Modeling Data & Processes for Service Specifications in `Colombo`

Daniela Berardi[1], Diego Calvanese[2], Giuseppe De Giacomo[1]
Richard Hull[3], Maurizio Lenzerini[1], and Massimo Mecella[1]

[1] *Università di Roma "La Sapienza"*,
Dipartimento di Informatica e Sistemistica,
Via Salaria 113, I-00198, Roma, Italy
{`berardi,degiacomo,lenzerini,mecella`}`@dis.uniroma1.it`
[2] *Libera Università di Bolzano/Bozen*
Facoltà di Scienze e Tecnologie Informatiche
Piazza Domenicani 3, 39100 Bolzano, Italy
`calvanese@inf.unibz.it`
[3] *Bell Labs, Lucent Technologies*,
Network Data and Services Research Department,
600 Mountain Ave., Murray Hill, NJ 07974, USA
`hull@lucent.com`

Abstract. In this paper we present how to model data and processes in `Colombo`, a conceptual framework for Service Oriented Computing where web services are characterized in terms of *(i)* message exchanges, *(ii)* data flow, and *(iii)* effects on the real world. While all these aspects have been separately considered in the literature, dealing with all of them together is particularly challenging: `Colombo`, by proposing a semantically rich service model for services, enables a single coherent framework, on top of which automatic service interoperability and composition are feasible.

1 Introduction

Service Oriented Computing (SOC [1]) is the computing paradigm that utilizes web services (also called *e*-Services or, simply, services) as fundamental elements for realizing distributed applications/solutions. Web services are self-describing, platform-agnostic computational elements that support rapid, low-cost and easy composition of loosely coupled distributed applications.

Through web services, enterprises can set up new business opportunities, by integrating their applications, making them interoperable and building new complex value added services. But in order to achieve such an integration, many challenging research issues should be solved, the most hyped one being *service composition*; this in turn requires semantically rich *service models*.

Service composition addresses the situation when a client request cannot be satisfied by any available web service, but by suitably combining "parts of" the available web services. Composition involves two different issues [1]. The first, typically called *composition synthesis*, is concerned with synthesizing a specification of how to coordinate the component services to fulfill the client request. Such a specification can be produced either *automatically*, i.e., using a tool that implements a composition algorithm, or *manually* by a human. The second issue, often referred to as *orchestration*, is concerned with how to actually achieve the coordination among services, by executing the specification produced by the composition synthesis and by suitably supervising and monitoring both the control flow and the data flow among the involved services. Most of the work on service orchestration is based on research in workflows [5].

In order to discuss *automatic* service composition, a conceptual framework can be introduced, as shown in Figure 1. It is constituted by the following elements:

– **(Available) Services**. They denote the set of services to be considered (for integration) in a composition.
– **Community Ontology**. Cooperating services need to share a common understanding on an agreed upon reference alphabet/semantics. Note that many scenarios of cooperative information systems, e.g., *e*-Government or *e*-Business, consider preliminary agreements on underlying ontologies, yet yielding a high degree of dynamism and flexibility. The community ontology realizes this semantic sharing. It is therefore a basic element in a service composition architecture, since, in general, it addresses and solves issues concerning the meaning of the offered operations, the semantics of the data flowing through the service operations, etc.
– **Mapping**. Available services that are considered in a cooperation, need to expose their behavior in terms of the community ontology. Therefore, a mapping should be defined between symbols of the alphabet of the community ontology and operations of the service (e.g., how operations offered by a service are defined in terms of the alphabet of operations in the community ontology).
– **Client Service Request**. A client willing to exploit a (possibly composite) service specifies a service request using the alphabet of the community ontology. The community realizes the client service request making use of the available services, via the mapping.

The community ontology acts as a composite service provider wrt its clients: the composite service is obtained by "suitably" coordinating the existing ones, thus satisfying the client request. Note also that the terms (i.e., operations) in the community ontology are virtual, since defined in terms of existing ones (i.e., operations offered by real services) via the mapping. Consequently, the composite services provided by the community ontology are *virtual*. The client thinks he is invoking an operation on a real service, but actually, that operation may correspond (via the mapping) to a bunch of operations (since the mapping can have any complexity) offered by different real services.

Therefore techniques for automatic service composition tightly depend on how services, the community ontology and the mappings are modeled in semantically rich ways. In fact, semantically richer are the service descriptions, more properties can be inferred and used during synthesis and orchestration.

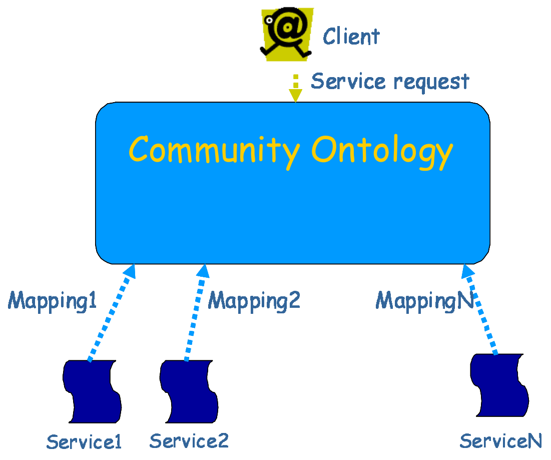


Fig. 1. Conceptual framework for automatic service composition

In this paper we present a model for services, called `Colombo`, that combines four fundamental aspects of web services, namely: *(i)* A world state, representing the "real world", viewed as a database instance over a relational database schema. This is similar to the family of "fluents" found in OWL-S [9]. *(ii)* Atomic processes, which can access and modify the world state, and may include conditional effects and non-determinism. These are inspired by the atomic processes of OWL-S. *(iii)* Message passing, including a simple notion of ports and links, as found in web services standards (e.g., WSDL, BPEL4WS) and some formal investigations (e.g., [4, 11]). *(iv)* The behavior of web services (which may involve multiple atomic processes and message-passing activities) is specified using finite state transition system, following and extending [2, 4]. Thus, `Colombo` provides a bridge between BPEL4WS and OWL-S, at the same time being compliant with the emerging SWSL for semantic web services [6]. We also assume that each web service instance has a *local store*, used to capture parameter values of incoming messages and the output values of atomic processes, and used to populate the parameters of outgoing messages and the input parameters of atomic processes. Conditional branching in a web service will be based on the values of the local store variables at a given time.

With respect to the conceptual framework, the community ontology is therefore basically constituted by the "world schema" (i.e., the intensional description of the world state) and the atomic processes; services are described (i.e., their mappings with respect to the community ontology) in terms of transition systems over such elements; the client request is expressed as a transition system over the community ontology and the result of the synthesis is again a transition system.

A client of a web service interacts with it by repeatedly sending and receiving messages, until a certain situation is reached. In other words, also the

Accounts

CCNumber	credit
1234	T
...	...

PREPaid

PREPaidNum	credit
5678	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
...	...	...	...	...

Fig. 2. World Schema Instance

client behavior can be abstractly represented as a transition system having exactly two states, between which the client toggles, called `ReadyToTransmit` and `ReadyToRead`, where the first is the start state and also the final state.

The rest of the paper is organized as follows. Section 2 illustrates the main modeling features of `Colombo` with an example, whereas all the formal details are omitted for brevity; the interested reader can refer to [3]. Section 3 outlines how automatic service composition can be achieved based on the proposed model, and Section 4 concludes the paper.

2 The `Colombo` Framework

In this section, we intuitively illustrate `Colombo` by means of an example involving web services that manage inventories, payment by credit or prepaid card, request shipments, and check shipment status.

The real world is captured by the *world (database) schema*, which is a finite set $\mathcal{W}$ of relations having the form: $R_k(A_1, \ldots, A_{m_k}; B_1, \ldots, B_{n_l})$, where $A_1, \ldots, A_{m_k}$ is a key for R_k, and where each attribute A_i, B_j is defined over *(i)* the boolean domain $Bool$, *(ii)* an infinite set of uninterpreted elements $Dom_=$ (denoted in the example by alphanumeric strings), on which only the equality relation is defined, or *(iii)* an infinite densely ordered set $Dom_\leq$ (denoted in the example by numbers). We set $Dom = Bool \cup Dom_= \cup Dom_\leq$. Figure 2 shows a *world instance*, i.e., a database instance over $\mathcal{W}$. For each relation, the key attributes are separated from the others by the thick separation between columns. The intuition behind these relations is as follows: `Accounts` stores credit card numbers and the information on whether they can be charged; `PREPaid` stores prepaid card numbers and the information on whether they can be still be used; `Inventory` contains item codes, the warehouse they are available in, if any, and the price; `Shipment` stores order id's, the source warehouse, the target location, status and date of shipping.

The available web services and the goal service specification are defined over a common alphabet $\mathcal{A}$ of atomic processes, as the one shown in Figure 3. Formally, an *atomic process* is an object p which has a signature of form (I, O, CE) with the following properties. The *input signature* I and *output signature* O are

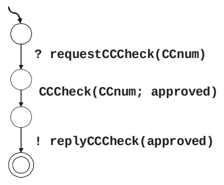
```
CCCheck
 I: c:$Dom_=$; % CC card number
 O: app:$Bool$; % CC approval
 effects:
  if $f_1^{Accounts}(c)$ then
   either modify Accounts(c;T) or
    modify Accounts(c;F) and approved:= T
  if $\neg f_1^{Accounts}(c)$ then
   approved:= F

checkItem:
 I: c:$Dom_=$; % item code
 O: avail:$Bool$; wh:$Dom_=$; p:$Dom_\leq$ % resp. item availa-
               % bility, selling warehouse and price
 effects:
  if $f_1^{Inventory}(c)$ then
   avail:= T and 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 avail:= F
```

```
charge:
 I: c:$Dom_=$; % Prepaid card number;
 O: paymentOK:$Bool$; % Prepaid card approval
 effects:
  if $f_1^{PrePaid}(c)$ then
   either modify PrePaid(c;T) or modify PrePaid(c;F)
   and paymentOK:= T
  if $\neg f_1^{PrePaid}(c)$ then paymentOK:= F

requestShip:
 I: wh:$Dom_=$; addr:$Dom_=$; % resp. source warehouse
                              % and target address
 O: oid:$Dom_=$; d:$Dom_\leq$; s:$Dom_=$; % resp. order id,
                              % shipping date and status
 effects:
  $\exists d, o$ oid:=new(o) and
   insert Shipment(new(oid); wh, addr, ‘‘requested’’, d)
   and d:=$f_4^{Shipment}(oid)$ and s := ‘‘requested’’

checkShipStatus:
 I: oid:$Dom_=$; % order id
 O: s:$Dom_=$; d:$Dom_\leq$; % 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:=$f_4^{Shipment}(oid)$
```

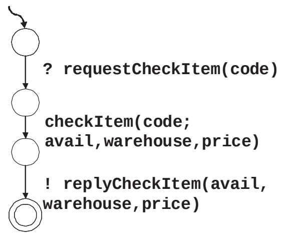
Fig. 3. Alphabet $\mathcal{A}$ of Atomic Processes

sets of typed variables, i.e., of variables belonging to *Dom*. The *conditional effect*, CE, is a set of pairs of form (c, E), where c is a (*atomic process*) *condition* and E is a finite non-empty set of (*atomic process*) *effect* (*specifications*). Intuitively, $\mathcal{A}$ represents the common understanding on an agreed upon reference alphabet/semantics that cooperating web services should share [5]. For succinctness we use a pidgin syntax for specifying the atomic processes in that figure. We denote the null value using ω. The special symbol "$-$" denotes elements of tuples that remain unchanged after the execution of the atomic process. When defining (conditional) effects of atomic processes, we specify the potential effects on the world state using syntax of the form '*insert*', '*delete*', and '*modify*'. These are suggestive of procedural database manipulations, but are intended as shorthand for declarative statements about the states of the world before and after an effect has occurred. Finally, we use the function $f_j^R(\langle a_1, \ldots, a_n \rangle)$ to fetch the $(n + j)$-th element of the tuple in R identified by the key $\langle a_1, \ldots, a_n \rangle$ (i.e., the j-th element of the tuple after the key).

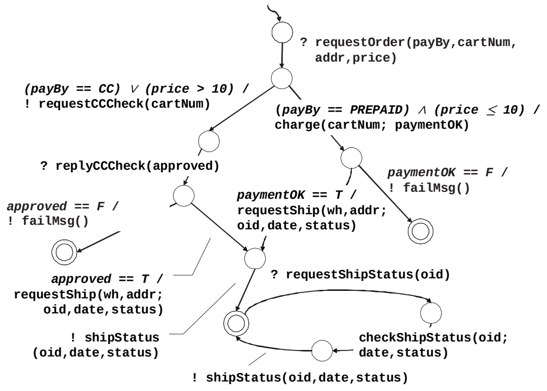
Figure 4 shows (the transition systems of) the available web services: `Bank` checks that a credit card can be used to make a payment; `Storefront`, given the code of an item, returns its price and the warehouse in which the item is available, if any; `Next Generation Warehouse (NGW)` allows for *(i)* dealing with an order either by credit card or by prepaid card, according to the client's preferences and to the item's price, and for *(ii)* shipping the ordered item, if the payment card is valid; `Standard Warehouse (SW)` deals only with orders by credit cards, and allows for shipping the ordered item, if the card is valid. Throughout the example we are assuming that other web services are able to change the status and, possibly, to postpone the date of item delivery using suitable atomic processes, which are not shown in Figure 3. Finally, in Figure 4, transitions concerning message exchanges are labeled with an operation to transmit or to read a message, by prefixing the message with `!` or `?`, respectively.



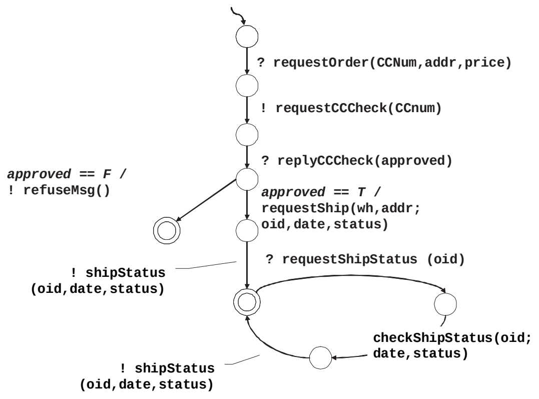
(a) Bank



(b) Storefront



(c) Next Generation Warehouse



(d) Standard Warehouse

Fig. 4. Transition systems of the available services

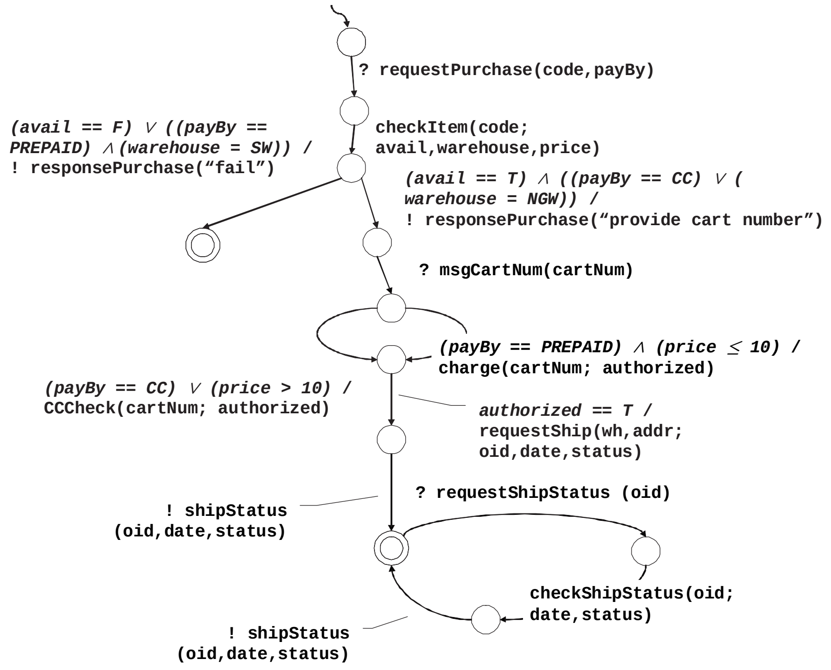

? requestPurchase(code,payBy)
checkItem(code;
avail,warehouse,price)
(avail == F) ∨ ((payBy ==
PREPAID) ∧ (warehouse = SW)) /
! responsePurchase("fail")
(avail == T) ∧ ((payBy == CC) ∨ (
warehouse = NGW)) /
! responsePurchase("provide cart number")
? msgCartNum(cartNum)
(payBy == PREPAID) ∧ (price ≤ 10) /
charge(cartNum; authorized)
(payBy == CC) ∨ (price > 10) /
CCCheck(cartNum; authorized)
authorized == T /
requestShip(wh,addr;
oid,date,status)
? requestShipStatus (oid)
! shipStatus
(oid,date,status)
checkShipStatus(oid;
date,status)
! shipStatus
(oid,date,status)

Fig. 5. Transition system of the goal service

All the available web services are also characterized by the following elements (for simplicity, not shown in the figure): *(i)* An internal local store, i.e., a relational database defined over the same domains as the world state (namely, *Bool*, $Dom_=$, and $Dom_\leq$). *(ii)* One port for each message (type) a service can transmit or receive. As an example, the web service `Bank` has two ports, one for receiving messages (of type) `requestCCCheck(CCnum)` and another for sending messages (of type) `replyCCCheck(approved)`. Each port for an incoming message has associated a queue (see below) and a web service can always transmit messages, but can receive them only if the queue is not full. A received message is then read (and erased from the queue) when the process of the web service allows it. *(iii)* One queue (of length one) for each message type the web service can receive. The queues are used to store messages that have been received but not read yet. For example, the web service `Bank` has one queue, for storing messages (of type) `requestCCCheck(CCnum)`. *(iv)* A set of links between pairs of services that allow communication among them. Specifically, let $\mathcal{F} = \{S_1, \ldots, S_n\}$ be a family of services. A *link* for $\mathcal{F}$ is a tuple of form (S_i, m, S_j, n) where m is a message that can be transmitted by S_i, n is a message that can be read by S_j, having the same type as m. A *linkage* for $\mathcal{F}$ is a set L of links such that the first two fields of L are a key for L, and likewise for the second two fields. In this paper we assume that a linkage L is established at the time of designing a system of interoperating services, and that L does not change at runtime.

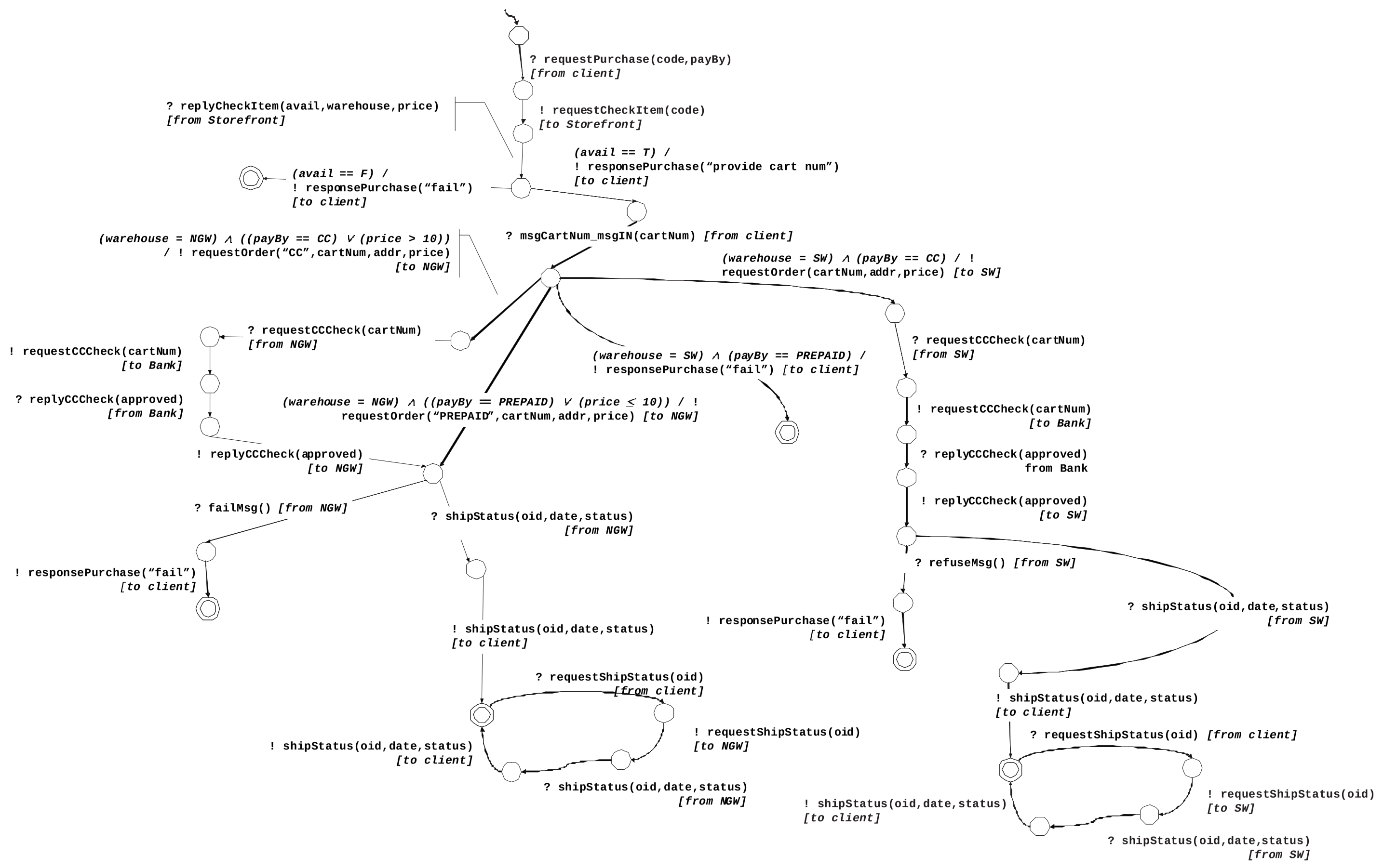

? requestPurchase(code,payBy) [from client]
! requestCheckItem(code) [to Storefront]
? replyCheckItem(avail,warehouse,price) [from Storefront]
(avail == F) / ! responsePurchase("fail") [to client]
(avail == T) / ! responsePurchase("provide cart num") [to client]
? msgCartNum_msgIN(cartNum) [from client]
(warehouse = NGW) ∧ ((payBy == CC) ∨ (price > 10)) / ! requestOrder("CC",cartNum,addr,price) [to NGW]
(warehouse = SW) ∧ (payBy == CC) / ! requestOrder(cartNum,addr,price) [to SW]
? requestCCCheck(cartNum) [from NGW]
! requestCCCheck(cartNum) [to Bank]
? replyCCCheck(approved) [from Bank]
! replyCCCheck(approved) [to NGW]
(warehouse = SW) ∧ (payBy == PREPAID) / ! responsePurchase("fail") [to client]
(warehouse = NGW) ∧ ((payBy == PREPAID) ∨ (price ≤ 10)) / ! requestOrder("PREPAID",cartNum,addr,price) [to NGW]
? failMsg() [from NGW]
! responsePurchase("fail") [to client]
? shipStatus(oid,date,status) [from NGW]
! shipStatus(oid,date,status) [to client]
? requestShipStatus(oid) [from client]
! requestShipStatus(oid) [to NGW]
? shipStatus(oid,date,status) [from NGW]
! shipStatus(oid,date,status) [to client]
? requestCCCheck(cartNum) [from SW]
! requestCCCheck(cartNum) [to Bank]
? replyCCCheck(approved) from Bank
! replyCCCheck(approved) [to SW]
? refuseMsg() [from SW]
! responsePurchase("fail") [to client]
? shipStatus(oid,date,status) [from SW]
! shipStatus(oid,date,status) [to client]
? requestShipStatus(oid) [from client]
! requestShipStatus(oid) [to SW]
? shipStatus(oid,date,status) [from SW]
! shipStatus(oid,date,status) [to client]

Fig. 6. Transition system of the mediator

Figure 5 shows (the transition system of) a goal service: it allows *(i)* to buy an item characterized by a given code; *(ii)* to pay for it either by credit card or prepaid, depending on the client's preferences, the item's price and the warehouse in which the item is stored; and *(iii)* to check the shipment status. Note that the goal service specifies both message-based interactions with the client (e.g., `?requestPurchase(code,payBy)` for receiving from the client the item code and the preferred payment method) and atomic processes that the available web service contained in the composition should execute.

With automatic composition techniques, we would like to automatically construct a mediator such as S_0 shown in Figure 6; a mediator is the specification to be orchestrated in order to realize the service requested by the client. As an aid to the reader, we explicitly indicate in the figure the sender or the receiver of each message, in order to provide an intuition of the linkage. Note that, differently from the goal service, the mediator specifies message-based interaction only, involving either the client or a web service. The mediator is also characterized by a local store, a set of ports and a queue for each incoming message (type), not shown in the figure.

An example of interactions between S_0, the client and the available web services is as follows. S_0 reads a `requestPurchase(code,payBy)` message that has been transmitted by a client (into the suitable queue) and stores it into its local store: such a message specifies the code of an item and the client's preferred payment method. Then, S_0 transmits the message `requestCheckItem(code)` to `Storefront` (i.e., into its queue) and waits for the answer (for simplicity we assume that the queue is not full). Thus, `Storefront` reads from its queue that message (carrying the item's code), executes the atomic process `checkItem(...)` by accessing the tuple of relation `Accounts` having as key the given code: at this point, the information on the warehouse the item is available in (if any) and its price can be fetched and transmitted to the mediator. Hence, S_0 reads the message `replyCheckItem(avail,warehouse,price)` and stores the values of its parameters into its local store. If no warehouse contains the item (i.e., `avail == F`), S_0 transmits a `responsePurchase(''fail'')` message to the client, informing her that the request has failed, otherwise (i.e., `avail == T`) S_0 transmits a `responsePurchase(''provide cart num'')` to the client, asking her for the card number, and the interactions go on.

For brevity, we omit formal details, that are conversely presented in [3]. In the next section we outline how we are able to achieve automatic composition on the basis of the proposed model.

3 Composition Synthesis with `Colombo`

In this section we outline how to solve the composition synthesis problem on the basis of the proposed model.

Let $\mathcal{W}$ be a world schema and $\mathcal{A}$ be an alphabet of atomic processes. Assume that a family of (pre-defined) services operating over $\mathcal{A}$ is available (e.g., in an extended UDDI directory). We also assume that the desired composite service is specified in terms of a *goal system*, i.e., a triple $\mathcal{G} = (C, \{G\}, L)$ where C is a client (modeled as a transition system, see Section 1); G is the goal service, defined over $\mathcal{A}$; and L is a linkage involving only C and G.

In the general case, given goal system $\mathcal{G} = (C, \{G\}, L)$, the *composition synthesis problem* is to *(a)* select a family $S_1, \ldots, S_n$ of services from the pre-existing set, *(b)* construct a web service S_0 (the "mediator") which can only send, receive and read messages, and *(c)* construct a linkage L' over $C, S_0, S_1, \ldots, S_n$ such that $\mathcal{G}$ and $\mathcal{S} = (C, \{S_0, S_1, \ldots, S_n\}, L')$ are equivalent, i.e., the behaviors of $\mathcal{G}$ and $\mathcal{S}$ are indistinguishable relative to what is observable in terms of client messaging and atomic process invocations (and their effects).

Decidability of the composition synthesis problem remains open for most cases of the general `Colombo` framework. We can achieve decidability and complexity results under the assumptions[4] that: *(i)* concurrency is prevented in our systems; *(ii)* in any enactment of $\mathcal{G}$, only a finite number of domain values are read (thus providing a uniform bound on the size of the "active domain" of any enactment); *(iii)* all messages in a composition are either sent by the mediator S_0 or received by the mediator (note that this assumption affects the form of the linkages). Finally, we say that a mediator service is (p, q)*-bounded* if it has at most p states in its transition system and at most q variables in its global store.

Theorem 1. *Let $\mathcal{G} = (C, \{G\}, L)$ be a goal system and $\mathcal{U}$ a finite family of available web services. For each p, q it is decidable whether there is a set $\{S_1, \ldots, S_n\} \subseteq \mathcal{U}$ and a (p, q)-bounded mediator S_0, and linkage L', such that $\mathcal{S} = (C, \{S_0, S_1, \ldots, S_n\}, L')$ is equivalent to $\mathcal{G}$. An upper bound on the complexity of deciding this, and constructing a mediator if there is one, is doubly exponential time over the size of $p, q, \mathcal{G}$ and $\mathcal{U}$.* $\square$

We expect that the complexity bound can be refined, but this remains open at the time of writing. More generally, we conjecture that a decidability result and a complexity upper bound can be obtained for a generalization of the above theorem, in which the bounds p, q do not need to be mentioned. In particular, we believe that based on $\mathcal{G}$ and $\mathcal{U}$ there exist p_0, q_0 having the property that if there is a (p, q)-bounded mediator for any p, q, then there is a (p_0, q_0)-bounded mediator.

From Infinite to Finite: the Case Tree. The proof of Theorem 1 is based on a technique that instead of reasoning over (the infinitely many) concrete values in *Dom*, allows us to reason over a finite, bounded set of *symbolic values*. The technique for achieving this reduction is inspired by an approach taken in [8]. Intuitively, part of the construction consists in creating "symbolic images" of most of the constructs that we currently have for concrete values. For example, corresponding to a concrete world state $\mathcal{I}$ we will have a symbolic world state $\widehat{\mathcal{I}}$.

[4] We feel that the results obtained here are themselves quite informative and non-trivial to demonstrate, and can also help show the way towards the development of less restrictive analogs.

In particular, given a (concrete) execution tree $\mathcal{T}$ for some system $\mathcal{S}$ of services, which has infinite branching, it will turn out that the corresponding symbolic execution tree $\widehat{\mathcal{T}}$ will have a strong (homomorphic) relationship to $\mathcal{T}$, but have finitely bounded branching. In general, results that hold in the concrete realm will have analogs in the symbolic realm. The details of the reduction can be found in [3].

Characterization of Composition Synthesis in PDL. To complete the proof of Theorem 1 we outline how the composition synthesis problem can be characterized by means of a Proportional Dynamic Logic formula (PDL). For the necessary details about PDL, we refer to [7]. Intuitively, the PDL formula we construct consists of *(i)* a general part imposing structural constraints on the model, *(ii)* a description of the initial state of each of the service, the goal, and the mediator, and *(iii)* a characterization of what happens every time an action is performed. The only part of the execution that is left unspecified by the PDL formula is the execution of the mediator to be synthesized. Since the execution of the mediator is characterized by which messages are sent to which component services (and consequently, also by which messages are received in response), the PDL formula contains suitable parts that "guess" such messages, including their receiver. In each model of the formula, such a guess will be fixed, and thus a model will correspond to the specification of a mediator realizing the composition (see [3] for more details). Hence, starting from a model of the PDL formula, we are able to construct a mediator.

4 Conclusions

In this paper we have presented `Colombo`, a framework in which web services are characterized in terms of *(i)* message exchanges, *(ii)* data flow, and *(iii)* effects on the real world.

The model is very rich with respect to current approaches, and how outlined in previous sections, it allowed us to develop a novel techniques, based on case tree building and on an encoding in PDL, for computing the composition of web services under certain assumptions.

5 Acknowledgement

The authors would like to thank Jianwen Su and the members of the SWSL working group for valuable discussions. This work has been mainly supported by the INTEROP Network of Excellence (IST-508011).

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services. Concepts, Architectures and Applications.* Springer, 2004.

2. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of e-Services that Export their Behavior. In *Proc. of ICSOC 2003*.
3. D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic Composition of Transition-based Semantic Web Services with Messaging. Tech. Rep. DIS, 2005.
4. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proc. of WWW 2003*.
5. G. De Giacomo and M. Mecella. Service Composition. Technologies, Methods and Tools for Synthesis and Orchestration of Composite Services and Processes. Tutorial at ICSOC 2004.
6. B. Grosof, M. Gruninger, M. Kifer, D. Martin, D. McGuinness, B. Parsia, T. Payne, and A. Tate. Semantic Web Services Language Requirements. `http://www.daml.org/services/swsl/`, 2004.
7. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.
8. R. Hull and J. Su. Domain Independence and the Relational Calculus. *Acta Informatica*, 31(6):513–524, 1994.
9. The DAML-S Coalition. Bringing Semantics to Web Services: The OWL-S Approach. In *Proc. of SWSWPC 2004*.
10. S. McIlraith, T. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46 – 53, 2001.
11. P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *Proc. of ISWC 2004*.