Automatic Composition of Web Services in `Colombo`*

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

[1] *Università di Roma "La Sapienza"*, [2] *Libera Università di Bolzano/Bozen*
`lastname@dis.uniroma1.it` `calvanese@inf.unibz.it`
[3] *Bell Labs, Lucent Technologies*,
`hull@lucent.com`

Abstract. In this paper we present `Colombo`, a framework 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, `Colombo` is the first attempt to address all of them in a single coherent framework. Dealing with all these aspects together, in fact, is particularly challenging, especially with respect to the problem of automatic web service composition. In this paper, we introduce novel techniques to synthesize composite web services in this setting, under certain simplifying assumptions.

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 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. Orchestration has been widely addressed by other research areas, and most of the work on service orchestration is based on research in workflows [5].

In this paper we address the problem of automatic composition synthesis of web services. Specifically, we introduce an abstract model, called `Colombo`, that combines

* This paper is an extended abstract of [3]. The authors would like to thank Maurizio Lenzerini, Jianwen Su and the members of the SWSL working group for valuable discussions. This work has been supported by MIUR through the "FIRB 2001" project *MAIS*, and "Società dell'Informazione". It has been also supported by the European projects SEWASIE (IST-2001-34825), EU-PUBLI.com (IST-2001-35217) and INTEROP Network of Excellence (IST-508011).

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.

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 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.

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, thus differentiating with most approaches to automatic composition (e.g. [10]), in which the goal is expressed as a condition on a situation in the real world.

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 [2]), under certain assumptions (and we refer to this as $\texttt{Colombo}^{k,b}$); *(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 techniques for web service composition where web services are described in such a rich framework.

The rest of the paper is organized as follows. Section 2 illustrates `Colombo` with an example. Section 3 formally states the problem of web service composition, and gives an intuition of our automatic composition techniques. Section 4 concludes the paper.

2 The `Colombo` Framework

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

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. 1. World Schema Instance

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 1 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 2. 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 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 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, 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`

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. 2. Alphabet $\mathcal{A}$ of Atomic Processes

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 2. Finally, in Figure 3, transitions concerning message exchanges 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, $Bool$, $Dom_=$, 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.

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

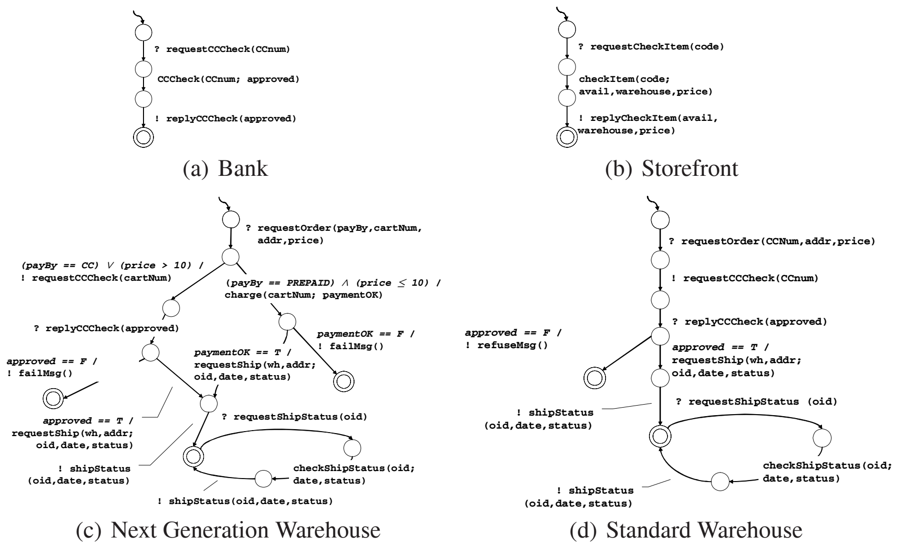

Fig. 3. Transition systems of the available services

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 techniques, 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 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

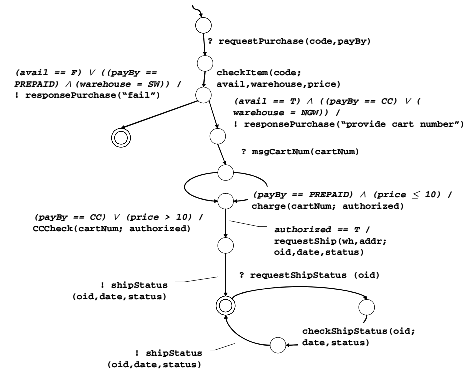

Fig. 4. Transition system of the goal service

`responsePurchase(''provide cart num'')` to the client, asking her for the card number, and the interactions go on.

3 The Composition Synthesis Problem Statement

In this section we formally address the composition synthesis problem.

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 sythesis problem remains open for most cases of the general `Colombo` framework. In the context of `Colombo`$^{\mathrm{k,b}}$ we can achieve decidability and complexity results under the assumptions[1] 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.

[1] 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.

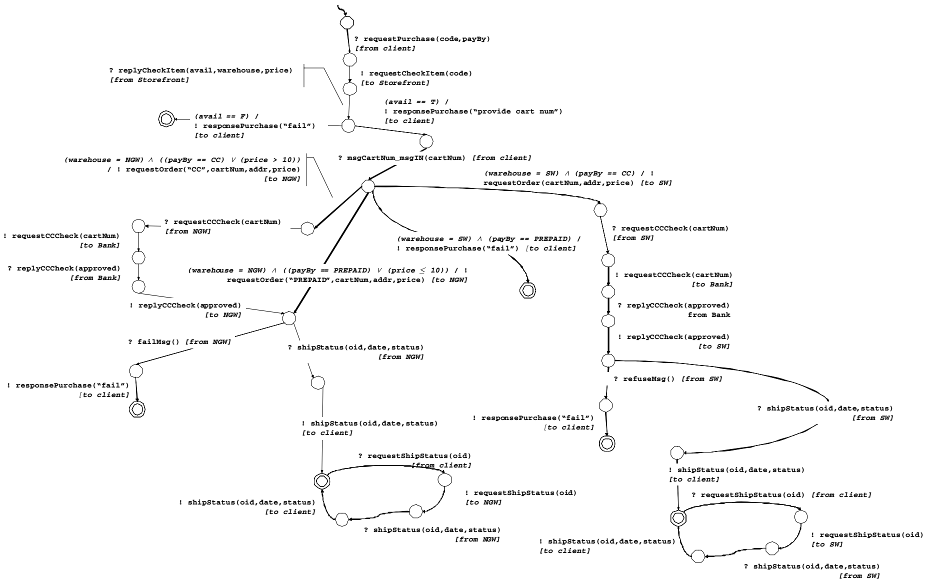

Fig. 5. Transition system of the mediator

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 $\mathcal{U}$.* □

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 $\widehat{\mathcal{I}}$. 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. We have presented novel techniques, based on case tree building and on an encoding in PDL, for computing the composition of web services under certain assumptions ($\mathtt{Colombo}^{k,b}$).

A problem related to composition synthesis is that of choreography synthesis, which consists in selecting a set of available services and in constructing a set of links among them, in order to realize a goal system. Such a problem can, in fact, be seen as a a specialized version of composition synthesis, hence our synthesis techniques apply [3].

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*.