Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Università degli Studi di Roma "La Sapienza"

Automatic Composition of Transition-based Semantic Web Services with Messaging

Daniela Berardi
Diego Calvanese
Giuseppe De Giacomo
Richard Hull
Massimo Mecella

Technical Report n. 6
2005

Aracne

I *Technical Reports* del Dipartimento di Informatica e Sistemistica "Antonio Ruberti" svolgono la funzione di divulgare tempestivamente, in forma definitiva o provvisoria, i risultati di ricerche scientifiche originali. Per ciascuna pubblicazione vengono soddisfatti gli obblighi previsti dall'art. 1 del D.L.L. 31.8.1945, n. 660 e successive modifiche.
Copie della presente pubblicazione possono essere richieste alla Redazione.

Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Università degli Studi di Roma "La Sapienza"
Via Eudossiana, 18 - 00184 Roma
Via Buonarroti, 12 - 00185 Roma
Via Salaria, 113 - 00198 Roma
www.dis.uniroma1.it

www.aracneeditrice.it
info@aracneeditrice.it

Redazione:
00173 Roma
via Raffaele Garofalo, 133 A/B
06 93781065
telefax 72678427

ISBN 88–548–0121-6

I edizione: giugno 2005

Finito di stampare nel mese di giugno del 2005
dalla tipografia « Grafica Editrice Romana S.r.l. » di Roma
per conto della « Aracne editrice S.r.l. » di Roma
Printed in Italy

Automatic Composition of Transition-based Semantic Web Services with Messaging

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

[1]*Università di Roma "La Sapienza"*,
<lastname>@dis.uniroma1.it

[2]*Libera Università di Bolzano/Bozen*
calvanese@inf.unibz.it

[3]*Bell Labs, Lucent Technologies*,
hull@lucent.com

Abstract: In this paper we present Colombo, a framework in which web services are characterized in terms of *(i)* the atomic processes (i.e., operations) they can perform; *(ii)* their impact on the "real world" (modeled as a relational database); *(iii)* their transition-based behavior; and *(iv)* the messages they can send and receive (from/to other web services and "human" clients). As such, Colombo combines key elements from the standards and research literature on (semantic) web services. Using Colombo, we study the problem of automatic service composition (synthesis) and devise a sound, complete and terminating algorithm for building a composite service. Specifically, the paper develops *(i)* a technique for handling the data, which ranges over an infinite domain, in a finite, symbolic way, and *(ii)* a technique to automatically synthesize composite web services, based on Propositional Dynamic Logic.

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.

SOC poses many challenging research issues, the most hyped one being *web service composition*. Web service *composition* addresses the situation when a client request cannot be satisfied by any available service, but by suitably combining "parts of" available 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. Orchestration has been widely addressed by other research areas, and most of the work on service orchestration is based on research in workflows.

In this paper we address the problem of automatic composition synthesis of web services. Specifically, we introduce an abstract model, 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, referred to as world schema. This is similar to the family of "fluents" found in semantic web services models such as OWL-S [15], and more generally, found in situation calculii [17]. *(ii)* Atomic processes (i.e., operations), 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 [3], BPEL4WS [2]) and some formal investigations (e.g., [6, 10]). *(iv)* The behavior of web services (which may involve multiple atomic processes and message-passing activities) is specified using finite state transition system, in the spirit of [5, 6, 10]. The first three elements parallel in several respects the core elements of the emerging SWSL (Semantic Web Service Language) ontology for semantic web services [11]. The fourth element provides an abstract approach to formally model the internal process model of a web service, also reflected as an option in the SWSL ontology.

We also assume that: *(v)* 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. (The conditions in atomic process conditional effects are based on both the world state and the parameter values used to invoke the process.) (*vi*) Finally, we introduce a simple form of integrity constraints on the world state.

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 client behavior can be abstractly represented as a transition system.

In order to address the problem of automatic web service composition, we introduce the notion of "goal service", denoting the behavior of a desired composite service: it is specified as a transition-based web service, that interacts with a client and invokes atomic processes. Our challenge is to build a mediator, which uses messages to interact with pre-existing web services (e.g., in an extended UDDI directory), such that the overall behavior of the mediated system faithfully simulates the behavior of the goal service.

The contribution of this paper is multifold: *(i)* `Colombo` unifies and extends the most important frameworks for services and service composition; *(ii)* it presents a technique to reduce infinite data value to finite symbolic data; *(iii)* it exploits and

extends techniques based on Propositional Dynamic Logic to automatically synthesize a composite service (see [5]), under certain assumptions (and we refer to this as $\texttt{Colombo}^{\mathrm{k,b}}$; *(iv)* it provides an upper bound on the complexity of this problem. To the best of our knowledge, the work reported in this paper is the first one proposing an algorithm for web service composition where web services are described in terms of *(i)* atomic processes, *(ii)* transition-based process models, *(iii)* their impact on a database representing the "real world", and *(iv)* message-based communication. As stated in [13], Service Oriented Computing can play a major role in transaction-based data management systems, since web services can be exploited to access and filter data. The framework developed in this paper shows the feasibility of such an idea.

BPEL4WS [2] allows for (manually) specifying the coordination among multiple web services, expressed in WSDL. The data manipulation internal to web services is based on a "blackboard approach", i.e., a set of variables that are shared within each orchestration instance. Thus, on the one hand BPEL4WS provides constructs for dealing with data flow, but on the other hand, it has no notion of world state.

OWL-S [15] is an ontology language for describing semantic web services, in terms of their inputs, outputs, preconditions and (possibly conditional) effects, and of their process model. On the one hand OWL-S allows for capturing the notion of world state as a set of fluents, but on the other hand it is not clear how to deal with data flow (within the process model).

Several works on automatic composition of OWL-S services exists, e.g., [16, 18]. Most results are based on the idea of *sequentially* composing the available web services, which are considered as black boxes, and hence atomically executed. Such an approach to composition is tightly related to Classical Planning in AI. Consequently, most goals express conditions on the real world, that characterize the situation to be reached: therefore, the automatically devised composition can be exploited only *once*, by the client that has specified the goal. Conversely, in `Colombo` the goal is a specification of the *transition system* characterizing the process of a desired composite web service. Thus, it can be *re-used* by several clients that wants to execute that web service.

`Colombo` extends the Roman model, presented in [5], mainly by introducing data and communication capabilities based on messages. The level of abstraction taken in [5] focuses on (deterministic, atomic) actions, therefore, the transition system representing web service behavior is deterministic. Also, all the interactions are carried out through action invocation, instead of message passing. Finally, in [5] there is no difference between the transition system representing the client behavior and the one specifying the goal, as it is in `Colombo`.

`Colombo` has its root also in the Conversation model, presented in [6, 10], extending it to deal with data and atomic processes. Web services are modeled as Mealy machines (equipped with a queue) and exchange sequence of messages of given types (called conversations) according to a predefined set of channels. It is shown how to synthesize web services as Mealy machines whose conversations (across a given set of channels) are compliant with a given specification. In [10] an extension of the framework is proposed where services are specified as guarded automata, having local XML variables in order to deal with data semantics.

In [19] web services (and the client) are represented as possibly non-deterministic

transition systems, communicating through messaging, and composition is achieved exploiting advanced model cheking techniques. However, a limited support for data is present and there is no notion of local store. It would be interesting to apply our techiques for finitely handling data ranging an infinite domain to their framework, in order to provide an extension to it.

Finally, it is interesting to mention the work in [8], where the authors focus on data-driven services, characterized by a relational database and a tree of web pages. In such a framework, the authors study the automatic verification of properties of a single service, which are defined both in a linear and in a branching time setting.

The rest of the paper is organized as follows. Section 2 illustrates `Colombo` with an example. Section 3 introduces the formal concepts of `Colombo`. In Section 4 the problem of web service composition is formally stated and an upper bound on its complexity is provided. Section 5 shows our technique for handling the data, which ranges over an infinite domain, in a finite, symbolic way. Section 6 presents our tecnhique to automatically synthesize composite web services in `Colombo` based on Propositional Dynamic Logic. Section 7 concludes the paper and highlights future work. In the appendices, technical results are provided.

2 An Example

In this section, we illustrate `Colombo` and give an intuition of our automatic web service composition technique by means of an example involving web services that manage inventories, payment by credit or prepaid card, request shipments, and check shipment status.

The world schema is constituted by four relations, defined over *(i)* the boolean domain $Bool$, *(ii)* an infinite set of uninterpreted elements $Dom_{=}$ (on which only the equality relation is defined) denoted by alphanumeric strings, and *(iii)* an infinite densely ordered set $Dom_{\leq}$, denoted by numbers. An instance of the world schema is shown in Figure 1. 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.

Figure 2 shows the alphabet $\mathcal{A}$ of the atomic processes, that are invoked by the available web services, and are used in the goal service specification. Intuitively, $\mathcal{A}$ represents the common understanding on an agreed upon reference alphabet/semantics cooperating web services should share [7]. For succinctness we use a pidgen 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. Throughout the paper, 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

Acconts

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
...	...	...	...	...

Figure 1: World Schema Instance

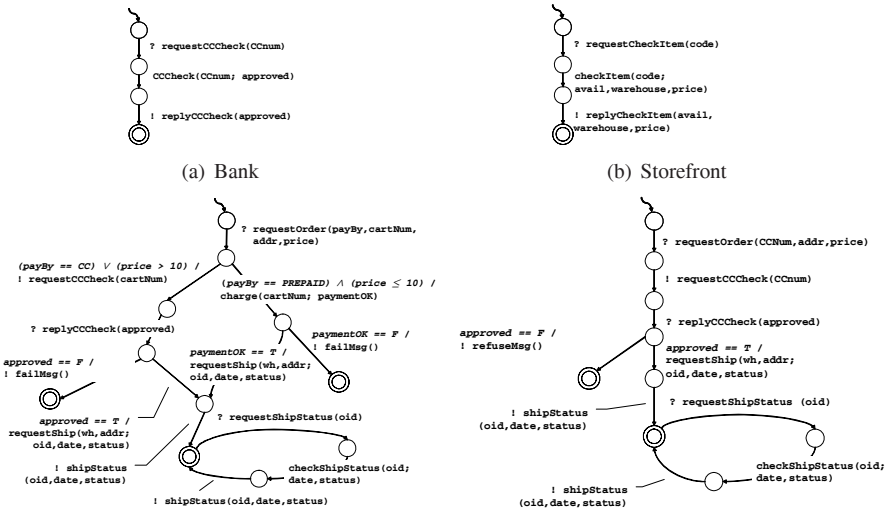
about the states of the world before and after an effect has occurred. Finally, the access function $f_j^R(\langle a_1, \ldots, a_n \rangle)$ (see Section 3) is used to fetch the $n + j$-th element of the tuple in R identified by the key $\langle a_1, \ldots, a_n \rangle$ (i.e., the j-th element of the tuple after the key).

Figure 3 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; `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 process, which are not shown in Figure 2. In the figure, transitions concerning messages are labeled with an operation to transmit or to read a message, by prefixing the message with `!` or `?`, respectively.

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, the set $Bool$ of booleans, the set $Dom_=$ of alphanumeric strings, and the set $Dom_\leq$ of numbers), is used to store parameters values of received messages that have been read and need to be processed during the execution of the web service. *(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) `CCnum` and another for sending messages (of type) `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,

```
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 ¬f_1^Accounts(c) then
   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 and wh:=f_2^Inventory(c) and p:=f_3^Inventory(c)
    and either no-op on Inventory or
    modify Inventory(c;F, -, -)
  if ¬f_1^Inventory(c) or f_1^Inventory(c) = ω
   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 ¬f_1^PrePaid(c) then paymentOK:= 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:
    ∃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_≤; % resp. shipping date & status
   effects:
    if f_1^Shipment(oid) = ω then no-op and s,d uninit
    else s:=f_3^Shipment(oid) and d:=f_4^Shipment(oid)
```

Figure 2: Alphabet of Atomic Processes



(a) Bank (b) Storefront

(c) Next Generation Warehouse (d) Standard Warehouse

Figure 3: Transition systems of the available services

for storing messages (of type) `CCnum`.

Figure 4 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 our composition technique, we are able to automatically construct a mediator such as S_0 shown in Figure 5. 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 notion of *linkage* that will be introduced in the following sections. 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 are 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 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 the message (carrying the item's code), executes the atomic process `checkItem(code)` by accessing the tuple of relation `Accounts` having as key the given code: at this point, the information on the ware-

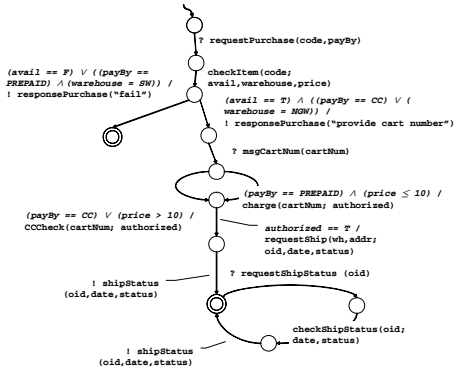


Figure 4: Transition system of the goal service

house 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., if `avail == T`) S_0 transmits a `responsePurchase(''provide cart num'')` to the client, asking her for the card number, and the interactions go on.

3 The Model

This section provides an overview of the formal model used in our investigation, focusing on $\texttt{Colombo}^{\mathrm{k,b}}$.

Model of the "real world". A *world (database) schema* 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 associated with $Bool$, $Dom_=$ or $Dom_\leq$. A *world instance* is a database instance over $\mathcal{W}$.

We allow for constraints over relations (see below for notion of "accessible term"). A *key-accessible constraint* is an expression of the form $\varphi = \forall x_1, \ldots, x_n(\psi)$, where the x_i's are distinct variables, and where ψ is a boolean expression over atoms over accessible terms over a set of constants and variables $\{x_1, \ldots, x_n\}$. A world instance $\mathcal{I}$ *satisfies* this constraint if for all assignments α for variables $x_1, \ldots, x_n$, formula ψ is true in $\mathcal{I}$ when interpreted according to α.

Atomic Processes. Atomic processes in Colombo, inspired by OWL-S atomic processes, may access/modify one or more of relations in the world schema. In typical applications a given relation of the world schema may be accessible by just one web