricted to be a communication to or from the client. The *Mediatorstate* stereotype has three tagged values, which are: 'TriggerType', 'To' and 'From'. 'TriggerType' represents the cause of the transition from this state to the next. 'TriggerType' can only take two values which are: 'send' or 'receive'. 'To' is used to represent the *WebService* that the *MeditorService* sends a message to, and 'From' is used to represent the *WebService* that the *MeditorService* receives a message from. 'To' and 'From' tagged values are only used to improve the readability of *MediatorService* state machine. *Standard UML2.0 State machine* captures all the concepts need to represent the guarded automata used in Colombo model. In Standard UML2.0, a *guard* is a Boolean expression that must evaluate to true before a given transition can fire. OCL is the natural candidate for expressing these Boolean expressions. 'Guard' element in Standard UML2.0 plays the same role. Finally Standard UML 2.0 Class diagram can be used to represent World Schema, and Standard UML 2.0 object diagram can be used to represent World Schema instance, World State *I* in a straightforward manner.

By applying the proposed UML profile on the example presented in Section 3.1, Figure 8 presents extended Standard UML 2.0 Activity diagram of the 'CheckItem' atomic process of Figure 3. Futhermore, Figure 9 shows the extended Standard UML 2.0 State Machine diagram of the 'Standard

Warehouse' web service of Figure 4.

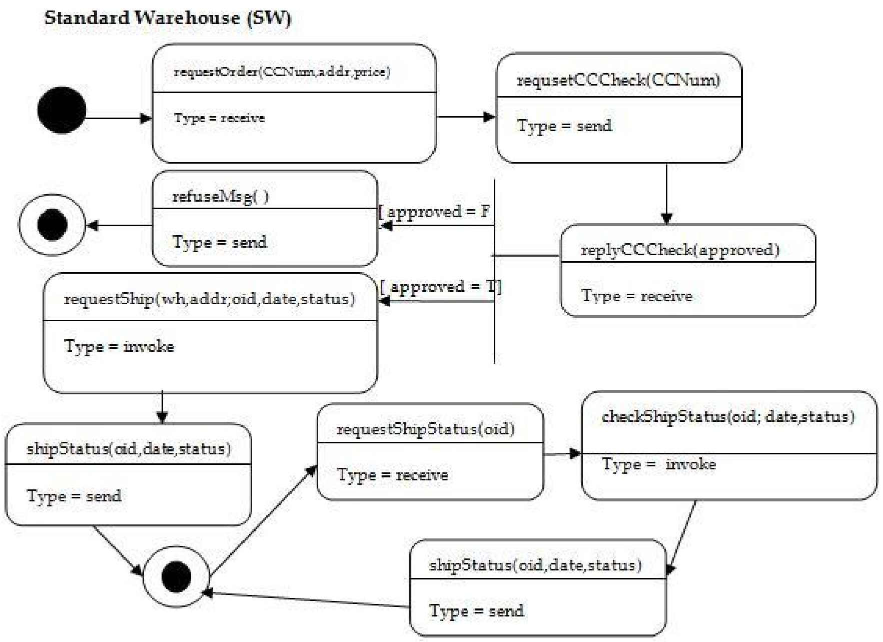


Figure 9 Representing 'Standard Warehouse' web service

5- XML DOCUMENT TYPES FOR COLOMBO

To be able to utilize the automatic composition algorithm proposed in [1], there should be a mapping between the proposed UML profile and Colombo. However, Colombo is still a conceptual model, which means that it does not have a supporting language. We propose a set of coherent XML document types that can be a base for a Colombo language, and also we propose the transformations between UML and XML. We have to point out that the proposed XML is not executable, howver it is an abstract language. Services in Colombo can be specified using three XML document types that represent, which are: (i) the world schema, (ii) the alphabet of atomic processes with links to relevant documents representing the world schema, and (iii) the behavior of a web service with links to appropriate documents representing the alphabet of atomic processes. Next, we present the proposed Document Type Definitions (DTD) and the intuition behind each proposed XML document. DTD serves as a grammar for the underlying XML document. DTD can also serve as schema for the data represented by the XML document.

5-1 XML DOCUMENT TYPE FOR WORLD SCHEMA

Figure 10 presents the DTD tree for the proposed XML document for representing world schema. The root of the XML document is named as WorldSchema. The WorldSchema element can have zero or more relations. Each relation contains an 'id' as its identifier, one or more 'attribute' element, and one and only one 'primaryKey' element. Each attribute has an 'id' as its identifier, 'datatype' and 'ParameterType' elements.

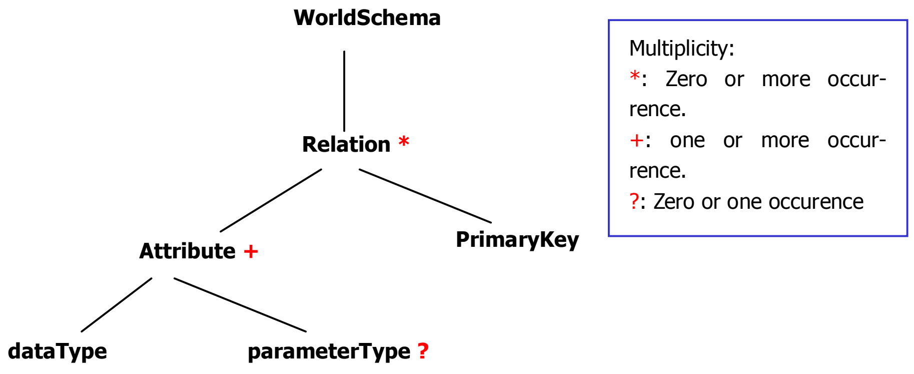


Figure 10 DTD tree of the proposed XML document for representing World Schema

The World Schema of the running example presented in Figure 2 can be represented as an XML document as shown in Figure 11. Due to space limitation, only a part of the XML document is shown in Figure 11 that conforms to the proposed DTD of Figure 10

```
(…)
relation: Relation rdf:ID = "Accounts"
            relation: attribute rdf:ID = "CCNumber"
                        relation:attribute rdf:datatype = "&xsd;#integer"
                        relation:attribute rdf:parameterType = "#cardNumb-
er"
            relation: attribute rdf:ID = "Credit"
                        relation:attribute rdf:datatype = "&xsd;#Boolean"
                        relation:attribute rdf:parameterType = "#..."
            relation:primanyKey rdf:resource = "CCNumber"
relation: Relation rdf:ID = "Inventory"
            relation: attribute rdf:ID = "Code"
                        relation:attribute rdf:datatype = "&xsd;#String"
                        relation:attribute rdf:parameterType = "#..."
            relation: attribute rdf:ID = "Available"
                        relation:attribute rdf:datatype = "&xsd;#Boolean"
                        relation:attribute rdf:parameterType = "#..."
                        (...)
            relation:primanyKey rdf:resource = "Code"

(...)
```

Figure 11 XML document representing world schema of Figure 2

5-2 XML DOCUMENT TYPE FOR ALPHABET OF ATOMIC PROCESSES

The XML Document type proposed for representing the alphabet of atomic processes A is inspired from OWL-S [6]. Figure 12 presents the proposed DTD.

```
<!DOCTYPE AtomicProcessAplha [
        <!ELEMENT AtomicProcessAplha (ID,AtomicProcess*, Ontology*)>
        <!ELEMENT AtomicProcessAplha:ID (#PCDATA) >
        <!ELEMENT AtomicProcess (presents, describedBy, supports,  Profile, Process) >
        <!ELEMENT Profile (Category, hasInput+ , hasOutput+ , hasPrecondition* , hasPost-
        condition* , hasEffect* ) >
        <!ELEMENT Category (addrParam, value, code ) >
        <!ELEMENT addrParam (#PCDATA) >
        <!ELEMENT value (#PCDATA) >
        <!ELEMENT code (#PCDATA) >
        <!ELEMENT Profile:hasInput (#PCDATA) >
        <!ELEMENT Profile:hasOutput (#PCDATA)>
        <!ELEMENT Profile:hasPrecondition SYSTEM "…" >
        <!ELEMENT Profile:hasPostcondition SYSTEM "…" >
        <!ELEMENT Profile:hasEffect SYSTEM "…" >

        <!ELEMENT Process (hasInput+ , hasOutput+ , hasPrecondition* , hasPostcondition* ,
        hasEffect*) >
        <!ELEMENT Process:hasInput (parameterType ) >
        <!ELEMENT Process:hasInput parameterType SYSTEM "…" >
        <!ELEMENT Process:hasOutput (conditionalOutput, CoCondition, parameterType) >
        <!ELEMENT conditionalOutput (#PCDATA)>
        <!ELEMENT coCondition SYSTEM "…">
        <!ELEMENT Process:hasOutput parameterType SYSTEM "…" >
        <!ELEMENT Process:hasEffect (conditionalEffect, ceCondition, ceEffect) >
        <!ELEMENT Process:hasEffect:conditionalEffect (#PCDATA)>
        <!ELEMENT Process:hasEffect: ceCondition SYSTEM "…">
        <!ELEMENT Process:hasEffect: ceEffect SYSTEM "…">

        <!ELEMENT Process:hasPrecondition SYSTEM "…" >
        <!ELEMENT Process:hasPostCondition SYSTEM "…" >
        <!ELEMENT Ontology (class*)>
        <!ELEMENT class (PCDATA)>
        ]>
```

Figure 12 DTD of the proposed XML document for representing Alphabet of atomic processes

The AtomicProcessAplha element (representing the root of the XML document) contains one and only one 'ID' element as its identifier, zero or moreAtomicProcess element, and zero or more Ontology element. Each AtomicProcess element contains a presents element, a describedBy element, a supports element (presents, describedBy, and supports elements simulate the service profile, service model and service grounding parts of the OWL-S model), a Profile and a Process elements. Each Profile element contains a Category element, one or more hasInput element, one or more hasOutput element, zero or more hasPrecondition element, zero or more hasPostcondition element, and one or more hasEffect element. Each Process element contains one or more hasInput element, one or more hasOutput element, zero or more hasPrecondition element, zero or more hasPostcondition element, and one or more hasEffect element. The Profile element can be viewed as a summarization of the Process element. Each Ontology element contains zero or more class elements representing the ontology concepts. Figure 13 shows a part of the XML document representing alphabet of atomic processes of Figure 3, which conforms with the proposed DTD.

```
Profile:category
Profile:      addParam: NAICS rdf:ID = "NAICS.category"
              profile:value
               Commercial Banking
              Profile:code
                 522110
---------------------------------------------------------------------
process:AtomicProcess rdf:ID="CCCheck"
              process:hasEffect
              process:ConditionalEffect rdf:ID= "CheckCreditApproved"
              process:ceCondition rdf:resource = "#EnoughMoney"
              process      :ceEffect rdf:resource = "#ModifyAccounts"
              process:hasEffect
              process:ConditionalEffect rdf:ID= "CheckCreditDisapproved"
              process:ceCondition rdf:resource = "#NoEnoughMoney"
              process      :ceEffect rdf:resource = ""
---------------------------------------------------------------------
process:hasOutput
              process:ConditionalOutput rdf:ID= "CheckCreditCardApproved"
              process:coCondition rdf:resource= "#EnoughMoney"
---------------------------------------------------------------------
process:hasInput
              process:Input rdf:ID= "CheckCreditCardCreditCardNo"
              process:parameterType rdf:resource = "#cardNumber"
```

Figure 13 Part of the XML Document of 'CCCheck' atomic process of Figure 3

5-3 XML DOCUMENT TYPE FOR WEB SERVICES BEHAVIOR

Figure 14 presents the proposed DTD for representing web services behavior. The Service element, which represents the root of the XML document, contains an APAlphabet element, an ID element, an initialState element, one or more finalState elements, and one or more transitionStep elements. APAlphabet element is a reference to the appropriate atomic process alphabet used by the service.

```
<!DOCTYPE Service[
            <!Element Service (APAlphabet, ID,  initialState, finalState+, transitionStep+)
>
            <!Element APAplhabet SYSTEM "" >
            <!Element WorldSchema SYSTEM "" >
            <!Element ID (#PCDATA)>
            <!Element initialState (#PCDATA)>
            <!Element finalState (#PCDATA)>
            <!Element transitionStep (state, guard, transitionEvent, transition) >
            <!Element state (#PCDATA) >
            <!Element guard SYSTEM "" >
            <!Element transitionEvent (#PCDATA) >
            <!Element transition (#PCDATA) >
```

Figure 14 DTD of the proposed XML document for representing web services behavior

```
(...)
service:Service rdf:ID = “Bank”
            service:APAlphabet rdf:resource = “#atomic_process_alphabet”
            service:WorldSchema rdf:resource = “#WorldSchema”

            initialState=“Start”
            finalState=“End”
(...)
service:transitionStep
            service: transitionStep:state rdf:ID = “Start”
            service: transitionStep:guard rdf:resource=“#guard1”
            service: transitionStep:transitionEvent = “?requestCCCheck(CCnum)”
            service: transition: transitionStep rdf:ID = “1”
service: transitionStep
            service: transitionStep:state rdf:ID = “1”
            service: transitionStep:guard rdf:resource=“#guard2”
            service: transitionStep:transitionEvent = “CCCheck(CCnum;approved)”
            service: transitionStep:transition rdf:ID = “2”
(…)
```

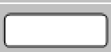
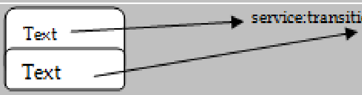
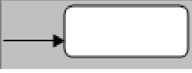
Figure 15 XML document for representing Bank web service

The ID element is an identifier of the service. initialState element represents the initial state of the guarded automata GA. finalState element represents a final state of the GA. transitionStep element contains a state, a gurad, a transitionEvent and a transition elements. State element represents current state. guard is a Boolean expression. Finally, transitionEvent is the cause of the transition (e.g. receiving a message), and transition is the destination state. Figure 15 presents the XML document for representing the bank web service of Figure 4 that conforms to the proposed DTD discussed in Figure 14.

5-4 TRANSFORMATIONS BETWEEN UML AND XML

Table 2 presents a summary of the proposed transformation rules between UML and XML. These transformation rules can be embedded into UML model Transformation Tool (UMT) [15]. This is a tool used to transform UML to any other source (e.g. XML, C#). Embedding the transformation rules ino UMT is considered as one of our future work directions. Next we will present the prototypical implementation of the approach proposed in this paper.

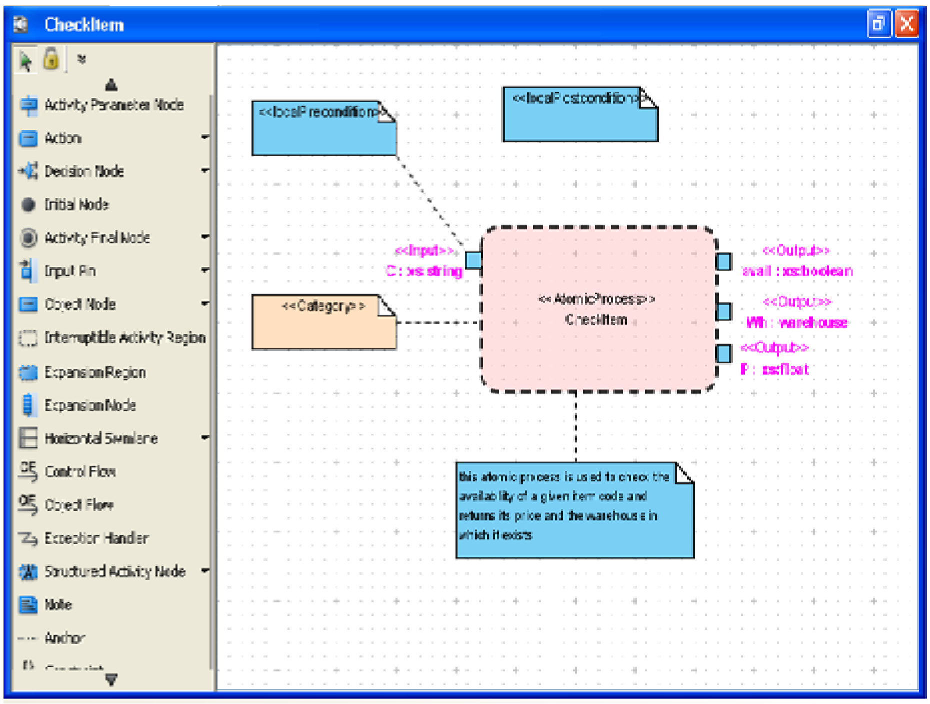
Table 2 Summary of the transformation rules between UML and XML

	UML	**Proposed XML Documents**
World Schema	Class name	Relation:Relation rdf:ID...
	Attribute	Relation:attribute
	Primary key	Relation:primaryKey
Atomic Processes Alphabet	<<Category>>	profile: category
	<<AtomicProcess>>	atomicProcess:AtomicProcess rdf:ID...
	<<Effect>>	profile:hasEffect
		Process:hasEffect
	<<Pre-condition>>	profile:hasPrecondition
		Process:hasPrecondition
	<<Post-condition>>	profile:hasPostcondition
		Process:hasPostcondition
	<<Input>>	profile:hasInput
		Process:hasInput
	<<Output>>	profile:hasOutput
		Process:hasOutput
	<<Ontology>>	Colombo:Ontology
	<<OntClass>>	Colombo:Class
Web Service Behavior	●	initialState
	◉	finalState
		service:state
	[guard logical expression]	service:guard
		service:transitionEvent
		Service:transition

6- PROTOTYPE IMPLEMENTATION

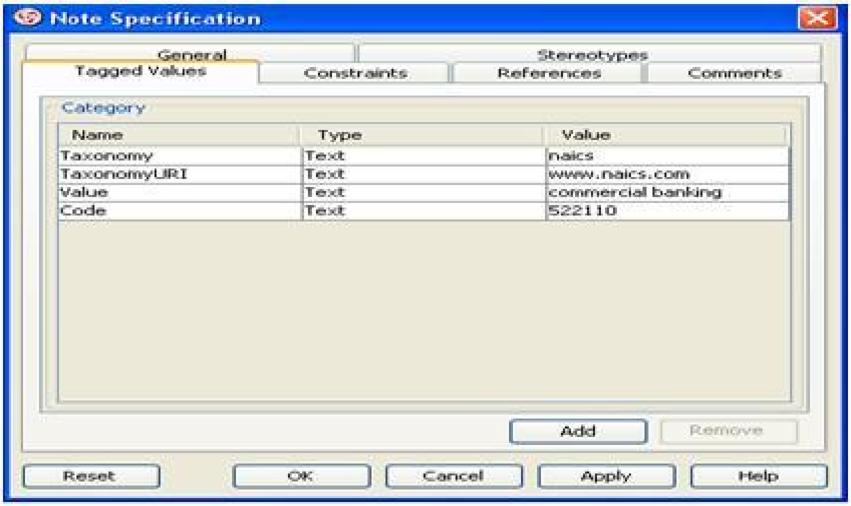
An effective and scalable implementation of the proposed UML profile discussed above is a challenging yet necessary step to ascertain the soundness of the approach proposed in this paper. The prototypical implementation is developed by creating a prototype of a CASE tool that would be used to facilitate the design of composite web services. We have used Visual Paradigm for UML (VP-UML) Version 6 [25] for this purpose, mainly because it supports UML 2.0 and its extension mechanisms and allows the automatic transformation from UML to XML. Besides, it allows converting to a wide variety of source code (such as C#, C++, JAVA, PHP…etc) to UML. VP-UML permits the user to select a language (such as C#, C++, XML…etc) that allows the attributes to be typed compatible to the selected language. This feature is of a great value when transforming the UML diagrams to any other language. Figure 16 presents a snapshot of the implemented activity diagram of 'CheckItem' atomic process of Figure 3. Moreover, Figure 17 presents the introduced tagged values for the Category stereotype of the CheckItem atomic process. Furthermore, the state machine diagram of the Bank web service of the running example is presented in

Figure 18. Finally, Figure 19 presents a cascaded view of the implemented mapping from the UML diagrams of the running example to XML (refer to [26] for more details about the experimintal prototype).



this atomic process is used to check the availability of a given item code and returns its price and the warehouse in which it exists

Figure 16 Activity diagram of 'CheckItem' atomic process


Note Specification

General | Stereotypes | Tagged Values | Constraints | References | Comments

Category

Name	Type	Value
Taxonomy	Text	naics
TaxonomyURI	Text	www.naics.com
Value	Text	commercial banking
Code	Text	522110

Add | Remove

Reset | OK | Cancel | Apply | Help

Figure 17 Tagged values of Category stereotype for 'CheckItem' atomic process

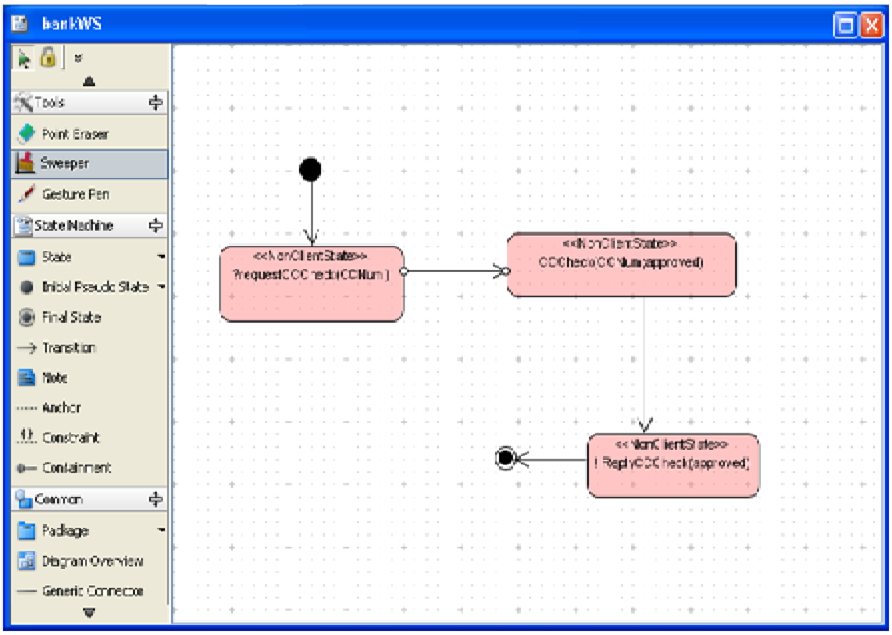
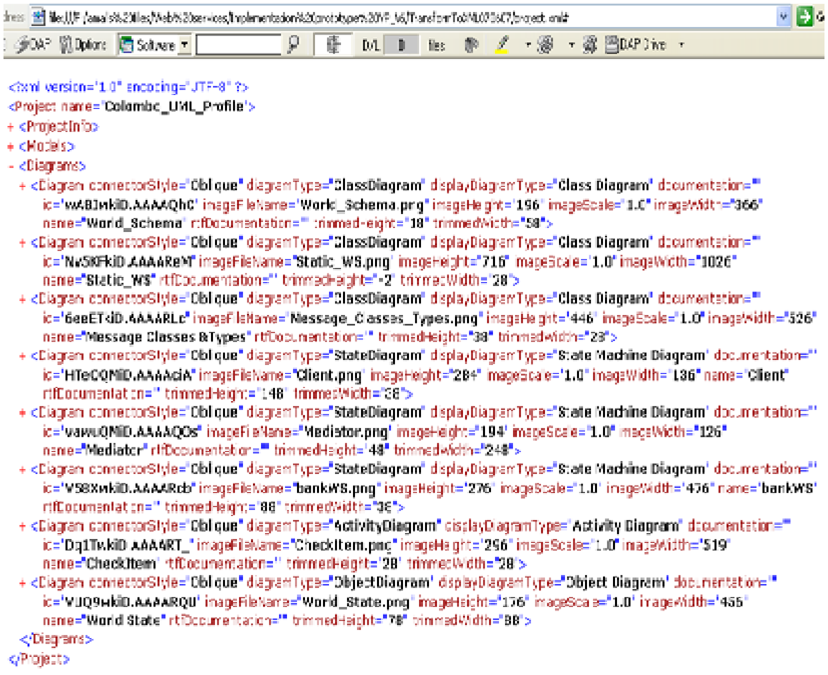


Figure 18 Implemented state machine diagram of 'Bank' web service



```
<?xml version="1.0" encoding="UTF-8" ?>
<Project name="Colombo_UML_Profile">
+ <ProjectInfo>
+ <Models>
- <Diagrams>
  + <Diagram connectorStyle="Oblique" diagramType="ClassDiagram" displayDiagramType="Class Diagram" documentation=""
      id="wABJmkiD.AAAAQhC" imageFileName="World_Schema.png" imageHeight="196" imageScale="1.0" imageWidth="366"
      name="World_Schema" rtfDocumentation="" trimmedHeight="18" trimmedWidth="58">
  + <Diagram connectorStyle="Oblique" diagramType="ClassDiagram" displayDiagramType="Class Diagram" documentation=""
      id="Nw5KFkiD.AAAAReM" imageFileName="Static_WS.png" imageHeight="716" imageScale="1.0" imageWidth="1026"
      name="Static_WS" rtfDocumentation="" trimmedHeight="-2" trimmedWidth="28">
  + <Diagram connectorStyle="Oblique" diagramType="ClassDiagram" displayDiagramType="Class Diagram" documentation=""
      id="6eeETkiD.AAAARLc" imageFileName="Message_Classes_Types.png" imageHeight="446" imageScale="1.0" imageWidth="526"
      name="Message Classes &Types" rtfDocumentation="" trimmedHeight="38" trimmedWidth="23">
  + <Diagram connectorStyle="Oblique" diagramType="StateDiagram" displayDiagramType="State Machine Diagram" documentation=""
      id="HTeCQMiD.AAAAciA" imageFileName="Client.png" imageHeight="284" imageScale="1.0" imageWidth="136" name="Client"
      rtfDocumentation="" trimmedHeight="148" trimmedWidth="38">
  + <Diagram connectorStyle="Oblique" diagramType="StateDiagram" displayDiagramType="State Machine Diagram" documentation=""
      id="vawuQMiD.AAAAQOs" imageFileName="Mediator.png" imageHeight="194" imageScale="1.0" imageWidth="126"
      name="Mediator" rtfDocumentation="" trimmedHeight="48" trimmedWidth="248">
  + <Diagram connectorStyle="Oblique" diagramType="StateDiagram" displayDiagramType="State Machine Diagram" documentation=""
      id="V58XmkiD.AAAARcb" imageFileName="bankWS.png" imageHeight="276" imageScale="1.0" imageWidth="476" name="bankWS"
      rtfDocumentation="" trimmedHeight="88" trimmedWidth="38">
  + <Diagram connectorStyle="Oblique" diagramType="ActivityDiagram" displayDiagramType="Activity Diagram" documentation=""
      id="Dq1TwkiD.AAAART_" imageFileName="CheckItem.png" imageHeight="256" imageScale="1.0" imageWidth="519"
      name="CheckItem" rtfDocumentation="" trimmedHeight="28" trimmedWidth="28">
  + <Diagram connectorStyle="Oblique" diagramType="ObjectDiagram" displayDiagramType="Object Diagram" documentation=""
      id="VUQ9mkiD.AAAARQU" imageFileName="World_State.png" imageHeight="176" imageScale="1.0" imageWidth="455"
      name="World State" rtfDocumentation="" trimmedHeight="78" trimmedWidth="88">
  </Diagrams>
</Project>
```

Figure 19 A Cascaded view of the corresponding XML document

7- CONCLUCION

In this paper we have proposed to model Colombo web services in UML. For this purpose, we have proposed a UML profile that can be used to model every component of the Colombo model. Consequently, the web service developer will deal with the UML high level graphical models, and this in turn will have a positive impact on her productivity and will facilitate her work. We have outlined a prototypical tool by using Visual Paradigm for UML (VP-UML) version 6.

Colombo model is a conceptual model, which means that it does not have a supportive language. We have proposed a set of related XML document types that can be the core of a Colombo language. Furthermore, we have proposed the transformation rules between UML and Colombo proposed XML documents. Next the automatic composition algorithm proposed in [1] can be utilized assuming that it accepts and produces XML documents that conform to the DTDs proposed in this paper.

REFERENCES

[1] D. Berardi , D. Calvanese, G. Giacomo, R. Hull, and M. Mecella, "Automatic Composition of Transition-based Semantic Web Services with Messaging," 31st Very Large Database (VLDB) Conference, Norway, pp. 613-624, 2005.

[2] R. Hull, M. Benedikt, V. Christophides, and J. SU, "E-Services: A Look behind the Curtain," Principles of Database (PODS) International Conference, USA, pp. 1-14, 2003.

[3] "Web Service Description Language (WSDL) Version 1.1," http://www.w3.org/TR/wsdl.

[4] "Web Service Description Language (WSDL) Version 2.0 Part 2: Predefined Extensions (W3C Working Draft)," http://www.w3.org/TR/2004/WD-wsdl20-extensions-20040803.

[5] UDDI.org, *UDDI Technical White Paper*, 2000.

[6] A. Ankolekar , M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara, "DAML-S: Web Services Description for Semantic Web," International Semantic Web Conference (ISWC'02), Italy, pp. 348-363, 2002.

[7] D. Berardi, "Automatic Service Composition: Models, Techniques, and Tools," Dipartimento di Informatica e Sistemistica, Universit`a di Roma "La Sapienza", Roma, 2005.

[8] D. Berardi, D. Calvanese, G. Giacomo, M. Lanzerini, and M. Mecella, "Automatic Composition of E-Services that Export their Behavior," 1st International Conference on Service Oriented Computing (ICSOC), Italy, pp. 43-58, 2003.

[9] T. Bultan, X. Fu, R. Hull, and J. Su, "Conversation Specification: A New Approach for the Design and Analysis of E-Service Composition," International World Wide Web Conference (WWW), Hungary, pp. 403-410, 2003.

[10] D. Berardi, D. Calvanese, G. Giacomo, R. Hull, and M. Mecella, "Modeling Data & Processes for Service Specifications in Colombo," the Open Interop. Workshop on Enterprise Modeling & Ontologies for Interope-rability (EMOI INTEROP'05), Portugal, 2005.

[11] M. Lopez-Sanz, C. Acuna, C. Cuesta, and E. Marcos, "Modelling of Service-Oriented Architectures with UML," *Electronic Notes in Theoretical Computer Science,* vol. 194, pp. 23-37, 2008.

[12] R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik, "Model-Driven Web Services Development," IEEE International Conference on E-Technology, E-Commerce and E-Service (EEE'04), Taiwan, pp. 42-25, 2004.

[13] R. Gronmo, and J. Oldevik, "An Empirical Study of the UML Model Transformation Tool (UMT)," 1st International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA'05), Switzerland, 2005.

[14] S. Thone, R. Depke, and G. Engels, " Process-Oriented, Flexible Composition of Web Services with UML," International Workshop on Conceptual Modeling Approaches for E-business: A Web Service Perspective (eCOMO 2002), Finland, pp. 390-401, 2002.

[15] W. Provost. "UML for Web Services, XML.com," http://www.xml.com/lpt/a/ws/2003/08/05/uml.html

[16] D. Carson. "Hypermodel," www.ontogenics.com.

[17] B. Bauer, and M. Huget, "Modelling web service composition with UML 2.0," *International Journal of Web Engineering and Technology,* vol. 1, no. 4/2004, pp. 484-501.

[18] "BPEL," Business Process Execution Language for Web Services. http://www.ibm.com/developerworks/webservices/library/specification/ws-bpelsubproc/, 2005].

[19] R. Gronmo, M. Jaeger, and H. Hoff, "Transformations between UML and OWL-S," International European Conference on Model Driven Architecture –Foundations and Applications (ECMDA-FA), Germany, pp. 269-283, 2005.

[20] D. Elenius, G. Denker, D. Martin, F. Filham, J. Khouri, S. Sadaati, and R. Sena-nayake, "The OWL-S Editor – A Development Tool for Semantic Web Services," 2nd European Semantic Web Conference (ESWC 2005), Greece, pp. 78-92, 2005.

[21] A. Gomez-Perez, R. Gonzalez-Cabero, and M. Lama, "ODE SWS: A Framework for Designing and Composing Semantic Web Services," 2nd International IEEE Conference on Intelligent Systems, Bulgaria, pp. 24-31, 2004.

[22] P. Rajasekaran, J. Miller, K. Verma, and A. Sheth, "Enhancing Web Services Description and Discovery to Facilitate Composition," Semantic Web Services and Web Process Composition, 1st International Workshop (SWSWPC 2004), USA, pp. 55-68, 2004.

[23] K. Scott, *Fast Track UML 2.0*: APress, 2004.

[24] D. Djuric, "MDA-based Ontology Infrastructure," *Computer Science Information Systems (ComSIS)*, pp. 91-116, 2004.

[25] "Visual Paradigm for UML," http://www.visual-paradigm.com/.

[26] A. Elgammal "Efficient Techniques for Building Web Services," Masters dissertation, Information Systems Dept., Faculty of Computers and Information, Cairo University, Cairo, 2007.