Parallel Web Service Composition in MoSCoE: A Choreography-based Approach

Jyotishman Pathak[1,2] **Samik Basu**[1] **Robyn Lutz**[1,3] **Vasant Honavar**[1,2]
[1]Department of Computer Science, Iowa State University, Ames IA 50011
[2]Center for Computational Intelligence, Learning & Discovery, Iowa State University, Ames IA 50011
[3]Jet Propulsion Lab/California Institute of Technology, Pasadena CA 91109
{jpathak, sbasu, rlutz, honavar}@cs.iastate.edu

Abstract

We present a goal-driven approach to model a choreographer for realizing composite Web services. In this framework, the users start with an abstract, and possibly incomplete functional specification of a desired goal service. This specification is used to compose a choreographer that allows communication between the client and the set of available component services, and is functionally equivalent to the goal service. However, if such a composition cannot be realized, the proposed approach identifies the cause(s) for the failure of composition. This information can be used by the user to minimally reformulate the goal to reduce the 'gap' between the desired functionality. The process can be iterated until a feasible composition is realized or the user decides to abort. The approach ensures that (i) a choreographer, if one is produced by our composition algorithm, in fact realizes the user-specified goal functionality; and (ii) the algorithm is guaranteed to find a composition that meets the user needs as captured in the goal specifications (whenever such a composition exists).

1 Introduction

Many real-world applications of Web services, e.g., e-Business, e-Science, call for effective approaches to automated or semi-automated assembly of composite Web services by integrating independently developed component services. Consequently, a variety of approaches based on planning techniques of artificial intelligence, logic programming, automata-theory have been developed (see [5,7] for a survey). However, these techniques suffer from a very significant limitation in that they require the user (or service developer) to provide a specification of the desired behavior of the composite service (goal) in its entirety using languages such as OWL-S [1, 15, 17] or BPEL [13]. This becomes a problem when modeling complex Web services because the complexity of the composition graph grows rapidly with the increasing complexity of the desired goal service. More importantly, the current approaches adopt a "single-step request-response" paradigm to service composition. That is, if a specified goal service is unrealizable (which would be the case if the goal service specification is incomplete), the process simply fails. It is typically difficult for a developer to provide the complete goal service specification that is needed in the absence of a detailed knowledge of the specifications of the component services available. This argues for an iterative approach to service composition wherein an abstract (and perhaps incomplete) goal service specification can be iteratively reformulated (with guidance from the system) until a composition that realizes the desired goal functionality is found, or the user decides to abort.

To address this need, we have introduced a framework for Modeling Service Composition and Execution (MoSCoE) [9, 10]. MoSCoE models services using Symbolic Transition System (STS) which are labeled transition systems augmented with guards on transitions and state variables over an infinite-domain. MoSCoE, given an *abstract* (high-level and possibly incomplete) STS specifications of a goal service T_g, and of available component services $T_1 \ldots T_n$, identifies a subset of the component services that when *composed* with a choreographer T_{cr} realize the goal service T_g. A unique feature of MoSCoE is its ability, in the event of failure to realize a goal service, to identify the specific states and transitions of the goal STS that need to be modified[1]. This information enables the user to *reformulate* the goal specification (iteratively) until a composition that realizes the goal specification is found or the user decides to abort.

In our previous work, we have described an algorithm for selecting and composing Web Services through iterative reformulation of functional specifications in the case of *sequential composition* [10]. In this paper, we turn our attention to modeling a choreographer for *parallel composition* of available component services to realize a goal service. In this setting, the component services interact via the choreographer. The role of the choreographer is to repli-

[1]This analysis is performed at the *design* time, and not at *run* time.

cate the input and output actions of the goal transition system and sending (receiving) messages to (from) the components. Thus, the component services provide the required functionality needed to realize the goal service.

The specific contributions of this paper include:

1. A sound and complete algorithm for selecting a subset of the available component services that can be assembled into a parallel composition that realizes the goal service with the user-specified functionality, and for determining a choreographer to interact with the component services. The proposed approach uses a variant of STS with guards on transitions to deal with the case when data and process flow are modeled in an infinite-domain.

2. An approach to determining the cause of failure of modeling a choreographer for parallel composition to assist the user in modifying and reformulating the goal specification in an iterative fashion.

The rest of the paper is organized as follows: Section 2 introduces an example used to illustrate the main ideas in this paper. Section 3 formulates the service composition problem in terms of STS as well as provides an algorithm for modeling a choreographer and identifying the cause for failure of composition. This section also includes correctness and complexity analysis of the proposed approach. Section 4 briefly discusses related work, and finally Section 5 concludes a summary and a brief outline of some directions for further research.

2 Illustrative Example

We present a simple example where a service developer is assigned to model a new Web service, `Health4U`, which allows senior citizens to make a doctor's appointment to receive medical attention for a particular ailment. To achieve this, `Health4U` relies on five existing (possibly independent) services: `Appointment`, `MedInsurance`, `MedRecord`, `e-Ride` and `Validate`. `Appointment` accepts patient data (*name*, *ailment* s/he is suffering from) and scheduling information (preferred *date* and *time*) as input to make an appointment. `Appointment` takes into account: (a) information about patient's insurance coverage plan to identify the designated physicians from whom the patient can receive treatment, and (b) the medical history (if any) that provides information about patient's previous appointments for the particular ailment. To obtain the needed information, `Appointment` communicates with `MedInsurance` (case (a)) and `MedRecord` (case (b)), both of which require the patient's *SSN* (Social Security Number). `Appointment` attempts to schedule an appointment for the patient with a physician who has treated the patient in the past. If no such physician is available, it makes an

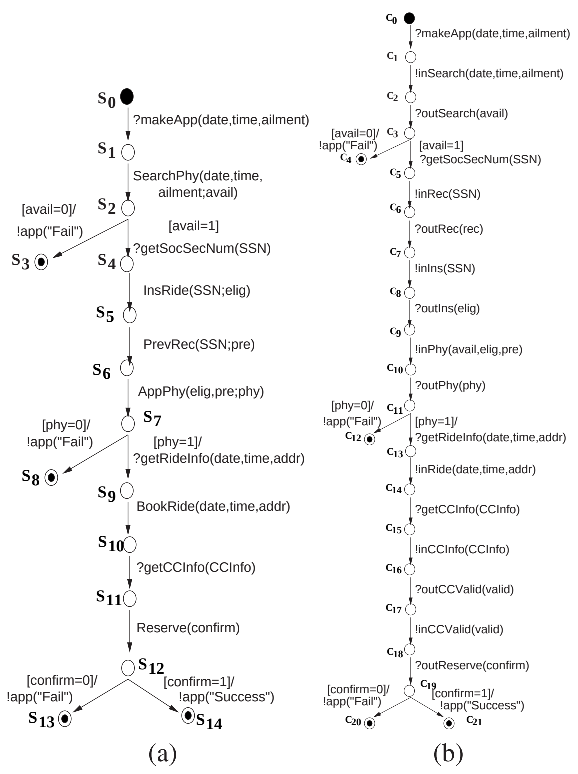


Figure 1: (a) STS representation of `Health4U` (b) The Choreographer

appointment with a physician who is among those designated by the insurance provider. Furthermore, `Health4U` arranges transportation for the patient to the medical center via the `e-Ride` service. This service needs the *date* and *time* for pick-up, as well as the patient's *address*. In addition, `e-Ride` communicates with `Validate` to determine whether the patient has provided a valid payment information (e.g., credit card) before completing the reservation.

We proceed now to outline how the composition of a service like `Health4U` can be accomplished by MoSCoE. MoSCoE receives from the service developer an STS specification (see Section 3.1) of the desired goal service `Health4U` as shown in Figure 1(a) . MoSCoE uses the goal service specification to construct a choreographer that enables the interaction between (a subset of) the component services to provide the desired goal service functionality. Figure 1(b) shows a choreographer that realizes `Health4U` using component services shown in Figure 2.

We use `?msgHeader(msgSet)` to refer to input actions and `!msgHeader(msgSet)` to refer to output actions of services. Communication between different services occurs via synchronization between actions with the same `msgHeader` resulting in the transfer of `msgSet` from the entity performing an output action to the one performing an input action. Example of such an action that results in a change of *state* from s_0 to s_1 is `?makeApp(date, time, ailment)` is shown in Figure 1(a). This is

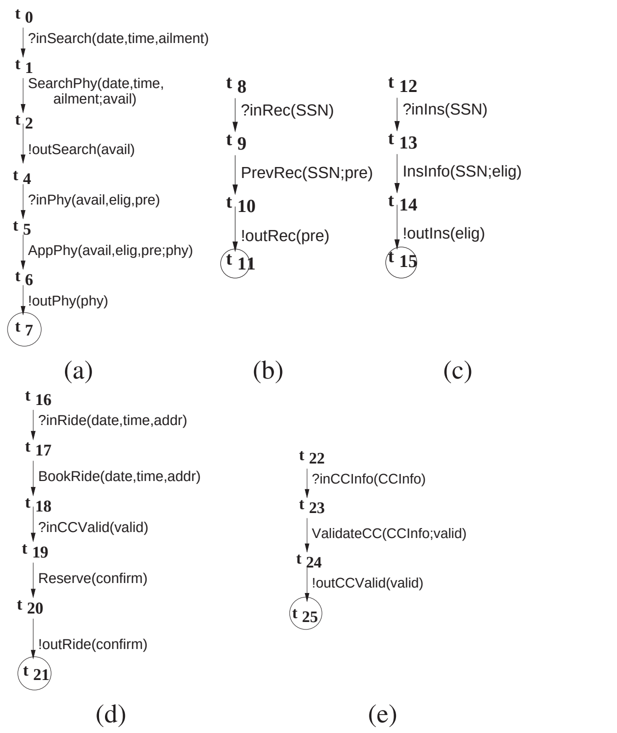


Figure 2: STS representation of (a) `Appointment` (b) `MedRecord` (c) `MedInsurance` (d) `e-Ride` (e) `Validate`

an input action where `makeApp` is the message header and input messages requested are `data`, `time` and `ailment`. The services also include atomic transition actions denoted by `funcName(inputSet;output)`. `SearchPhy(date,time,ailment; avail)` corresponding to the transition from state s_1 to s_2 in Figure 1(a) is an example of such an atomic transition action (the first three arguments are input arguments and the last argument is the return value of the function). A transition is annotated by guards which control whether or not the transition is enabled. Guards in MoSCoE (denoted by `[guards]`) correspond to constraints between variables, and are essentially pre-conditions for the atomic functions. Absence of a guard on a transition implies that the guard is *true* (always enabled).

In the above example, the choreographer replicates the input action `makeApp(date, time, ailment` (see transition from c_0 to c_1 in Figure 1(b)) as required by the goal service (Figure 1(a)) and sends message via `!inSearch(date,time,ailment)` (transition from c_1 to c_2) to the component service `Appointment` (Figure 2(a)). The service `Appointment` synchronizes with output action from the choreographer via the input action `?inSearch(date,time,ailment)` and the messages `date`, `time` and `ailment` are transfered from the choreographer to the service `Appointment`.

We describe the composition framework outlined above in more precise terms in the sections that follow.

3 Service Composition in MoSCoE

3.1 Symbolic Transition System

Preliminaries & Notations. We use the traditional definitions of variables, functions and predicates. Expressions are denoted by functions and variables. Guards, denoted by γ, are predicates over other predicates and expressions. Variables in a term t are represented by a set $vars(t)$. Substitutions, denoted by σ, map variables to expressions. A substitution of variable v to expression e is denoted by $[e/v]$. A term t under the substitution σ is denoted by $t\sigma$. An action is a term that takes one of the following forms:

1. `?msgHeader(msgSet)`: input action. Variables of the input action are in `msgSet`, i.e. $vars($`?msgHeader(msgSet)`$)$ = `msgSet`.
2. `!msgHeader(msgSet)`: output action. Variables of the output action are also in `msgSet`, $vars($`!msgHeader(msgSet)`$)$ = `msgSet`.
3. τ: an internal or unobservable action of a composition. Two entities synchronize on input and output action with the same message header to generate such an action.
4. `funcName(I; O)`: function invocation with input parameters `I` and return valuation `O`. We say that $ivars($`funcName(I;O)`$)$ = `I`, $ovars($`funcName(I;O)`$)$ = $\{$`O`$\}$ and $vars($`funcName(I;O)`$)$ = `I` $\cup\{$`O`$\}$.

Definition 1 (Symbolic Transition System) *A symbolic transition system is a tuple* $(S, \longrightarrow, s0, S^F)$ *where* S *is a set of states represented by terms,* $s0 \in S$ *is the start state,* $S^F \subseteq S$ *is the set of final states and* $\longrightarrow$ *is the set of transition relations of the form* $s \xrightarrow{\gamma,\alpha} t$ *where:*

1. *an action* α *such that*
   (a) $vars(\alpha) \subseteq vars(s)$ *if* α *is an output action*
   (b) $vars(\alpha) \cap vars(s) = \emptyset$ *if* α *is an input action*
   (c) $ivars(\alpha) \subseteq vars(s) \wedge ovars(\alpha) \cap vars(s) = \emptyset$ *if* α *is a function invocation*
2. *a guard* γ *such that* $vars(\gamma) \subseteq vars(s)$*, and*
3. $vars(t) \subseteq vars(s) \cup vars(\alpha)$.

For example, Figure 1(a) shows an STS representation of the `Health4U` service described in Section 2. Here, a transition from state s_2 to s_4 is annotated with an input transition function `?gotSocSecNum(SSN)` (which corresponds to item 1(b) in Definition 1) and a guard `[avail=1]` (which corresponds to item 2 in Definition 1). It can be also observed that $vars(s_4) \subseteq vars(s_2) \cup vars($`?gotSocSecNum(SSN)`$)$.

Semantics of STS. The semantics of an STS is given with respect to substitutions of variables present in the system. A state represented by the term s is interpreted under substitution σ ($s\sigma$). A transition $s \xrightarrow{\gamma,\alpha} t$, under *late semantics*, is said to be *enabled* from $s\sigma$ if $\gamma\sigma = \texttt{true}$. The transition under substitution σ is denoted by $s\sigma \xrightarrow{\alpha\sigma} t\sigma$.

Such *late semantics* form a natural interpretation of STSs by capturing the substitutions of input-variables at the destination state of a transition. For instance, consider an input transition of the form $s \xrightarrow{?\mathtt{m}(\vec{x})} t$. From the definition of STS, $\vec{x} \cap vars(s) = \emptyset$. A consequence of late semantics is that if t contains elements in $\vec{x}$, their valuations are left to be interpreted by guards in subsequent transitions.

Equivalence between STSs. To identify equivalent $\mathtt{STS}$s in the presence of guarded transitions with input/output actions, function invocations and unobservable actions τ, we will use *weak* and *late* bisimulation equivalence relation. Given an $\mathtt{STS} = (S, \longrightarrow, s0, S^F)$, the weak, late bisimulation relation with respect to substitution θ, denoted by $\approx^\theta_w$, is a subset of $S \times S$ such that

$$\begin{array}{l} s_1 \approx^\theta_w s_2 \Rightarrow \\ \left\{ \begin{array}{l} (\forall s_1\theta \xrightarrow{\alpha_1\theta} t_1\theta : \exists s_2\theta \xrightarrow{\alpha_2\theta}_w t_2\theta : \forall\sigma : (\alpha_1\theta\sigma = \alpha_2\theta\sigma) \\ \hfill \wedge t_1 \approx^{\theta\sigma}_w t_2) \end{array} \right\} \\ \wedge\ s_2 \approx^\theta_w s_1 \end{array} \tag{1}$$

In the above, $s_2\theta \xrightarrow{\alpha_2\theta}_w t_2\theta$ denotes transitive closure of transitions over τ transitions, i.e., a transition may contain zero or more τ transitions preceding and following action α_2. Furthermore, α can be an ϵ or empty transition. Two states are said to be equivalent with respect to weak, late bisimulation, under the substitution θ, if they are related by the *largest* bisimilarity relation $\approx^\theta_w$. Two STSs are said to be bisimulation equivalent if and only if their start states are bisimilar.

For example, consider checking the bisimilarity of states p_1 and q_1 in the the STSs given in Figure 3. The state $p_{11}(x)$ is bisimilar to $q_{11}(x)$ when $x = 0$, and is bisimilar to $q_{12}(x)$ when $x \neq 0$. Similarly, $p_{12}(x)$ is bisimilar to $q_{11}(x)$ when $x \neq 0$, and is bisimilar to $q_{12}(x)$ when $x = 0$. However, p_1 and q_1 are not bisimilar as the input action $?c(x)$ from p_1 to $p_{11}(x)$, if matched with input action $?c(x)$ from q_1 to $q_{11}(x)$, demands that $p_{11}(x)$ and $q_{11}(x)$ are bisimilar for all possible valuations of x (i.e., for both $x = 0$ and $x \neq 0$).

Definition 2 (Parallel Composition of STSs) *Given two symbolic transition systems* $STS_1 = (S_1, \longrightarrow_1, s0_1, S^F_1)$ $STS_2 = (S_2, \longrightarrow_2, s0_2, S^F_2)$*, their parallel composition, under the restriction set L, is denoted by* $(STS_1 \parallel STS_2)\backslash L = (S_{12}, \longrightarrow_{12}, s0_{12}, S^F_{12})$ *where* $S_{12} \subseteq S_1 \times S_2$, $s0_{12} = (s0_1, s0_2)$, $S^F_{12} = \{(s_1, s_2) \mid s_1 \in S^F_1 \wedge s_2 \in S^F_2\}$ *and* $\longrightarrow_{12}$ *relation is of the form:*

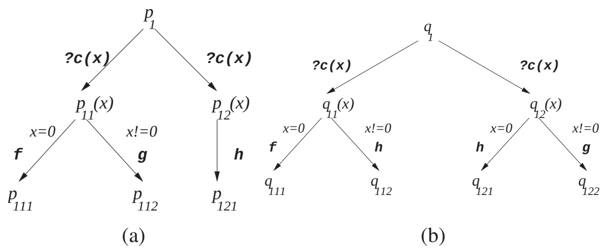


Figure 3: Example Symbolic Transition Systems. (a) $\mathtt{STS}_1$ (b) $\mathtt{STS}_2$

1. $s \xrightarrow{g_1, ?m(\vec{x})} s' \wedge t \xrightarrow{g_2, !m(\vec{x})} t' \wedge m \in L \Rightarrow (s,t) \xrightarrow{g_1 \wedge g_2, \tau} (s', t')$,

2. $s \xrightarrow{g_1,\alpha} s' \wedge header(\alpha) \notin L \Rightarrow (s,t) \xrightarrow{g_1,\alpha} (s',t)$ *and*

3. $t \xrightarrow{g_2,\alpha} t' \wedge header(\alpha) \notin L \Rightarrow (s,t) \xrightarrow{g_2,\alpha} (s,t')$.

In the above, restriction set L includes the message headers on which the participating $\mathtt{STS}$s must synchronize and generate a τ action. We use $header(\alpha)$ to return the message header of input and output actions; for function invocation and τ it returns a constant which is never present in L.

3.2 The Service Composition Problem

Given a goal service T_g and a set of available component services $T_1, T_2, \ldots, T_n$, solving the service composition problem entails identifying a composition of the necessary component services that realizes the functionality of T_g. In the setting of *parallel composition* that is the focus of this paper, this entails generating a choreographer T_{cr} which realizes the functionality of T_g by orchestrating the necessary interactions among the selected component services. As noted earlier, the choreographer T_{cr} replicates the behavior of the input/output actions of the goal and is responsible for communications between component services; it relies on the component services for function invocations needed to realize the goal service. In MoSCoE, the operation of the goal service as well as the component services are represented by the corresponding STSs.

Based on the definition of parallel composition and equivalence relation described in Section 3.1, and the previously introduced notion of a choreographer, the service composition problem can be described as:

$$\exists T_{cr} : (\ldots((T_{cr}||T_i)||T_j)|| \ldots ||T_k)\backslash L \approx^{\mathtt{true}}_w T_g$$

where, L contains all the input and output message headers of the component services. Thus, solving a parallel service composition problem entails to constructing a choreographer which can enable interaction between the component services so as to yield a behavior that is *equivalent* (weak, late bisimilar) to that of the desired goal service.

```
/*
  $r$  is the goal state; $s_i$ is the component state; $t$ is the generated choreographer state.
  $G$  is the conjunction of guard conditions that will be accumulated along each DFS path. All variables in $G$ are universally quantified.
  $R$  is a store that contains all the input & output message headers of the component services.
*/
1: proc generate($r$, $[s_1, s_2, \ldots, s_n]$, $t$, $G$, $R$)
2: {
3:   if visited($r$, $[s_1, s_2, \ldots, s_n]$, $t$, $G$, $R$); // This path has already been traversed
4:   else  mark as visited($r$, $[s_1, s_2, \ldots, s_n]$, $t$, $G$, $R$);

5:  forall ($r \xrightarrow{g,\,a} r'$) && ($G \wedge g$) do
6:     case 1:  /* input action from the client */
7:           $a = ?\mathrm{m}(\vec{x}) \Rightarrow$ create a transition $t \xrightarrow{g,a} t'$; $R := R \cup \vec{x}$; call generate($r'$, $[s_1, s_2, \ldots, s_n]$, $t'$, $G \wedge g$, $R \cup \vec{x}$);

8:     case 2:  /* output action to the client */
9:           $a = !\mathrm{m}(\vec{x}) \Rightarrow$  if ($\vec{x} \in R$) { create a transition $t \xrightarrow{g,a} t'$; call generate($r'$, $[s_1, s_2, \ldots, s_n]$, $t'$, $G \wedge g$, $R$); }
10:                      else $\boxed{\text{Requested output cannot be created for client. Return partial choreographer}}$

11:     case 3:  /* function-invocation to be provided by the components */
12:           $a =$ funcName(I; O) && no $s_i$ has a transition on the action a $\Rightarrow$
13:             select the component $T_i$ that is capable of generating the function;
14:           if ($s_i \xrightarrow{g_i,?\mathtt{m}(\vec{x})} s'_i$) && ($\vec{x} \notin R$) {
15:             if ($\mathtt{m} \in FL_{ij}$) { msgH:=m; $k := j$;} else $\boxed{\text{Return partial choreographer. Failure at action a.}}$
16:             while (($s_k \xrightarrow{g_k,a_k} s'_k$) && $header(a_k) \neq$ msgH) {
17:               if ($a_k = ?\mathtt{m}_k(\vec{y})$) && ($\vec{y} \notin R$) {
18:                 if ($\mathtt{m}_k \in FL_{kl}$) { msgH := $\mathtt{m}_k$; $k := l$;}
19:               } end of if-17
20:               elseif (($a_k = ?\mathtt{m}_k(\vec{y})$) && ($\vec{y} \in R$)) || ($a_k = !\mathtt{m}_k(\vec{y})$) {
21:                 if ($G \Rightarrow g_k$) {
22:                   create transition $t \xrightarrow{G,\overline{a_k}} t'$ to communicate with $s_k$;
23:                   call generate($r$, $[s_1, s_2, \ldots, s'_k, \ldots, s_n]$, $t'$, $G$, $R \cup \vec{y}$); if ($t'$ is the root of a partial choreographer), select next transition
                    from $s_k$; else break;
24:                 }
25:               } // end of elseif-20
26:               else { $\boxed{\text{Return partial choreographer. Failure at action a.}}$; break; }
27:             } // end of while-16
28:             if ($s_k \xrightarrow{g_k,a_k} s'_k$) && ($header(a_k) =$ msgH) {
29:               if ($G \Rightarrow g_k$) {
30:                 create transition $t \xrightarrow{G,\overline{a_k}} t'$ to communicate with $s_k$; call generate($r$, $[s_1, s_2, \ldots, s'_k, \ldots, s_n]$, $t'$, $G$, $R \cup vars(a_k)$);
31:               }
32:               else { $\boxed{\text{Return partial choreographer. Failure at action a.}}$;}
33:             }
34:             elseif ($s_k \notin S^F_k$) || (funcName(I;O) $\notin done$) $\boxed{\text{Return partial choreographer. Failure at action a.}}$;
35:             else return;
36:           } // end of if-14
37:           elseif ($s_i \xrightarrow{g_i,?\mathtt{m}(\vec{x})} s'_i$) && ($\vec{x} \in R$) && ($G \Rightarrow g_i$) {
38:             create transition $t \xrightarrow{G,!\mathtt{m}(\vec{x})} t'$ to communicate with $s_i$; call generate($r$, $[s_1, s_2, \ldots, s'_i, \ldots, s_n]$, $t'$, $G$, $R$);
39:             }
40:           }
41:           elseif ($s_i \xrightarrow{g_i,!\mathtt{m}(\vec{x})} s'_i$) && ($G \Rightarrow g_i$) {
42:             create transition $t \xrightarrow{G,?\mathtt{m}(\vec{x})} t'$ to communicate with $s_i$; call generate($r$, $[s_1, s_2, \ldots, s'_i, \ldots, s_n]$, $t'$, $G$, $R \cup \vec{x}$);
43:             }
44:           }
45:           else { $\boxed{\text{Return partial choreographer. Failure at action a.}}$; }

46:     case 4: $a =$ funcName(I; O) && $s_i$ has a transition on action a $\Rightarrow$
47:           if ($s_i \xrightarrow{g_i,\,a} s'_i$) && ($G \wedge g \Rightarrow g_i$)
48:             $done = done \cup$ funcName(I;O); call generate($r'$, $[s_1, s_2, \ldots, s'_i, \ldots, s_n]$, $t$, $G \wedge g$, $R \cup ovars(a)$);
49:           else $\boxed{\text{Return thepartial choreographer with failure at guarded action (g,a).}}$
50: }
```

Figure 4: Algorithm for Modeling the Choreographer & Failure-Cause Detection

3.3 Synthesis of a Choreographer

We now proceed to describe an algorithm for constructing a choreographer for a desired service from a set of component services. Since the goal service specification includes the descriptions of the desired *functions*, we select the subset of component services whose `STS`s provide the necessary *function* invocations to yield a set of candidate component services which the choreographer can work with.

Because the task of a choreographer is to orchestrate the interactions among component services, the algorithm for constructing the choreographer requires information regarding dependencies between components, i.e., the dependency of an input message of a component on the output of another. For example, a component T_i requires an input of the form $\texttt{?m}(\vec{x})$ and a component T_j provides an output of the form $\texttt{!m}(\vec{x})$, we say that T_i is dependent on T_j via the message header `m`. In such a setting, the choreographer needs to synchronize with the output message from T_j and pass on the output of T_j as an input message to T_i. To make this notion of dependency more precise, we define *flow links* which capture the dependencies between multiple component services.

Definition 3 (Flow Links) *For services T_i and T_j, if $\texttt{?m}(\vec{x})$ and $\texttt{!m}(\vec{x})$ are present in the specifications of the respective components $\texttt{STS}_i$ and $\texttt{STS}_j$, then `m` is said to be a member of the flow link (from j to i component) set denoted by FL_{ij}.*

For example, consider the component services `e-Ride` (Figure 2(d)) and `Validate` (Figure 2(e)). In order for `e-Ride` to reserve a ride, it needs valid payment information. This information is provided by `Validate` after it validates the credit card information provided by the patient. Hence, there *must be* a flow link from `Validate` to `e-Ride`.

The algorithm for modeling a choreographer (Figure 4) that is "equivalent" to the goal service works as follows: the procedure $\texttt{generate}(r, [s_1, s_2, \ldots, s_n], t, G, R)$ is invoked by providing the start states of the goal STS (r), the component STSs in $\mathcal{S}$ ($s_1, s_2, \ldots, s_n$), and the choreographer STS (t) that is being modeled. The initial guard condition G is set to `true` and R corresponds to a store that contains all the input and output message headers of the component services, which is initially empty. A global set *done* is used to keep track of whether a particular function invocation requested by the goal service is realized in the composition. There are four cases to consider:

Case 1: If the transition from the current state r in the goal STS to state r' has an input action, i.e., receiving a message from the client, then a corresponding transition with the input action is created in the choreographer (`line 7`) and R is updated with the `msgSet` of the input action.

Case 2: If the transition from the current state r in the goal STS to state r' has an output action, i.e., transmitting a message to the client, then a corresponding transition with the output action is created in the choreographer if the `msgSet` of the action is already present in R (`line 9`). Note that here the `msgSet` required to produce the output message can be only retrieved from R (assuming it was placed there as a result of preceding interactions between the component services).

Case 3: This case corresponds to a situation in which the transition action in the goal is a function invocation a and none of the component services can provide a transition on that action from their current states s_i. In such a scenario, the algorithm first selects a component service T_i which can provide the required function a (`line 13`)[2]. Now there are three scenarios: s_i has an input action for which the choreographer cannot provide input messages (`line 14`); s_i has an input action for which the choreographer can provide input messages (`line 37`); and s_i has an output action (`line 41`).

The last two of the preceding three scenarios are easily dealt with: the choreographer transitions are generated to provide appropriate output or input message as the case may be and the procedure `generate` is invoked recursively. Thus, in the last case, i.e., `line 41`, the store R is updated to include the output messages from the state s_i. The first scenario (`line 14`) is more involved. As the `msgSet` required at the input action from state s_i is not present in R (`line 14`), the flow links (Definition 3) are explored to determine a component T_j which can provide the message as output. However, it is possible that T_j, in turn, is at a state s_j which needs a different input or output message. If the message is on input action provided by the choreographer or if the message in on output action, then appropriate choreographer transition is created and `generate` is invoked recursively (`lines 20--24`). At `line 22`, $\overline{a_k}$ denotes the complement of a_k, i.e. $\overline{a_k} := !\mathrm{m}_k(\vec{y})$ if $a_k = ?\mathrm{m}_k(\vec{y})$; otherwise $\overline{a_k} := ?\mathrm{m}_k(\vec{y})$. In this case, after the recursive call to `generate`, a new transition from s_k is selected at the while-condition (`line 16`). If the input message at s_j cannot be provided by the choreographer another component via flow link is selected and the process is iterated (`lines 17--19`).

Outside the `while` loop, if there exists a component which has the output action at its current state (s_k in Figure at `line 28`) required by the input action at state s_i of T_i responsible for providing the function invocation (`lines 13,14,15`), then the choreographer transition communicating with this component (`line 30`) is generated. Finally, at `line 34, 35`, if the state s_k is not a final state or the global

[2] In practice, there might be more than one component service that can provide the required atomic action a, in which case, each choice is explored to find a feasible choreographer.

store $done$ does not include `funcName(I;O)`, i.e., there exists a transition with function-invocation from s_k (fall-through case from `lines 16,28`) or `funcName(I;O)` requirement is not provided along any of the paths by recursion, then failure is reported; otherwise the procedure returns with no error.

Case 4: Finally, this case considers a situation when the transition action in the goal is a function invocation a and there exists a component T_i which has a transition from its current state s_i on action a (`line 46--50`). The message store R is updated with the return values of the function and global store $done$ is updated to reflect that `funcName(I;O)` invocation requirement is realized.

We use a constraint solver to check the (un)satisfiability of guards on STS transitions. All the variables in the guard are universally quantified. At present, MoSCoE works with only equality and disequality constraints on infinite domain variables for which satisfiability checking of guards is decidable [2] [3]. The preceding algorithm may fail to construct a choreographer because of either due to the absence of an action that is necessary to achieve the goal service functionality or the unsatisfiability of guards. Analysis of the cause of such failure is discussed Section 3.4.

3.3.1 Modeling a Choreographer for `Health4U`

In what follows, we show how to model a choreographer for the `Health4U` composite service introduced in Section 2 using the formal framework and algorithm described above. Figure 1(a) shows an STS representation of the `Health4U` goal service and Figure 2 shows the corresponding STSs of a set of available services (as noted earlier, we assume that the STS specifications of component services are supplied to MoSCoE by the respective service providers). Given the goal service specification and a set of available component services, MoSCoE's task is to construct a choreographer (Figure 1(b)), which enables the interaction between the client and component services, and is "bisimulation equivalent" to the goal service.

MoSCoE begins with the start state s_0 of the goal STS and considers its transition to state s_1. Here, the transition takes place due to an input action `?makeApp(...)` from the client (Case 1), so MoSCoE creates an appropriate transition ($c_0 \longrightarrow c_1$) in the choreographer to receive the input message. For the transition $s_1 \longrightarrow s_2$ in the goal STS, the associated action is a function invocation (`SearchPhy(...)`). However, since none of the current component states ($t_0, t_8, t_{12}, t_{16}, t_{22}$) can make a transition on this action (Case 3), MoSCoE first selects the component `Appointment` because it can provide the requested function, and then creates an appropriate transition in the choreographer to send a message to `Appointment`. Once `Appointment` executes the function `SearchPhy(...)`, it transmits an output message (in this case, indicating the availability of physician(s) for treatment of the ailment on the requested date and time). This behavior is modeled by the choreographer in the transition $c_2 \longrightarrow c_3$ (Case 1) which receives the message from `Appointment`. Depending on whether a physician is available or not, MoSCoE creates transitions $c_3 \longrightarrow c_4$ and $c_3 \longrightarrow c_5$ to send/receive output/input message to/from the client (Cases 2 & 1), respectively. MoSCoE proceeds in a similar fashion to model transitions for function invocations `InsInfo(...)` and `PrevRec(...)`, and reach goal state s_6 and choreographer state c_9. Now, to model a corresponding transition for function invocation `AppPhy(...)`, the choreographer refers to the message store R for previous message exchanges between the client and component services, and generates an output message `!inPhy(avail,elig,pre)`. Note that the values for the variables (`avail`, `elig`, & `pre`) in the message were placed in R as a result of previous message exchanges between the choreographer and component services. Since R contains every message that the choreographer receives from the client and the component services, to select the relevant components (and their messages), the choreographer exploits the flow links (Definition 3) between the components, as illustrated in Case 3 of the algorithm. This process for constructing the choreographer terminate with success when for each transition leading to a final state in the goal, a corresponding transition in the choreographer is established.

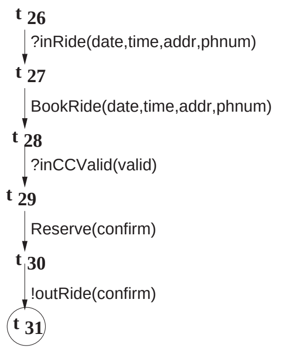
Now we proceed to discuss the scenario in which the algorithm for constructing a choreographer fails.

3.4 Analyzing the Failure of Composition & Reformulation of the Goal Specification

The algorithm described in Figure 4 for constructing a choreographer that realizes a specified goal service using the available component services fails when some aspect of the goal specification cannot be realized using the available component services. In the event of such failure, MoSCoE seeks to provide to the user information about the cause of the failure in a form that can be used to reformulate the goal specification. Recall that choreographer construction fails when there exists no choreographer that can enable the interaction among the available components to realize a behavior that is "bisimulation equivalent" to that of the goal service. In particular, bisimulation equivalence is not satisfied when:

1. The choreographer composed with components fails to create weak transition relation (see weak bisimilarity relation in Section 3.1). Weak transitions are generated by transitive closure of τ-transitions obtained via synchronization between choreographer and components.

[3]Investigation of larger classes of infinite state systems for which the construction of choreographer can be made decidable [8] is a topic of ongoing research.



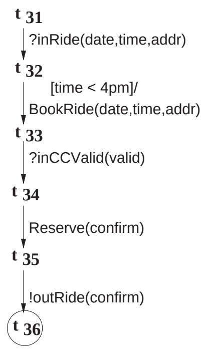


Figure 5: STS representation of (a) `e-Ride`' (b) `e-Ride`"

2. The actions between the goal and component transitions do not match.

3. The guard conditions are unsatisfiable.

Returning to the choreographer construction algorithm (Figure 4), we note that failures might be encountered during different stages of execution of the algorithm. For instance, `line 10` might result in a failure cause corresponding to Case 1 because the messages required for generating the output message to the client are not present in R. Similarly, in `lines 15` and `26` the failures might arise because either the input message required by a component services cannot be provided by some other component service or by the client itself. In `line 32, 49`, failure might occur because the guard conditions do not hold (the guards on the component transition are stronger than those on the goal). Finally, a failure could occur when there is a mismatch between an action that is required by the goal and actions that are provided by the available components (see `lines 34, 45`).

3.4.1 Failure Cause Analysis for `Health4U`

In our example from Section 2, suppose we replace the `e-Ride` component service (Figure 2(d)) with component services `e-Ride`' and `e-Ride`" yielding two separate instances of the `Health4U` composition problem (Figure 5)(a) & 5(b)). Suppose the behavior of `e-Ride`' is exactly the same as that of `e-Ride`, but it additionally requires a phone number to reserve a ride. Suppose on the other hand that `e-Ride`" can only reserve a ride if the time for pickup is before 4pm. Note that in both these instances, the algorithm for constructing the choreographer fails when it encounters the transition $s_9 \longrightarrow s_{10}$ in the goal STS (see Figure 1(a)). Specifically, in the case of the component service `e-Ride`', the actions for `Health4U` and `e-Ride`' do not match, whereas in the case of `e-Ride`", the corresponding guard condition is not satisfied. Thus, in the case of `e-Ride`' a failure results from an exception being raised either at `line 34` or `45`, indicating that a particular action present in the goal STS does not match with the component action for the particular transition. In the case of `e-Ride`" a failure arises due to an exception being raised either at `line 32` or `49`, indicating a mismatch in guards for the corresponding transition relation in the goal STS. MoSCoE provides such information about the cause of a failed attempt at service composition to the service developer. The developer can then reformulate the original goal specification (e.g., changing the function parameters or pre-conditions) to realize a suitable choreographer. These steps can be iterated until such a choreographer is eventually realized or the user decides to abort.

3.5 Analysis of the Composition Algorithm

Theorem 1 (Soundness & Completeness) *Given a goal service T_g with start state $s0_g$ and n component services $T_1 \dots T_n$ with the corresponding start states $s0_1 \dots s0_n$ the procedure* `generate`*$(s0_g, [s0_1, s0_2, \dots, s0_n], t0, true, \emptyset)$ (Figure 4) is guaranteed to terminate with a choreographer T_{cr} with start state $t0$ if and only if $(\dots((T_{cr}||T_1)||T_2)|| \dots ||T_n)\backslash L \approx_w^{true} T_g$ whenever such a choreographer exists, and with failure otherwise.*

Proof Sketch:. We prove the theorem by contradiction. Suppose the procedure `generate`$(s0_g, [s0_1, s0_2, \dots, s0_n], t0, true, \emptyset)$ (Figure 4) yields a choreographer T_{cr} with start state $t0$ which when used to orchestrate the component services under the restrictions imposed by the guards L, fails to realize the goal service T_g, i.e., the composition is not bisimulation equivalent to T_g. There are four cases to consider: (i) for an input action in T_g, there is no corresponding input action in T_{cr}; (ii) for an output action in T_g, there is no corresponding output action in T_{cr}; (iii) a function-invocation present in T_g is not modeled by the composition; and finally (iv) some sequence of actions in the goal is not provided by the composition due to the unsatisfiability of one or more guards.

However, case (i) is ruled out by the algorithm because for each message sent from the client to T_g, a corresponding input action is created in T_{cr} to receive the message (Case 1 of `generate`). Case (ii) is ruled out because for each output message that is to be sent to the client (as modeled in T_g), a corresponding output action is created in T_{cr} if that message can be retrieved from the message store R (otherwise an exception is raised resulting in termination of the algorithm with failure (Case 2 of `generate`)). Case (iii) is ruled out because the function invocations in T_g are modeled by first determining the component(s) that can provide the relevant functions and then creating the relevant transitions in T_{cr} to communicate with the respective component(s) (otherwise the algorithm terminates with failure). Note that the communications between T_{cr} and any T_i leads to transitions labeled by τ (Definition 2). The desired goal-function will be matched by the composition after zero steps if there is a component at a state with outgoing transition labeled by the function; otherwise the composition will lead to a state with an outgoing transition labeled by the desired function, after multiple τ-steps representing component-choreographer synchronous communications. Finally, in all the above cases, if the guards do not match or the guards in the component(s) are stronger than those in T_g (and T_{cr}), the algorithm terminates with an appropriate failure cause, thereby ruling out case (iv).

Next, consider the case where there exists a choreographer T_{cr} that can orchestrate the component services $T_1 \ldots T_n$ under the constraints imposed by L to realize the behavior specified by T_g but the procedure `generate` terminates with a partial T_{cr} or fails to terminate. We can rule out this possibility of generation of partial T_{cr} through an argument similar to the one used above. Finally, the component services T_is and the goal service T_g are defined over guarded transitions with no variable operations. As such the variable domain can be finitely partitioned making the state-space of the component and the goal services finite. Therefore, the procedure `generate`, which exhaustively explores the state-space of the services, terminates for all possible valuations of the variables.

Complexity. The worst-case complexity of the composition algorithm is determined by the number of recursive invocations of `generate`. Assume that $|T_g|$ is the number of states in the goal service STS, $|T_c|$ is the number of states in each component service STS, and n is the total number of component services. In the worst case, each state in the goal STS can be associated with any potential combination of states in the component STSs, yielding $|T_c|^n$ combinations. Additionally, each pairing of a goal state with a combination of component states is interpreted in the context of a guard G and the messages stored in R. Guards and message stores are updated whenever the procedure `generate` explores a transition from a goal or a component state. The number of distinct Gs and Rs is $O(2^{|T_g| \times |T_c|^n})$. The worst-case complexity of `generate` is therefore O($|T_g| \times |T_c|^n \times 2^{|T_g| \times |T_c|^n}$)[4].

4 Related Work

A variety of approaches to automated service composition, based on the planning techniques of artificial intelligence, logic programming, and automata-theory have been developed (see [5, 7] for a survey).

Of particular interest in the context of the work described in this paper are approaches to service composition within a transition system based framework. Fu et al. [6] model Web services as automata extended with a queue, and communicate by exchanging sequence of asynchronous messages, which are used to synthesize a composition for a given specification. Their approach is extended in Colombo [4] which models services as labeled transition systems with composition semantics defined via message passing, where the problem of determining a feasible composition is reduced to satisfiability of a deterministic propositional dynamic logic formula. Our framework has been inspired by, and builds on insights from Colombo.

Pistore et al. [12, 13, 17] represent Web services using non-deterministic state transition systems, which communicate through message passing. This approach constructs a parallel composition of *all* the available component services and then generates a plan that controls the services, based on the functional requirements specified using a temporal logic-based language. In contrast, services (goal and components) in our framework are represented using Symbolic Transition Systems augmented with guards over variables with infinite domains. We avoid the expensive step of generating a parallel composition of *all* the available services before developing a controller, by constructing a choreographer on-the-fly, i.e., transitions in the components and goals are explored by our algorithm as and when needed.

Model-driven techniques for service composition where standard UML tools are used to provide a higher level of abstraction of the desired composite service have received some attention in the literature. For example, SELF-SERV [3] uses state-charts for modeling composite services. Skogan et al. [14] use UML for capturing composite Web service patterns. Timm and Gannod [16] rely on translation into OWL-S of semantic web services specified in UML.

The proposed framework, MoSCoE, works with abstract specification of a goal service either directly using STS or indirectly using UML state machines [10] which can be translated into STS. As noted earlier, MoSCoE is inspired by Colombo, and several of the other approaches to service composition cited above. MoSCoE is unique in its ability to identify of reasons for failure of an attempt to compose a goal service using available components, which together with its ability to work with abstract (and possibly incomplete) goal service specifications, provides a basis for failure-guided iterative reformulation of the goal service.

5 Summary and Discussion

We have addressed the problem of realizing a desired composite service through a parallel composition of a subset of available component services. Specifically, we have presented a theoretically sound and complete algorithm for constructing a choreographer that enables the interactions among component services to realize the behavior of the desired goal service. We use Symbolic Transition Systems (STSs) augmented with state variables over an infinite do-

[4]However, domain-specific information about component services (e.g., a partial-order among services), if available, can be used to reduce the complexity of composition by a priori ruling out some of the possible combinations that would otherwise have to be tried.

main and guards over transitions to model the services. A unique feature of the proposed approach is its ability to work with an abstract (possibly incomplete) specification of a desired goal service. In the event the goal service cannot be realized (either due to incompleteness of the specification provided by the developer or the limited functionality of the available component services), the proposed algorithm identifies the causes for failure and communicates them to the service developer. The resulting information guides further iterative reformulation of the goal service until a composition that realizes the desired behavior is realized or the user chooses to abort. These results complement our previous results on the sequential composition of services [10].

So far, our framework for modeling service composition and execution has focused on services which demonstrate a deterministic behavior without loops. Handling non-deterministic behavior that often characterizes real-world services is an important area of ongoing research. We also plan to investigate the complexity of incorporating services with loops in our composition algorithm.

In this paper, we have assumed that the component services are all specified using variable, function, relation names and constants based on common semantics, an assumption that is unlikely to hold in the case of services offered by autonomous service providers. In this context, approaches based on inter-ontology mappings to bridge the semantic gap between semantically heterogeneous services [11] deserve further investigation. We also assumed that the component services are published using STSs. In practice, these specifications can be obtained from service descriptions provided in high-level languages such as BPEL or OWL-S by applying translators proposed in [13, 17]. However, in our case, the translators in [13, 17] will have to be enhanced to handle pre-conditions of atomic functions which correspond to guards in transitions of STSs.

The practical feasibility of approaches to any approach to automated service composition is ultimately limited by the computational complexity of the service composition algorithms. Hence, approaches to reducing the number of candidate compositions that need to be examined e.g., by exploiting domain specific information to impose a partial-order over the available services, or reducing the number of goal reformulation steps needed by exploiting relationships among failure causes or between failure causes and services or between services needs further investigation.

Other work in progress is aimed at translating the choreographer into executable code for realizing the composite service. More details are available at `http://www.cs.iastate.edu/~jpathak/moscoe.html`.

Acknowledgment. This research has been supported in part by the Iowa State University Center for Computational Intelligence, Learning & Discovery (`http://www.cild.iastate.edu`), NSF-ITR grant 0219699 to Vasant Honavar, NSF grant 0509340 to Samik Basu, and NSF grant 0541163 to Robyn Lutz.

References

[1] V. Agarwal, K. Dasgupta, and et al. A Service Creation Environment Based on End to End Composition of Web Services. In *14th Intl. Conference on World Wide Web*, pages 128–137. ACM Press, 2005.

[2] S. Basu, M. Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. M. Verma. Local and Symbolic Bisimulation Using Tabled Constraint Logic Programming. In *Intl. Conference on Logic Programming*, volume 2237, pages 166–180. Springer-Verlag, 2001.

[3] B. Benatallah, Q. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[4] D. Berardi, D. Calvanese, D. G. Giuseppe, R. Hull, and M. Mecella. Automatic Composition of Transition-based Semantic Web Services with Messaging. In *31st Intl. Conference on Very Large Databases*, pages 613–624, 2005.

[5] S. Dustdar and W. Schreiner. A Survey on Web Services Composition. *International Journal on Web and Grid Services*, 1(1):1–30, 2005.

[6] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *13th Intl. conference on World Wide Web*, pages 621–630. ACM Press, 2004.

[7] R. Hull and J. Su. Tools for Composite Web Services: A Short Overview. *SIGMOD Record*, 34(2):86–95, 2005.

[8] R. Kumar, C. Zhou, and S. Basu. Finite Bisimulation of Reactive Untimed Infinite State Systems Modeled as Automata with Variables. In *American Control Conference*, 2006.

[9] J. Pathak, S. Basu, R. Lutz, and V. Honavar. MoSCoE: A Framework for Modeling Web Service Composition and Execution. In *IEEE 22nd Intl. Conference on Data Engineering Ph.D. Workshop*, page x143. IEEE CS Press, 2006.

[10] J. Pathak, S. Basu, R. Lutz, and V. Honavar. Selecting and Composing Web Services through Iterative Reformulation of Functional Specifications. In *18th IEEE International Conference on Tools with Artificial Intelligence*, 2006.

[11] J. Pathak, N. Koul, D. Caragea, and V. Honavar. A Framework for Semantic Web Services Discovery. In *7th ACM Intl. Workshop on Web Information and Data Management*, pages 45–50. ACM press, 2005.

[12] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *19th Intl. Joint Conferences on Artificial Intelligence*, pages 1252–1259, 2005.

[13] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *15th Intl. Conference on Automated Planning and Scheduling*, pages 2–11, 2005.

[14] D. Skogan, R. Grønmo, and I. Solheim. Web Service Composition in UML. In *8th IEEE Intl. Enterprise Distributed Object Computing Conference*, pages 47–57. IEEE Press, 2004.

[15] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated Discovery, Interaction and Composition of Semantic Web Services. *Journal of Web Semantics*, 1(1):27–46, 2003.

[16] J. Timm and G. Gannod. A Model-Driven Approach for Specifying Semantic Web Services. In *3rd Intl. Conference on Web Services*, pages 313–320. IEEE press, 2005.

[17] P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *3rd Intl. Semantic Web Conference*, pages 380–394. Springer-Verlag, 2004.