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"*
berardi@dis.uniroma1.it
degiacomo@dis.uniroma1.it
mecella@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. SOC poses many challenging research issues, the most hyped one being *web service composition*. 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.

Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

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, 14], 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], WS-BPEL [2]) and some formal investigations (e.g., [6, 9]). *(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, 9]. The first three elements parallel in several respects the core elements of the emerging Semantic Web Services Framework (SWSF [10]). The fourth element provides an abstract approach to formally model the internal process of a web service, also reflected as an option in SWSF.

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) and the client, 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 (see [5]) based on Propositional Dynamic Logic to automatically synthesize a composite service, 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 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 [12], 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.

WS-BPEL [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 [14] 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., [15, 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 roots also in the Conversation model, presented in [6, 9], 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 [9] 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 [4], technical results are provided.

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

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 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. 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 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).

```
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
  % 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 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(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:=$f_4^{Shipment}(oid)$
```

Figure 2: Alphabet of Atomic Processes

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

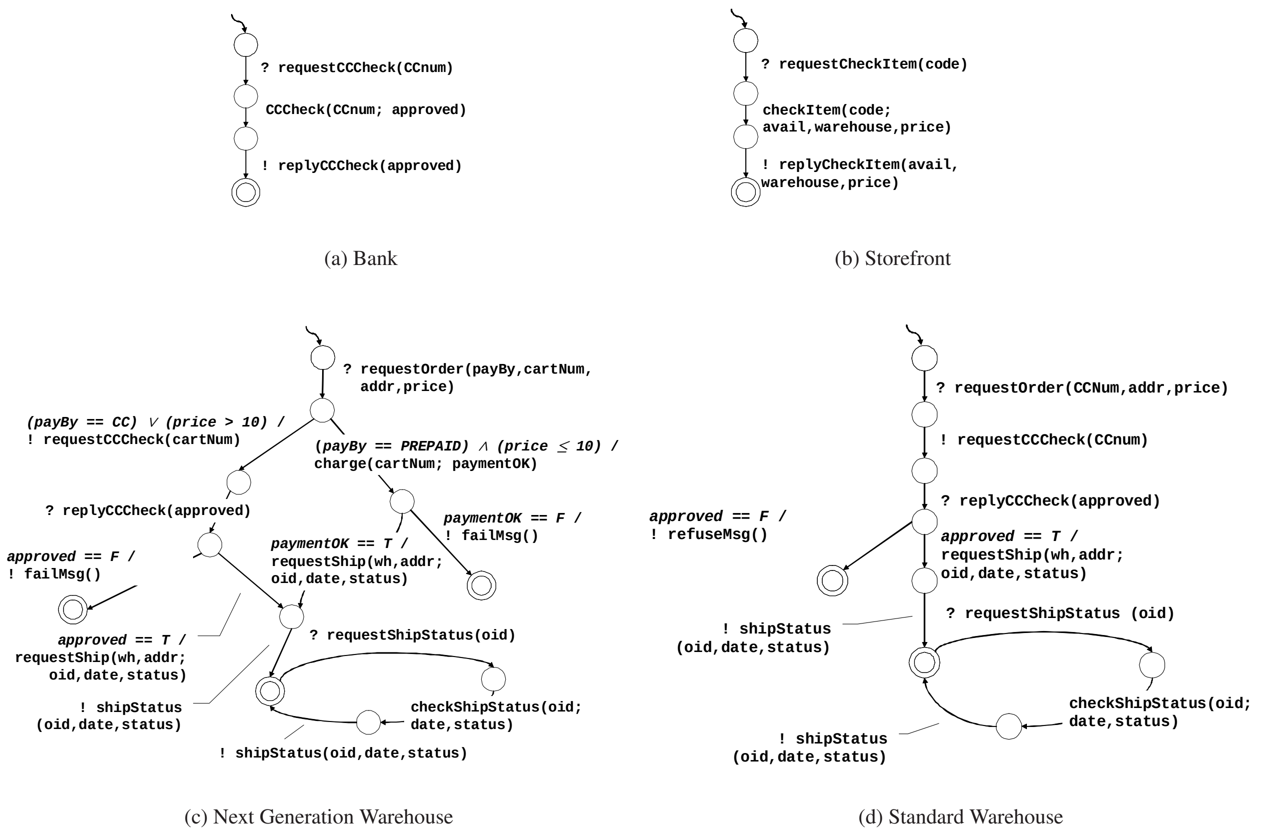
? requestCCCheck(CCnum)
CCCheck(CCnum; approved)
! replyCCCheck(approved)
(a) Bank
? requestCheckItem(code)
checkItem(code; avail,warehouse,price)
! replyCheckItem(avail, warehouse,price)
(b) Storefront
? requestOrder(payBy,cartNum, addr,price)
(payBy == CC) ∨ (price > 10) / ! requestCCCheck(cartNum)
(payBy == PREPAID) ∧ (price ≤ 10) / charge(cartNum; paymentOK)
? replyCCCheck(approved)
paymentOK == F / ! failMsg()
paymentOK == T / requestShip(wh,addr; oid,date,status)
approved == F / ! failMsg()
approved == T / requestShip(wh,addr; oid,date,status)
? requestShipStatus(oid)
! shipStatus (oid,date,status)
checkShipStatus(oid; date,status)
! shipStatus(oid,date,status)
(c) Next Generation Warehouse
? requestOrder(CCNum,addr,price)
! requestCCCheck(CCnum)
? replyCCCheck(approved)
approved == F / ! refuseMsg()
approved == T / requestShip(wh,addr; oid,date,status)
? requestShipStatus (oid)
! shipStatus (oid,date,status)
checkShipStatus(oid; date,status)
! shipStatus (oid,date,status)
(d) Standard Warehouse

Figure 3: Transition systems of the available services

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, 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

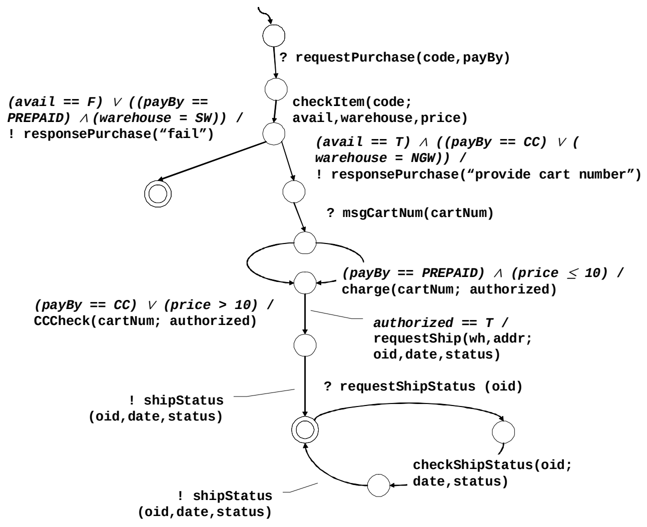
? requestPurchase(code,payBy)
(avail == F) ∨ ((payBy == PREPAID) ∧ (warehouse = SW)) / ! responsePurchase("fail")
checkItem(code; avail,warehouse,price)
(avail == T) ∧ ((payBy == CC) ∨ (warehouse = NGW)) / ! responsePurchase("provide cart number")
? msgCartNum(cartNum)
(payBy == PREPAID) ∧ (price ≤ 10) / 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)

Figure 4: Transition system of the goal service

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 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., 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 `Colombo`^{k,b}. More details can be found in [4].

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 the notion of "accessible term", which however has an intuitive meaning). 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 service or by several web services, or by all web services. Furthermore, when executing, the atomic processes can make a finitely bounded nondeterministic choice. This can be viewed as indicating that the world instance holds only partial information about the state actually observable by the atomic processes.

The syntax for describing conditions, integrity constraints, and for describing the local stores of web services, is based on the use of symbols denoting *constants* (taken from $Dom = Bool \cup Dom_= \cup Dom_\leq$) and *variables*. (These variables are typed as *Bool*, *Eq*, *Leq*.) At a given point in time during execution of a web service, there may be an *assignment* α of variables (e.g., in the local store of some web service) to elements of *Dom*. For a variable v, α may assign a value from *Dom*, or ω (null value).

Notation: Let $R(A_1, \ldots, A_n; B_1, \ldots, B_m)$ be a relation in the world schema $\mathcal{W}$. We define a family of n-ary functions f_j^R for $j \in [1..m]$, as follows. Let $\mathcal{I}$ be an instance over $\mathcal{W}$, and $a_1, \ldots, a_n$ be (not necessarily distinct) elements of *Dom*. Then the *value* of $f_j^R(a_1, \ldots, a_n)$ in $\mathcal{I}$ is defined to be either *(i)* the null value ω if $\langle a_1, \ldots, a_n \rangle \notin \pi_{\{A_1,\ldots,A_n\}}(\mathcal{I}(R))$, or *(ii)* it is equal to the unique b_j's where $\langle a_1, \ldots, a_n, b_1, \ldots, b_n \rangle \in \mathcal{I}(R)$. We refer to the functions f_j^R as the *access functions*.

Given constants C and variables V, the set of *accessible terms* over C, V is defined recursively to include all terms contructed using C, V and the f_j^R functions. An *atom* over C, V is an expression of form *(i)* $init(t)$, *(ii)* $t = t'$, *(iii)* $t < t'$, or *(iv)* $t > t'$, where t, t' are accessible terms. Atoms and propositional formulas constructed using them are given a

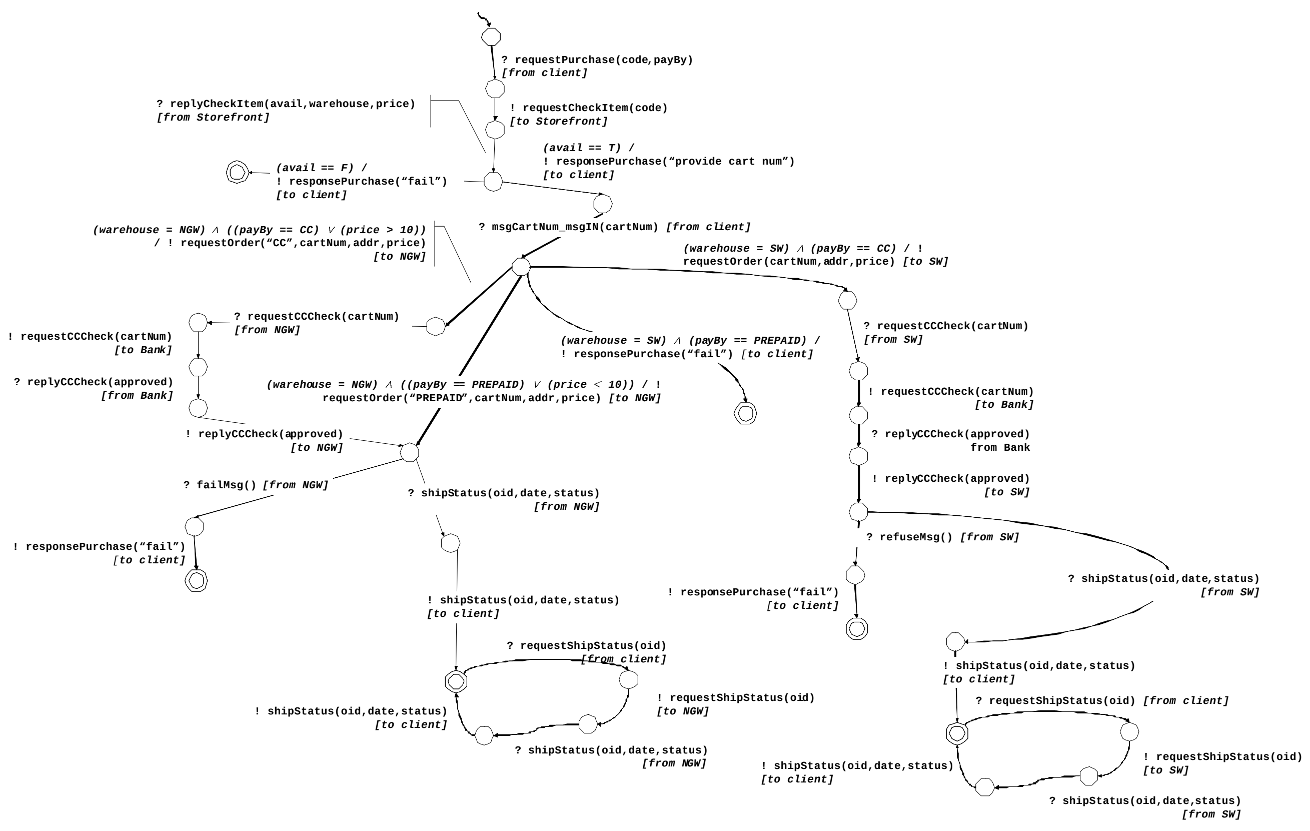
? requestPurchase(code,payBy) [from client]
? replyCheckItem(avail,warehouse,price) [from Storefront]
! requestCheckItem(code) [to Storefront]
(avail == T) / ! responsePurchase("provide cart num") [to client]
(avail == F) / ! responsePurchase("fail") [to client]
(warehouse = NGW) ∧ ((payBy == CC) ∨ (price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

Figure 5: Transition system of the mediator

truth value under an assignment α in the usual manner.

Definition: 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. 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*). Condition c is a boolean expression over atoms over accessible terms over some family of constants and the input variables $u_1, \ldots, u_n$.

An effect $e \in E$ is a pair (es, ev), where: es (the *effect on the world*) is a set of expressions having the forms (i) $insert\ R(t_1, \ldots, t_k; s_1, \ldots, s_l)$; (ii) $delete\ R(t_1, \ldots, t_k)$; or (iii) $modify\ R(t_1, \ldots, t_k; r_1, \ldots, r_l)$; where the t_i's and s_j's are accessible terms over some set of constants and $u_1, \ldots, u_n$, and where each r_j is either an accessible term or the special symbol '$-$' (denoting that that position of the identified tuple in R should be unchanged); and ev (*effect on outputs*) is a set of expressions of the form (iv) $v_j := t$, where $j \in [1..m]$ and t is an accessible term over some set of constants and $u_1, \ldots, u_n$; or (v) $v_j := \omega$, where $j \in [1..m]$ (There must be exactly one expression for each v_j.)

The definition of the *semantics* of an atomic process execution is relatively straightforward – based on the values for the input variables and the current world instance[1], if a conditional effect (c, E) has true condition then one element $e \in E$ is nondeterministically chosen. If the application of e on the world instance satisfies the global constraints Σ then e is used to modify the world instance and to determine the values of the output variables.

We write $(\alpha, \mathcal{I}) \vdash_{p(r_1,\ldots,r_n;v_1,\ldots,v_m)} (\alpha', \mathcal{I}')$ over $\mathcal{W}, \Sigma$, if the pair $(\alpha', \mathcal{I}')$ is one of the possible pairs resulting from the execution of an atomic process p, with inputs r_i's and outputs v_j's, as described above. The *trace* of this move is the syntactic object $p(c_1, \ldots, c_n; d_1, \ldots, d_m)$ where c_i is the domain value identified by $\alpha(r_i)$ (α is the identity on elements of Dom, see [4], and where d_j is the domain value $\alpha'(v_j)$.

Messages, Ports, and Links. A *message type* has a name m and a signature of form $\langle d_1, \ldots, d_n \rangle$, where $n \geq 0$ and each $d_i \in \{Bool, Eq, Leq\}$.

In Colombo, a (*service*) *port signature* of a service S, denoted Port or Port$S$, is a set P of pairs having the form $(m, \texttt{in})$ or $(m, \texttt{out})$, where the m's are message types, in and out denote the "direction" of the message flow and each pair in P has a distinct message type. Let $\mathcal{F} = \{S_1, \ldots, S_n\}$ be a family of services (with or without one client) having associated port signatures $\{P_1, \ldots, P_n\}$. A *link* for $\mathcal{F}$ is a tuple of the form (S_i, m, S_j, n) where $(m, \texttt{out}) \in P_i$, $(n, \texttt{in}) \in P_j$, and m, n have identical sig-

[1]Intuitively, it depends on α, $\mathcal{I}$, and Σ, and results in an assignment α' and world state $\mathcal{I}'$.

natures. (It can occur that $i = j$, although perhaps not typical in practice.) 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. It is not required that every port of a service S occur in L.

In this paper we will 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.

Local & Queue Store, Transmit, Read, Has-seen. Let S be a non-client web service. The *local store* $\texttt{LStore}_S$ of S is a finite set of typed variables. For each incoming port $(m, \texttt{in})$ of S we assume that there is a distinguished boolean variable π_m in $\texttt{LStore}_S$, which is set true if there is at least one message in the queue. Also, each non-client service S has a *queue store* QStore, used to hold the parameter values of incoming messages, which can be thought of as being held by a queue. Wlog, we focus on queues of length 1.

As illustrated in Section 2, for passing messages between services we have two basic operations: *transmit* and *read*, denoted using $!m$ and $?m$, respectively. A transmit is based on an explicit step of the sending service, and is reflected as an asynchronous *receive* at the receiving service. In $\texttt{Colombo}^{\mathrm{k,b}}$, a transmit will block if the corresponding queue of the receiver is full. (An alternative is to view the send as failed and let the sending service continue with other activities.) Similarly, in $\texttt{Colombo}^{\mathrm{k,b}}$ the read operation will block until there is something in the appropriate queue (although other semantics are possible).

With regards to a client service C in $\texttt{Colombo}^{\mathrm{k,b}}$, we bundle the receive and the read as just *receive*. We do not model the local or queue stores of clients, but maintain simply a unary relation, denoted HasSeen or $\texttt{HasSeen}_C$, which holds elements of *Dom*. Intuitively, at a given time in an execution of C, $\texttt{HasSeen}_C$ will include all of constants appearing in service specification ($\texttt{Constants}_C$), and also all domain elements that occur in messages that have been transmitted to C.

Abstract Model of Internal Service Process. In $\texttt{Colombo}^{\mathrm{k,b}}$, a *guarded automaton* is a tuple $(Q, \delta, F, \texttt{LStore}, \texttt{QStore})$ where Q is a finite set of *states*, $F \subset Q$ is a set of *final states*, and LStore (QStore) is the local (queue) store. The *transition function* δ contains tuples (s, c, μ, s') where $s, s' \in Q$, c is a condition over $\texttt{LStore} \cup \texttt{QStore}$ (no access to the world instance), and μ is either a send, a read, or an atomic process invocation. The non-client services have *deterministic* signature, i.e., it is assumed that for each state in Q, store contents and a world instance, at most one out-going transition can be labeled with a condition that evaluates to true. The *Guarded Automaton signature* of (non-client) service S is denoted $\texttt{GA}(S)$.

In $\texttt{Colombo}^{\mathrm{k,b}}$, we assume for a client C that in $\texttt{GA}(C)$ there are exactly two states, called ReadyToTransmit and ReadyToRead, where the first is the start state and also the final state. In $\texttt{Colombo}^{\mathrm{k,b}}$ the client will toggle between the two states. We use the "has-seen" set HasSeen as an abstract representation of constants that the client has seen so far. The clients are *non-deterministic*, in terms of the message they choose to read, and in terms of the values they transmit.

The *moves-to* relation $\vdash$ will hold between pairs of the form $(\texttt{id}^S, \mathcal{I}), (\texttt{id}^{S'}, \mathcal{I}')$, where $\texttt{id}^S, \texttt{id}^{S'}$ are *instantaneous descriptions* (*id*'s) for S and $\mathcal{I}, \mathcal{I}'$ are world instances. This is defined in the usual way. The *trace* of a pair $(\texttt{id}^S, \mathcal{I}), (\texttt{id}^{S'}, \mathcal{I}')$ (where $(\texttt{id}^S, \mathcal{I}) \vdash_S (\texttt{id}^{S'}, \mathcal{I}')$) will provide, intuitively, a grounded record or log of salient aspects of the transition from $(\texttt{id}^S, \mathcal{I})$ to $(\texttt{id}^{S'}, \mathcal{I}')$, including, e.g., what parameter values were input/output from an atomic process invocation, or were received, read or sent.

For clients, an *id* is a pair of form $(s, \texttt{HasSeen})$. The *moves-to* relation and *trace* are defined for clients in the natural manner (see [4] for details).

System Execution and Equivalence. In general we focus on a *system*, which is a triple $\mathcal{S} = (C, \mathcal{F}, L)$, where C is a client, $\mathcal{F} = \{S_1, \ldots, S_n\}$ is a finite family of web services, and L is a linkage for $(C, \mathcal{F})$ (i.e., for $\{C\} \cup \mathcal{F}$).

For this paper we make the assumption of **No External Modifications:** when discussing the execution of one or more services $\mathcal{S}_1, \ldots, \mathcal{S}_k$, we assume that no other systems can modify the relations in the world schema that are accessed by the executions of $\mathcal{S}_1, \ldots, \mathcal{S}_k$.

The notion of (*initial*) *instantaneous description* (*id*) for system $\mathcal{S}$ is defined in a natural fashion to be a tuple $\texttt{id}^{\mathcal{S}} = (\texttt{id}^C, \{\texttt{id}^S \mid S \in \mathcal{F}\})$, based on a generalization of id for individual services. The *moves-to* relation for system $\mathcal{S}$, denoted $\vdash_{\mathcal{S}}$ or $\vdash$, is defined as a natural generalization of $\vdash$ for clients and services. More specifically, we have $(\texttt{id}^{\mathcal{S}}, \mathcal{I}) \vdash (\texttt{id}^{\mathcal{S}'}, \mathcal{I}')$ when (written informally, see [4] for more details)

(i) If a service performs an atomic process or a read, that is the only service that moves. For an atomic process the world instance can change, and for the read it cannot change.

(ii) If a service performs a transmit, then the target of that transmit (according to L) performs a receive in the same move. In this case the world instance cannot change.

In case (i), the *trace* of pair $(\texttt{id}^{\mathcal{S}}, \mathcal{I}) \vdash (\texttt{id}^{\mathcal{S}'}, \mathcal{I}')$ is the trace of the individual service that changed; in case (ii), the trace is the pair $(!m(c_1, \ldots, c_n), ?n(c_1, \ldots, c_n))$ where the $!m$ part is the trace of the sending service and the $?n$ part is the trace of the receiving service.

An *enactment* of $\mathcal{S}$ is a finite sequence $\mathcal{E} = \langle(\texttt{id}_1, \mathcal{I}_1), \ldots, (\texttt{id}_q, \mathcal{I}_q)\rangle$, $q \geq 1$, where (a) $\texttt{id}_1$ is an initial id for $\mathcal{S}$, and (b) $(\texttt{id}_p, \mathcal{I}_p) \vdash (\texttt{id}_{p+1}, \mathcal{I}_{p+1})$ for each $p \in [1..(q-1)]$. The enactment is *successful* if $\texttt{id}_q$ is in a final state of $\texttt{GA}(C)$ and each $\texttt{GA}(S)$.

The notion of execution tree for $\mathcal{S}$ is, intuitively an infinitely branching tree $\mathcal{T}$ that records all possible enactments. The root is not labeled, and all other nodes are labeled by pairs of form $(\mathtt{id}, \mathcal{I})$ where $\mathtt{id}$ is an id of $\mathcal{S}$ and $\mathcal{I}$ a valid world instance. For children of the root, the id is the initial id of $\mathcal{S}$ and $\mathcal{I}$ is arbitrary. An edge $((\mathtt{id}, \mathcal{I}), (\mathtt{id}', \mathcal{I}'))$ is included in the tree if $(\mathtt{id}, \mathcal{I}) \vdash (\mathtt{id}', \mathcal{I}')$; in this case the edge is labeled by $\mathtt{trace}((\mathtt{id}, \mathcal{I}), (\mathtt{id}', \mathcal{I}'))$. A node $(\mathtt{id}, \mathcal{I})$ in the execution tree is *terminating* if $\mathtt{id}$ is in a final state of $\mathtt{GA}(C)$ and each $\mathtt{GA}(S)$.

The *essence* of $\mathcal{T}$, denoted $\mathtt{essence}(\mathcal{T})$, is a collapsing of $\mathcal{T}$, created as follows. The root and its children remain the same. Suppose that v_1 is a node of $\mathcal{T}$ that is also in $\mathtt{essence}(\mathcal{T})$, and let $v_1, \ldots, v_n, v_{n+1}$, $n \geq 1$, be a path, where $\mathtt{trace}(v_i, v_{i+1})$ for each $i \in [1..n]$ involves message transmits or reads not involving the client, and $\mathtt{trace}(v_n, v_{n+1})$ involves an atomic process invocation or a transmit to or from the client. Then include edge (v_1, v_{n+1}) in $\mathtt{essence}(\mathcal{T})$, where v_{n+1} has the same label as in $\mathcal{T}$, and the this edge is labeled with $\mathtt{trace}(v_n, v_{n+1})$.

Note that for a system $\mathcal{S} = (C, \mathcal{F}, L)$ each pair of execution trees $\mathcal{T}$ and $\mathcal{T}'$ of $\mathcal{S}$ are isomorphic, and also $\mathtt{essence}(\mathcal{T})$ and $\mathtt{essence}(\mathcal{T})$ are isomorphic.

Suppose now that world schema $\mathcal{W}$ and global constraints Σ are fixed, and let $\mathcal{A}$ be an alphabet of atomic processes. Let $\mathcal{S} = (C, \{S \mid S \in \mathcal{F}\}, L)$ and $\mathcal{S}' = (C, \{S \mid S \in \mathcal{F}'\}, L')$ be two systems over $\mathcal{W}, \Sigma, \mathcal{A}$, and over the same client C.

We say that $\mathcal{S}$ is *equivalent* to $\mathcal{S}'$, denoted $\mathcal{S} \equiv \mathcal{S}'$ if for some (any) execution trees $\mathcal{T}, \mathcal{T}'$ of $\mathcal{S}, \mathcal{S}'$, respectively, we have that $\mathtt{essence}(\mathcal{T})$ is isomorphic to $\mathtt{essence}(\mathcal{T}')$. Intuitively, this means that relative to what is observable in terms of client messaging and atomic process invocations (and their effects), the behaviors of $\mathcal{S}$ and $\mathcal{S}'$ are indistinguishable.

4 The Composition Synthesis Problem Statement

In this section we formally define the composition synthesis problem, and also a specialized version of this called the choreography synthesis problem. We then state our main results, giving decidability and complexity bounds for composition and choreography synthesis in the restricted context of $\mathtt{Colombo}^{\mathrm{k,b}}$. The proofs for these results are sketched in Sections 5 and 6.

For this section we assume that a world schema $\mathcal{W}$, global constraints Σ, and an alphabet $\mathcal{A}$ of atomic processes are all fixed.

For both synthesis problems, assume that a family of *available* (or pre-defined) services operating over $\mathcal{A}$ is available (e.g., in an extended UDDI directory). We also assume that there is a "desired behavior", described using a specialized system. In particular, a *goal system* is a triple $\mathcal{G} = (C, \{G\}, L)$ where C is a client; G is a web service over alphabet $\mathcal{A}$, called the *goal service*; and L is a linkage involving only C and G.

In the general case, given the 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. The *choreography synthesis problem* is to (a) select a family $S_1, \ldots, S_n$ of services from the pre-existing set, and (b') construct a linkage L' over $C, S_1, \ldots, S_n$ such that $\mathcal{G}$ and $\mathcal{S} = (C, \{S_1, \ldots, S_n\}, L')$ are equivalent.

Decidability of the composition and choreography synthesis problems remains open for most cases of the general $\mathtt{Colombo}$ framework. We describe now a family of restrictions, in the context of $\mathtt{Colombo}^{\mathrm{k,b}}$, under which we can acheive decidability and complexity results for these problems. 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.

Let $\mathcal{G} = (C, \{G\}, L)$ be a goal system. Two key assumptions of the goal system are as follows:

Blocking behavior: (a) For each available service, if a state can be entered by a transition involving a message send, then the service either terminates at that state, or blocks and waits at that state for a message receive. (b) The client initiates by sending a message, and upon message receipt it either halts or sends a message.

Bounded Access: (a) There is a $k > 0$, such that in any enactment of the client C, the number of values that can be sent out is $\leq k +$ the number of values that are recieved by C. (b) For each $p > 0$ there is a $q > 0$ such that in each enactment of $\mathcal{G}$, if at most p new values come from the client, then only q distinct key-based searches can be executed by the atomic process invocations in $\mathcal{G}$.

The first restriction prevents concurrency in our systems, and the second one ensures that 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). Note that in $\mathtt{Colombo}^{\mathrm{k,b}}$, k and b denote the bounded access and the blocking behavior assumptions, respectively.

For the case of composition synthesis, we restrict the form of mediators and linkages that we will look for, as follows:

Strict Mediation: A system $\mathcal{S} = (C, \{S_0, S_1, \ldots, S_n\}, L')$ is *strict mediation* if in L' all messages are either sent by the mediator S_0 or received by the mediator.

We also make a simplifying assumption that essentially blocks services outside of the relevant system(s) from modifying the world state.

Finally, we say that a mediator service is (p, q)-*bounded* if it has at most p guarded automata states and at most q variables in its global store.

Theorem 4.1: Assume that all services are in $\texttt{Colombo}^{\text{k,b}}$, and assume No External Modifications. Let $\mathcal{G} = (C, \{G\}, L)$ be a goal system and $\mathcal{U}$ a finite family of available web services, all of which satisfy Blocking Behavior and Bounded Access. 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' satisfying Strict Mediation, 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 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 are 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.

We now describe how the choreography synthesis problem can be reduced to a special case of the composition synthesis problem. Let $\mathcal{G} = (C, \{G\}, L)$ be a goal system. Suppose that there is a solution $\mathcal{S} = (C, \{S_1, \ldots, S_n\}, L')$ for the choreography synthesis problem. Then we can build a mediator $\mathcal{S}_0$ and Strict Mediation linkage L'' so that (a) S_0 has exactly one state, (b) the local store of S_0 has only variables of the form π_m (which record whether a message of type m has been received), and $\mathcal{S}' = (C, \{S_0, S_1, \ldots, S_n\}, L'')$ is equivalent to $\mathcal{G}$. The converse also holds. Finally, note that the size of the global store of mediator S_0 is bounded by the total number of types of message that can be sent by the family $\mathcal{U}$ of available services.

From these observations and a minor variation on the proof technique of Theorem 4.1 we can obtain the following.

Theorem 4.2: Assume that all services are in $\texttt{Colombo}^{\text{k,b}}$, and assume No External Modifications. Let $\mathcal{G} = (C, \{G\}, L)$ be a goal system and $\mathcal{U}$ a family of available web services, all of which satisfy Blocking Behavior and Bounded World State Access. It is decidable whether there is a set $\{S_1, \ldots, S_n\} \subseteq \mathcal{U}$ and a linkage L' such that $\mathcal{S} = (C, \{S_1, \ldots, S_n\}, L')$ 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 $\mathcal{G}$ and $\mathcal{U}$.

5 From Infinite to Finite: the Case Tree

This section develops a key aspect needed for the proofs of Theorems 4.1 and 4.2, namely, it allows us to reason over a finite universe of domain values, rather than over the infinite universe *Dom*. The essence of the technique is that instead of reasoning over (the infinitely many) concrete values in *Dom*, we reason over a finite, bounded set of *symbolic values*. The technique for achieving this reduction is inspired by an approach taken in [13]. A key enabler for the reduction is the assumption that in $\texttt{Colombo}^{\text{k,b}}$ services, all conditions and data accesses rely on key-based lookups; another enabler is the assumption of Bounded Access.

As part of the construction, we will create "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 symbolic world state $\widehat{\mathcal{I}}$, corresponding to a moves-to relation $\vdash$ in the concrete realm we shall have a moves-to relation $\widehat{\vdash}$ in the symbolic realm, etc. 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.

We assume an infinite set *Symb* of *symbolic values* (disjoint from *Dom*); these will sometimes behave as values, and other times behave as variables.

Let C be a finite set of constants in *Dom* and Y a finite set of symbolic values. Let $\texttt{Atoms}(Y, C)$ be the set of all atoms over Y, C. This includes expressions of the following forms:

1. $\texttt{incorp}(y)$, with intuitive meaning that symbolic value y has been "incorporated" into an enactment;
2. $\texttt{bool}(y)$, $\texttt{eq}(y)$ and $\texttt{leq}(y)$, indicating intuitively the domain type associated with y.
3. $y = T$ and $y = F$ (can be true only if $\texttt{incorp}(y)$ and $\texttt{bool}(y)$).
4. $y = y'$ (can be true only if y and y' "have" been incorporated and "have" the same type).
5. $y < y'$, $y > y'$ (can be true only if $\texttt{leq}(y)$ and $\texttt{leq}(y')$).

An *sv-characterization* (*svc* for short) for Y, C is a maximal consistent conjunction over $\texttt{Atoms}(Y, C)$ and their negations. (Informally, the notion of "consistency" here prevents, e.g., $\texttt{eq}(y)$ and $\texttt{leq}(y)$, $y < y'$ and $y' < y$, etc.) Note that we do not allow any y to "have" the value ω. This is because symbolic values range exclusively over concrete elements of *Dom*.

Let Y, C be fixed, and $\sigma : Y \rightarrow Dom$. Then there is a unique svc $\widehat{\gamma}$ such that $\widehat{\gamma}[\sigma]$ is true. We denote this svc as $\texttt{svc}(\sigma)$. There is a natural equivalence relation $\sim_{Y,C}$ between assignments from Y to *Dom*, defined by $\sigma \sim_{Y,C} \sigma'$ iff for all atoms $a \in \texttt{Atoms}(Y, C)$, $a[\sigma]$ iff $a[\sigma']$. Note that this is equivalent to stating that $\texttt{svc}(\sigma) = \texttt{svc}(\sigma')$.

Conversely, for an svc $\widehat{\gamma}$, it is possible to construct a mapping $\sigma : Y \rightarrow Dom$ such that $\texttt{svc}(\sigma) = \widehat{\gamma}$.

Let Y, C be fixed, where C includes at least all constants occurring in service S. Let $\widehat{\gamma}$ be an svc over Y, C. Then an assignment $\widehat{\alpha} : \texttt{LStore}_S \rightarrow (Y \cup \{T, F, \omega\})$ is *valid* for $\widehat{\gamma}$ if (i) $\widehat{\alpha}(v) \in Y \cup \{\omega\}$ for v's not of form π_m, and $\widehat{\alpha}(\pi_m) \in Bool$ for each variable of form π_m; (ii) $\widehat{\gamma} \models \texttt{incorp}(\widehat{\alpha}(v))$ for each v not of form π_m; and (iii) $\widehat{\gamma} \models \texttt{bool}(\widehat{\alpha}(v))$ iff v is of type $Bool$, and likewise for eq and leq. The notion of assignment $\widehat{\beta} : \texttt{QStore}_S \rightarrow Y \cup \{\omega\}$ being *valid* is defined analogously.

A *symbolic id* of service S is a 4-tuple $\widehat{\texttt{id}} = (s, \widehat{\alpha}, \widehat{\beta}, \widehat{\gamma})$ where $\widehat{\gamma}$ is an svc, and $\widehat{\alpha}$, $\widehat{\beta}$ are valid assignments over LStore and QStore for $\widehat{\gamma}$.

We now turn to symbolic tuples, relational instances, and world states. A *symbolic tuple* has form $\langle \tau_1, \ldots, \tau_n \rangle$, where $\tau_i \in Symb \cup Dom$ for each $i \in [1..n]$.

Let $R(A_1, \ldots, A_n; B_1, \ldots, B_m)$ be a relation schema in the world schema, with key $A_1, \ldots, A_m$. The notion of "symbolic instance" of R abstractly represent the set of tuples that have been "visited" in R. We must also keep track of tuples that are currently "not in" R, which corresponds to tuples that have been deleted from R by some atomic execution. Formally, a *symbolic instance* of R is a pair (In^R, Out^R), where In^R is a finite set of symbolic tuples over $A_1, \ldots, A_n, B_1, \ldots, B_m$, and Out^R is a set of symbolic tuples over $A_1, \ldots, A_n$. The instance (In^R, Out^R) is *well-formed* for svc $\widehat{\gamma}$ if (informally)

1. if $\widehat{\gamma} \models \neg\texttt{incorp}(y_i)$, then y_i should not appear in In^R nor Out^R;
2. $\pi_{A_1,\ldots,A_n}(In^R) \cap Out^R$ is empty;
3. In^R is closed under the tuple-generating dependencies having the form $(R(\tau_1, \ldots, \tau_n, \eta_1, \ldots, \eta_m) \wedge \tau_j = \tau'_j \rightarrow R(\tau_1, \ldots, \tau'_j, \ldots, \tau_n, \eta_1, \ldots, \eta_m))$ (Intuitively, we are "closing" the symbolic instance to include all tuples that are equivalent under equalities implied by $\widehat{\gamma}$);
4. In^R "satisfies" the key dependency $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ "modulo the equalities in $\widehat{\gamma}$.

In the following we consider only well-formed symbolic instances.

Let $\widehat{\gamma}$ be an svc over Y, C. A (*valid*) *symbolic instance* of world schema $\mathcal{W}$ is a mapping $\widehat{\mathcal{I}}$ that maps each relation $R \in \mathcal{W}$ into a well-formed symbolic instance of R over Y, C. (We also write, e.g., $\mathcal{I}(In^R)$ to refer to the In component of $\mathcal{I}(R)$.)

Given an execution tree $\mathcal{T}$ of a system $\mathcal{S}$ in $\texttt{Colombo}^{k,b}$ satisfying the restrictions mentioned in Section 4, we can inductively build up a symbolic execution tree $\widehat{\mathcal{T}}$ that correspond to $\mathcal{T}$ but using symbolic values, symbolic ids, and symbolic world states. We let Y be a set of symbolic values which is "large enough" to accomodate the (bounded) number of look-ups that might occur in an execution of $\mathcal{S}$, and let C be the set of all constant values occurring in the specification of $\mathcal{S}$. At the root and children of the root the associated svc $\widehat{\gamma}$ will satisfy $\neg\texttt{incorp}(y)$ for all symbolic values y. Intuitively, as we proceed down a path of $\widehat{\mathcal{T}}$, we will extend $\widehat{\gamma}$ to incorporate symbolically the concrete values that have been read from the world state by atomic process invocations. Along each path the value of $\widehat{\gamma}$ is refined by "incorporating" new symbolic values and assigning for them relationships to the other incorporated symbolic values and to C. This process is additive or monotonic, in the sense that once a symbolic value y is incorporated into $\widehat{\gamma}$ its relationships to the other previously incorporated symbolic values does not change. After an atomic process invocation we may also have to modify the symbolic instances (In^R, Out^R) for each R in the world schema.

A subtlety in extending the svc $\widehat{\gamma}$ is that we must avoid running out of symbolic values. Suppose that $\widehat{\mathcal{I}}$ is a symbolic instance and $\widehat{\gamma}$ an svc. Let $R(A_1, \ldots, A_n; B_1, \ldots, B_m)$ have key $A_1, \ldots, A_n$. We say that $(\widehat{\gamma}, \widehat{\mathcal{I}})$ *knows* $f_j^R(\tau_1, \ldots, \tau_n)$ (where the τ_i's range over $Y \cup C$) if $\langle \tau_1, \ldots, \tau_n \rangle \in \pi_{A_1,\ldots,A_n}(\widehat{\mathcal{I}}(In^R))$.

Based on the above definitions, it is now possible to define the *moves-to* relation between symbolic ids of a service S. We focus on atomic process invocations here. Speaking informally, suppose that there is a transition from state s via atomic process $a(u_1, \ldots, u_n; v_1, \ldots, v_m)$. We describe when $((s, \widehat{\alpha}, \widehat{\beta}, \widehat{\gamma}), \widehat{\mathcal{I}}) \widehat{\vdash} ((s', \widehat{\alpha'}, \widehat{\beta'}, \widehat{\gamma'}), \widehat{\mathcal{I}'})$ will hold. First note that there is non-determinism here, corresponding to the "new" values that are read by the conditions or updates performed by a. For each family of non-deterministic choices, new $\widehat{\gamma''}$ and $\widehat{\mathcal{I}''}$ is constructed, corresponding to "new" values seen and taking advantage of what $(\widehat{\gamma}, \widehat{\mathcal{I}})$ "knows". Then, for each conditional effect (c, E) whose condition is "true" for $(\widehat{\gamma''}, \widehat{\mathcal{I}''})$, a pair $(\widehat{\gamma'}, \widehat{\mathcal{I}'})$ is constructed, where $\widehat{\gamma'} = \widehat{\gamma''}$, and $\widehat{\mathcal{I}'}$ is constructed from $\widehat{\mathcal{I}''}$ according to the effect E. The relation $\widehat{\vdash}$ for systems $\mathcal{S}$ is defined analogously.

We summarize our overview of this reduction from infinite to finite with the following.

Lemma 5.1: (Informally stated) Let $\mathcal{S}$ be a system of services in $\texttt{Colombo}^{k,b}$, and $\mathcal{T}$ an execution tree for $\mathcal{S}$, and let symbolic execution tree $\widehat{\mathcal{T}}$ be constructed as described above. Then there is a homomorphism h from $\mathcal{T}$ to $\widehat{\mathcal{T}}$ with the following properties: (i) h "preserves levels" (i.e., the depth of node $h(n)$ in $\widehat{\mathcal{T}}$ is the same as the depth of n in $\mathcal{T}$. (ii) If n is labeled by $(\texttt{id}, \mathcal{I})$, then $h(n)$ is labeled by $(\widehat{\texttt{id}}, \widehat{\mathcal{I}})$ with svc $\widehat{\gamma}$, where $(\widehat{\gamma}, \widehat{\mathcal{I}})$ is "consistent" with $\mathcal{I}$ (and also with the world state accesses that have occurred in the history above n). (iii) If n' is a child of n in $\mathcal{T}$, then the $\widehat{\vdash}$ relation holds between the labels of $h(n)$ and $h(n')$ in $\widehat{\mathcal{T}}$.

Importantly, the symbolic execution tree $\widehat{\mathcal{T}}$ described in the preceding lemma has bounded branching.

6 Characterization of Composition Synthesis in PDL

To complete the proofs of Theorems 4.1 and 4.2 we show now 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 [4, 11].

The intuition behind the encoding of composition synthesis in PDL, is the following: The execution of the various services that participate to the composition is completely characterized, in the sense that a model of the formula corresponds to a single execution tree of the system, in which the mediator activates the component services by sending them suitable messages, and the component services execute the actions of the goal while exchanging messages with the mediator. In fact, a model of the formula simultaneously represents both the execution of the component services, and the execution of the goal specification.

The set of non-deterministic outcomes that can be obtained every time an atomic process is executed by a component service (and by the goal) corresponds to the set of children nodes in the model of the PDL formula.

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.

More precisely, 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. In particular we have to consider the following types of actions:

1. client sends message,
2. client reads message,
3. mediator/goal sends message to client,
4. mediator/goal reads message from client,
5. mediator sends message to component service,
6. mediator reads message from component service,
7. service sends message to mediator,
8. service reads message from mediator,
9. service/goal executes atomic process.

For lack of space, here we will only give some hints on how the PDL encoding is defined. More details can be found in [4]. In specifying the encoding, we make use of the following meta-variables representing suitable PDL sub-formulas: *(i)* $\hat{\hat{\alpha}}$ denotes the PDL representation of an assignment over the set of variables of both the local stores `LStore` and the queue stores `QStore` of all services, including the goal. We also use $\widehat{\widehat{\alpha_p}}$ to denote the part of $\hat{\hat{\alpha}}$ relative to Service p, for $p \in \{0, 1, \ldots, n, g\}$ (here g denotes the goal); *(ii)* $\hat{\hat{\gamma}}$ denotes the PDL representation of the sv-characterization $\hat{\gamma}$; *(iii)* $\hat{\hat{\mathcal{I}}}$ denotes the PDL representation of a world state instance.

We make use of one proposition st^i_j for each state j of the guarded automaton for service S_i (all these are pairwise disjoint), and of one proposition $exec_i$, for each service S_i (either the mediator, a component service, or the goal), intended to be true when service S_i is executing.

To determine the execution of the mediator, we will use the following "guessed" propositions: $DO(!m)$ (resp., $DO(?m)$), stating that next a send (resp., a read) by the mediator will be performed[2]; $NEXT(st^0_i)$, stating that the mediator will make a transition to state i; $MAP(q^{\vec{0}}_m, \vec{u})$, stating that the mediator reads a message m using variables $\vec{u}$ as output parameters for the message; $MAP(\vec{u}, q^{\vec{i}}_m)$, stating that the mediator sends a message m to service S_i using variables $\vec{u}$ as input parameters.

As an example of the kind of (sub) formulas we use, consider the characterization of executing an atomic process. Lets assume that the service S_i is executing mimicking the call of an atomic process in the goal S_g. In particular, let S_i be in the state st^i_h with a transition labeled by a guarded action $\phi/a(\vec{x^i}; \vec{y^i})$ getting to a state $st^i_{h'}$ and let the goal S_g be in st^g_k with a transition labeled by a guarded action $\phi'/a(\vec{x^g}; \vec{y^g})$ getting to a state $st^g_{k'}$; and let us assume that both ϕ and ϕ' evaluate to true wrt assignment $\hat{\hat{\alpha}}$ and svc $\hat{\hat{\gamma}}$. Then we have

$$\begin{aligned}
&[*]((exec_i \wedge exec_g \wedge st^i_h \wedge st^g_k \wedge \hat{\hat{\gamma}} \wedge \hat{\hat{\alpha}} \wedge \hat{\hat{\mathcal{I}}}) \rightarrow \\
&\qquad \langle a \rangle \top \wedge [-a] \bot \wedge \\
&\qquad [a](st^i_{h'} \wedge st^g_{k'}) \wedge \\
&\qquad \textstyle\bigwedge_{(\hat{\hat{\gamma}}', \hat{\hat{\alpha}}', \hat{\hat{\mathcal{I}}}') \in \mathcal{E}} \langle a \rangle (\hat{\hat{\gamma}}' \wedge \hat{\hat{\alpha}}' \wedge \hat{\hat{\mathcal{I}}}') \wedge \\
&\qquad [a](\textstyle\bigvee_{(\hat{\hat{\gamma}}', \hat{\hat{\alpha}}', \hat{\hat{\mathcal{I}}}') \in \mathcal{E}} \hat{\hat{\gamma}}' \wedge \hat{\hat{\alpha}}' \wedge \hat{\hat{\mathcal{I}}}') \\
&\qquad [a](exec_i \wedge exec_g))
\end{aligned}$$

where each $(\hat{\hat{\gamma}}', \hat{\hat{\alpha}}', \hat{\hat{\mathcal{I}}}') \in \mathcal{E}$ is the PDL representation of a triple $(\hat{\gamma}', \hat{\alpha}', \hat{\mathcal{I}}')$ such that for the action $a(\vec{x^i}; \vec{y^i})/a(\vec{x^g}; \vec{y^g})$ we have that $(\hat{\gamma}, \hat{\alpha}, \hat{\mathcal{I}}) \vdash (\hat{\gamma}', \hat{\alpha}', \hat{\mathcal{I}}')$, where $\hat{\hat{\alpha'_i}}$ and $\widehat{\widehat{\alpha'_g}}$ are the only parts of $\hat{\hat{\alpha}}'$ that may be different from $\hat{\hat{\alpha}}$.

This formula states that every time S_i and S_g are executing and they are in states st^i_h and $st^g_{h'}$ respectively, and $\hat{\hat{\alpha}}$ and $\hat{\hat{\gamma}}$ hold, then: *(i)* the atomic process a is activated next (and no other action are possible);*(ii)* executing a leads S_i and S_g to the states st^i_k and $st^g_{k'}$, respectively; *(iii)* there is an execution branch for each $(\hat{\hat{\gamma}}', \hat{\hat{\alpha}}', \hat{\hat{\mathcal{I}}}') \in \mathcal{E}$; *(iv)* the only possible next $(\hat{\hat{\gamma}}', \hat{\hat{\alpha}}', \hat{\hat{\mathcal{I}}}')$ must be in $\mathcal{E}$; *(v)* the service S_i and the goal S_g will continue executing next.

Other examples can be found in [4].

Finally, among the structural part of the formula, prominent parts are those of the form

$$\begin{aligned}
&\langle * \rangle (exec_0 \wedge st^0_i \wedge \widehat{\widehat{\alpha_0}} \wedge \hat{\hat{\gamma}} \wedge DO(!m)) \rightarrow \\
&\qquad [*](exec_0 \wedge st^0_i \wedge \hat{\hat{\alpha}} \wedge \hat{\hat{\gamma}} \rightarrow DO(!m))
\end{aligned}$$

[2]In fact, due to Strict Mediation, $DO(?m)$ is completely determined by the execution of a send by a component service.

which state that a guessed proposition, $DO(!m)$ in this case, must assume the same value everywhere the mediator is executing in a certain state st_i^0 with a certain assignment $\widehat{\widehat{\alpha_0}}$ for its LStore and QStore and with a certain sv-characterization $\widehat{\widehat{\gamma}}$.

Lemma 6.1: Assume that all services are in $\texttt{Colombo}^{\mathrm{k,b}}$, and assume No External Modifications. Let $\mathcal{G} = (C, \{G\}, L)$ be a goal system and $\mathcal{U}$ a finite family of available web services, all of which satisfy Blocking Behavior and Bounded Access. For each p, q, let $\Phi_{p,q}^{\mathcal{G},\mathcal{U}}$ be the PDL formula constructed as above. Then, if $\Phi_{p,q}^{\mathcal{G},\mathcal{U}}$ is satisfiable, there exists a system $\mathcal{S} = (C, \{S_0, S_1, \ldots, S_n\}, L')$, where S_0 is a (p, q)-bounded mediator, $S_1, \ldots, S_n \in \mathcal{U}$, and the linkage L' satisfies Strict Mediation, that is (symbolically) equivalent to G.

Indeed, by the tree-model property of PDL, if $\Phi_{p,q}^{\mathcal{G},\mathcal{U}}$ is satisfiable, then it admits a tree-like model. From such a model we can extract directly a symbolic execution tree for the goal and for $\mathcal{S}$. To determine which services actually take part in the composition, it is sufficient to consider those services S_i for which $exec_i$ is true at least once.

Observe that, from a model of $\Phi_{p,q}^{\mathcal{G},\mathcal{U}}$, one can directly obtain also a specification of S_0. This can be done by considering for each of the p states of S_0 and for each value of $\widehat{\widehat{\alpha_0}}$ and $\widehat{\widehat{\gamma}}$, which of the guessed propositions are true. (Notice that the part of the PDL formula related to such guesses ensures that the state together with $\widehat{\widehat{\alpha_0}}$ and $\widehat{\widehat{\gamma}}$ determines once and for all the value of the guessed propositions in the whole model.) From the guessed propositions one can define the transitions of the guarded automaton for S_0, extracting from $\widehat{\widehat{\alpha_0}}$ and $\widehat{\widehat{\gamma}}$ the guards, and from the DO and MAP propositions (see [4]) the actions and their parameters respectively. Considering that the local store and the queue store for a (p, q)-bounded mediator whose linkage satisfies Strict Mediation are pre-determined, this provides a complete characterization of the mediator.

7 Conclusion and Future Work

In this paper we have presented Colombo, a framework for automatic web service composition, that addresses *(i)* message exchanges, *(ii)* data flow management, and *(iii)* effects on the real world, thus unifying the main approaches that are currently undertaken by the research community for the service composition problem. Through a complex example we have shown all the peculiarities of the approach. We have presented a novel technique, based on case tree building and on an encoding in PDL, for computing the composition of web services.

In future work we will remove some of the assumptions that we considered in this work (characterizing $\texttt{Colombo}^{\mathrm{k,b}}$). We will consider complex types (i.e., arbitrary XML data types that can be transmitted between services), more general accesses to data stores and queues of arbitrary, but yet finite, length.

Acknowledgement

This work has been supported by MIUR through the "FIRB 2001" project *MAIS* - WP 2, "FIRB 2003" project *eG4M* and "Società dell'Informazione" sub-project SP1 *"Reti Internet: Efficienza, Integrazione e Sicurezza"*. 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).

The authors would like to thank Maurizio Lenzerini and the members of the SWSL working group, in particular Michael Gruninger, Sheila McIlraith and Jianwen Su, for valuable discussions.

References

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

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services (BPEL4WS). http://www-106.ibm.com/developerworks/library/ws-bpel/, 2004.

[3] Ariba, Microsoft, and IBM. Web Services Description Language (WSDL). http://www.w3.org/TR/2001/NOTE-wsdl-20010315, 2001.

[4] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. On-line Appendix to the Paper "Automatic Composition of Transition-based Semantic Web Services with Messaging. *Tech. Rep. 06/2005*. http://www.dis.uniroma1.it/~mecella/publications/eService/AppendixVLDB2005.pdf, 2005.

[5] 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*.

[6] 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*.

[7] D. Berardi, G. De Giacomo and M. Mecella. Basis for Automatic Service Composition. Tutorial at WWW 2005.

[8] A. Deutsch, L. Sui, and V. Vianu. Specification and Verification of Data-driven Web Services. In *Proc. of PODS 2004*.

[9] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW 2004*.

[10] Semantic Web Services Framework (version 1.1). http://www.daml.org/services/swsf/1.1/, 2005.

[11] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.

[12] P. Helland. Data on the outside versus data on the inside. In *CIDR*, pages 144–153, 2005.

[13] R. Hull and J. Su. Domain Independence and the Relational Calculus. *Acta Informatica*, 31(6):513–524, 1994.

[14] OWL-based Web Service Ontology. OWL-S 1.1, November 2004. http://www.daml.org/services/owl-s/1.1/.

[15] S. McIlraith, T. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46 – 53, 2001.

[16] S. McIlraith, T. Son. Adapting Golog for Composition of Semantic Web Services. In *Proc. of KR 2002*, 482 – 496, 2002.

[17] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[18] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN Planning for Web Service Composition using SHOP2. *J. Web Sem.*, 1(4):377–396, 2004.

[19] P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *Proc. of ISWC 2004*.