Process Mediation in an Extended Roman Model

Gösta Grahne and Victoria Kiricenko

Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada H3G 1M8
grahne,kiricen@cs.concordia.ca

1 Introduction

A *mediator* is a software module that provides sharing of services and and agglomeration of resources into complex services. Mediators will play a pivotal role in successful infrastructures for Semantic Web Services. *Process mediation* in Web Services involves issues of process compatibility and composability. Evolving standards of web services, such as the web service Execution Environment (WMSX [21]) focus mostly on compatibility issues in a B2B environment, whereas the problem of dynamic composition of web services is still maturing in the research community. In this paper we contribute to the composition aspect of process mediation.

Web service process specification has reached well accepted standards, such as the Business Process Execution Language (BPEL4WS [9]). Moreover, it has recently been shown that *process algebra* provides a useful interpretation of BPEL specifications. Works such as [12, 19] provide the mapping from BPEL to process algebra, and show how the algebra can be used to describe Web Services during design stage. These works also demonstrate how process algebraic descriptions can be extracted from existing Web Services for reverse engineering purposes. This enables the use of numerous verification tools that are available for process algebra (e.g. LOTOS) in web service development.

When it comes to the question of composability of web services some of the most fruitful results have been achieved within the *Roman* model [3, 2, 7, 11, 14, 5], an evolving framework that takes a highly computational approach to Web Service composition. In this paper we extend the Roman model by incorporating features from process algebra that will allow a fuller use of important functionalities of Web Services, such as parallelism, nondeterminism, and task decomposition.

Our contributions are as follows:

1. We develop a rigorous extension to the Roman model, and give a formal semantics based on the process algebraic notion of simulation. Our extension not only allows for fuller coverage of the standard languages used for description and execution of web wervices in practice, but also unifies the approaches of modeling and mediating web services. Moreover, our extension

allows for formal verification of mediated web services though the use of the numerous tools available for process algebra.
2. We give an algorithm for mediating a requested Web Service from a resource pool of more basic services.

Process mediation is a complex task, and our work certainly does not address all the problems and all the mediation scenarios that may appear in the Web Service context. In the concluding section we outline some extensions that we are currently investigating.

2 Behavioral Models of Web Services

Web Services are distributed and independent pieces of software working together to achieve given tasks. The Business Process Execution Language (BPEL [9]) is a notation for describing executable business process behaviors. Such behavioral signatures provide the foundation for composing Web Services.

Behavioral signatures are often formalized as state based (infinite) labeled transition systems, i.e. (infinite) labeled graphs. Vertices represent processes and labeled edges represent activity. The node that an edge is incident upon represents the state that the process has evolved to, after performing the action of the label. This approach is also adopted in the Roman model [3, 2, 7, 11, 14, 5], with the restriction that the transition graphs are tree structured.

Process algebra is a mathematical framework for reasoning about behavioral signatures. Numerous processes algebras have been proposed and studied in the literature. Basic ones include CCS, CSP, ACP, extensions are π-calculus and timed CSP [10]. LOTOS [16] is for one of the most expressive process algebras, and it comes with an industrial strength suite of tools for specification and verification. Recently, Ferrara [12] has given a two-way mapping that allows an automated translation between BPEL and LOTOS, thus making the LOTOS suite available for Web Service development.

The "urelements" in all process algebras is a finite set of *atomic processes*. Although syntactically different, all process algebras share a core set of basic constructs for process evolution, namely, sequential composition, non-deterministic choice, parallel composition, communicating (synchronized) composition, and recursion [10]. In our framework we use these core constructs, We build on the Roman model, which provides a basic framework for Web Service specification and verification. The Roman model is a robust foundation, and it is currently being extended with e.g. asynchronous message exchange [6], value passing [5], temporal constraints [14], Presburger constraints, discrete time, and non-regular processes [11].

In the next section we give our extension of the Roman model. In order to have a uniform framework, we build our extension from "first principles." Due to space limitations, some proofs are omitted. They can be found in a longer version of the paper at the authors' websites.

3 An Enhanced Roman Model

Let $\mathbb{N} = \{1, 2, 3, \ldots, \}$. By $\mathbb{N}^*$ we denote the set of all *finite strings* over $\mathbb{N}$. The empty string over $\mathbb{N}^*$ is denoted by ϵ. A *tree domain* D is a non-empty subset of $\mathbb{N}^*$ that is closed under the prefix relation. (i.e. if $u.w \in D$ then $u \in D$).

Now let Q be a *state space*, $F \subseteq Q$ a designated set of *final states*, and Σ a finite set of *actions*, Then an *execution tree* T is a function $T : D \to Q \times (\Sigma \cup \lambda)$, such that $T(\epsilon) = (q, \lambda)$, for some $q \in Q$. The *root* of T is ϵ. A *leaf node* of T is an element $w \in dom(T)$, such that for all $i \in \mathbb{N}$, $wi \notin dom(T)$.

The intuition is that the root is the start state of the service, and if (p, b) is a child of (q, a), then the service was in state p, and went into state q after having executed action a. It came into state p after having executed action b, and at the root it has not yet executed any actions. When the service is satisfying a request, it moves from the root state to some final state, while executing the actions along the path.

We shall in the sequel denote D by $dom(T)$. Figure 1 shows an example of a (finite) execution tree. For this T, we have $dom(T) = \{\epsilon, 1, 2, 11, 12, 13, 21, 121\}$, shown to the left, $F = \{p_0, p_1, p_3\}, Q = \{p_0, p_1, p_2, p_3, \}$, and $\Sigma = \{a, b, c\}$. To the right we show the function T graphically, e.g. $T(12) = (p_2, a)$. The leaves are $\{11, 121, 13, 21\}$. Thus this service T can satisfy the requests $\{b, bb, bab, ba, ca\}$.

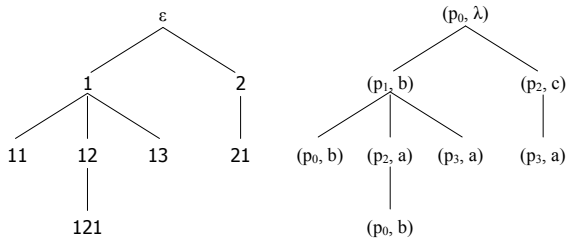


Fig. 1. A tree domain and a tree

To compare the capabilities of Web Services, we need a notion of equivalence, that tells us when two Web Services can perform the same tasks. Research in process algebra has revealed a wide spectrum of equivalences [10]. At one end is *trace* equivalence: two processes are considered equivalent if they can perform the same sequence of tasks. Trace equivalence represents a black box view of the processes. At the other end of the spectrum is *simulation* equivalence. This equivalence comes in various flavors, but the basic idea represents a game theoretic view: two processes are equivalent if they can match each others task sequences action by action.

Previously, in works on the Roman model [3, 2, 11, 14], a notion corresponding to trace inclusion has been used implicitly or explicitly. However, research in formal specification (see e.g. [10]) has shown that trace equivalence (two-way trace inclusion) is not always adequate. Consider the following example.

Example 1. An on-line computer store allows its users to search its product catalogs, to make a selection and then buy the selected item, or to save the results of the search for later use. Both execution trees in Figure 2 could be the web service for (this aspect of) the online computer store, as they both contain the same sets of traces ($\{search.buy, search.save\}$). However, in the execution tree to the right, lets call it T', a user has a choice of executing either *buy* or *save* after performing the *search* action, while in the execution tree to the left, called T, the web service makes a nondeterministic choice when a user initiates the *search* action. Note that one of the nondeterministic choices in T allows the user to subsequently initiate only the *buy* action and the other choice allows him to initiate only the *save* action, whereas in T' both the *buy* and the *save* action are enabled after the *search* action. It is clear that a notion of inclusion finer then trace inclusion is needed to distinguish between these two execution trees. □

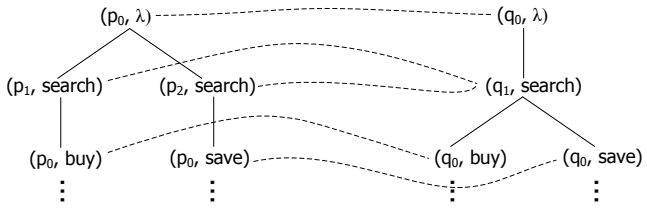


Fig. 2. Forward simulation $T \preceq T'$

We shall therefore use a notion derived from bisimulation, called *forward simulation* [17]. While bisimulation and its variants have received intense attention in the research community, forward simulation is much less investigated.

To define forward simulation formally, let Q and Q' be state spaces, with final states F and F', respectively, and let Σ be an alphabet. Let $T : dom(T) \rightarrow Q \times \Sigma$ and $T' : dom(T') \rightarrow Q' \times \Sigma$ be execution trees. We say that T can be *forward simulated* by T', denoted $T \preceq T'$ if there exists a *simulation relation* $\mathscr{R} \subseteq dom(T) \times dom(T')$, satisfying:

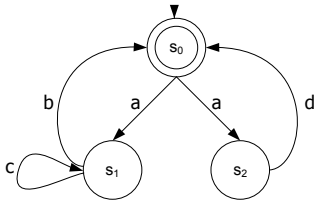
1. $(\epsilon, \epsilon) \in \mathscr{R}$
2. If $(w, w') \in \mathscr{R}$, for some $w \in dom(T)$, $w' \in dom(T')$, and $T(wi) = (s, a)$, for some $i \in \mathbb{N}$, then there exists $j \in \mathbb{N}$ and $s' \in Q'$ such that $(wi, w'j) \in \mathscr{R}$ and $T'(w'j) = (s', a)$.
3. For all $(w, w') \in \mathscr{R}$, where $T(w) = (p, a), T'(w') = (q, b)$, for some $a, b \in \Sigma$, if $p \in F$ then $q \in F'$.

Example 2. Consider again the two execution trees given in Example 1. Forward simulation requires that the simulating web service can simulate the simulee in a lock-step fashion. As we can see from the Example 1, T cannot simulate T'. On the other hand T' can simulate T. For every action that T takes,

T' can take the same action, and has at least the options of T for subsequent actions. In Figure 2 the dotted lines illustrate the simulation relation $\mathscr{R} = \{(\epsilon, \epsilon), (1, 1), (2, 1), (11, 11), (21, 12), \ldots\}$. □

Finite State Machines. So far, we have described web services on an abstract semantic level. In practice, web services need to be finitely specified or implemented. Following the Roman model, we shall represent a web service by a finite state machine.

Let Σ be a finite set of actions, as before. Actions will be denoted by letters $a, b, c \ldots$, and the empty string over Σ is denoted λ. A (nondeterministic) *Finite State Machine* (FSM) is a quintuple $A = (Q, \Sigma, \delta, p_0, F)$ where Q is the finite set of *states* of A, Σ is the finite set of *possible actions*, $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*, $s \in Q$ is the initial state, and F is the set of final states of A (see e.g. [15]).



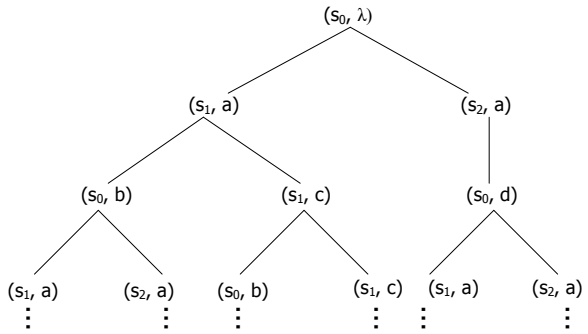


Fig. 3. An FSM an its execution tree

It should be noted that as opposed to the initial Roman model [3], our definition of an FSM allows for nondeterminism. This approach is more realistic as it is computationally closer to the specification languages typically used for description of the web services. Moreover, it better captures the dynamic, not always fully reliable, and partially redundant nature of web services. Nondeterminism has also been adopted in e.g. [11, 14], within the scope of the Roman model.

Given an FSM $A = (\{p_0, p_1, \ldots, p_n\}, \Sigma, \delta, p_0, F)$, its *execution tree* T_A is mapping $T_A : dom(T_A) \rightarrow Q \times \Sigma$, where T_A and $dom(T_A)$ are defined inductively as:

1. $dom(T_A) = \epsilon$ and $T_A(\epsilon) = (p_0, \lambda)$
2. If w is a leaf of $dom(T_A)$, with $T_A(w) = (p_j, a)$, and (p_j, b_1, p_{i_1}), $(p_j, b_2, p_{i_2}), \ldots, (p_j, b_k, p_{i_k}) \in \delta$ are all the transitions emanating from p_j, with $i_1 < i_2 < \ldots < i_k$, then extend $dom(T_A)$ and T_A as follows:
   (a) Add $w1, w2, \ldots, wk$ to $dom(T_A)$.
   (b) Extend T_A by setting $T_A(w1) = (p_{i_1}, b_1), T_A(w2) = (p_{i_2}, b_2), \ldots, T_A(wk) = (p_{i_k}, b_k)$.

Figure 3 shows an FSM and part of its execution tree.

We can now use forward simulation to compare FSM's. We say that a web service specified by an FSM A can be forward simulated by a web service specified by an FSM A', if $T(A) \preceq T(A')$.

Given the fact that execution trees are (typically) infinite, we are interested in a finite characterization of when $T(A) \preceq T(A')$.

Let $A = (Q, \Sigma, \delta, p_0, F)$ and $A' = (Q', \Sigma, \delta', q_0, F')$. We denote (with slight abuse of notation) $A \preceq A'$, if there exists a *simulation relation* $\mathscr{R} \subseteq Q \times Q'$ between the states of A and A', satisfying:

1. $(p_0, q_0) \in \mathscr{R}$
2. If $(p, q) \in \mathscr{R}$, and $(p, a, s) \in \delta$, for some $s \in Q$, then there exists a $t \in Q'$, such that $(q, a, t) \in \delta'$ and $(s, t) \in \mathscr{R}$.
3. For all $(p, q) \in \mathscr{R}$, if $p \in F$ then $q \in F'$.

Example 3. Consider the two web services defined by FSM's A and B in Figure 4. These FSM's are specifications of the execution trees in Figure 2. As expected, B can forward simulate A, but not vice versa. The dotted lines illustrate the simulation relation $\mathscr{R} = \{(p_0, q_0), (p_1, q_1), (p_2, q_1)\}$. □

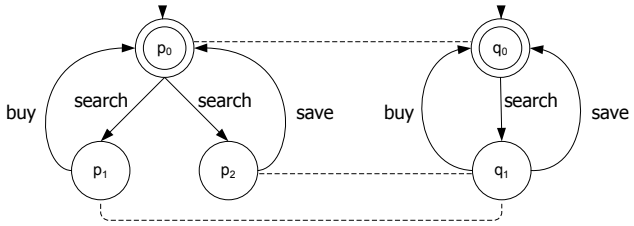


Fig. 4. Forward simulation $A \preceq A'$

It is easy to show that the two notions of forward simulation coincide.

Lemma 1. $A \preceq A' \Leftrightarrow T_A \preceq T_{A'}$

4 Web Service Communities

A *Web Service Community* is a set of cooperating Web Services $\{T_1, T_2, \ldots, T_k\}$, where each sub-service T_i is an execution tree with state space Q_i and action alphabet Σ_i. The action alphabet of the community is $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_k$.

Intuitively, actions of a Web Service Community are enacted by executing actions of the sub-services, either sequentially, or in parallel. This is formalized using a merge operator, inspired by [18].

The merge operator is a significant extension to the classical Roman model. Firstly, it allows for simultaneous execution of actions by two or several different services. This is a feature that is as essential to Web Services as it is to any distributed computations. Note that BPEL supports the `<flow>` construct that allows the specification of one or more activities to be performed concurrently.

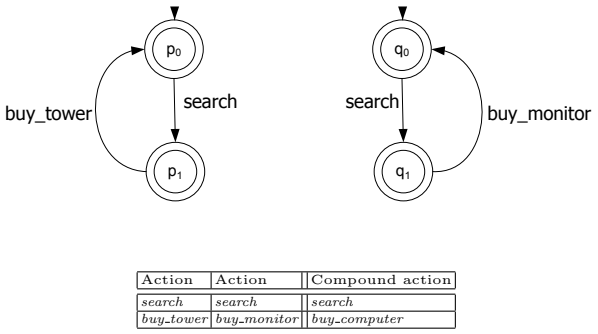
Secondly, the merge allows us to define processes that are composed of two or more sub-processes. This feature can be achieved in BPEL by using links within concurrent activities of the `<flow>` construct.

Formally, let a and b be actions. Then the *merge* of a and b, denoted $a \| b$, consists of either executing a followed by b, denoted $a.b$, or by executing $b.a$, (both $a.b$ and $b.a$ are sequential executions), or executing them in parallel. For the parallel execution we need a partial function $\gamma \colon \Sigma \times \Sigma \to \Sigma$. If $\gamma(a, b) = c$, then the parallel execution of a and b is visible externally as an action c. We demand that, if defined, $\gamma(a, b) = \gamma(b, a)$, and $\gamma(a, \gamma(b, d)) = \gamma(\gamma(a, b), d)$, for all $a, b, d \in \Sigma$, as γ models parallel execution (see [18]).

We now have two types of uses of the merge operator: The first is a parallel execution of one action. In this case we have $\gamma(a, a) = a$, as in the *search* action in Figure 5. The second use of merge is for task decomposition. When $\gamma(a, b) = c$ we take this to mean that externally visible action c is achieved internally by the community through executing a and b in parallel. The *buy_computer* action in Figure 5 is an example of an action achieved internally by two parallel actions, namely *buy_tower*, and *buy_monitor*.

Example 4. Consider the two Web Services defined by the FSMs A and A' given in the Figure 5 to the left and to the right, respectively. The two web services together form an web service community $\{A, A'\}$ and the figure also shows the merge capabilities of the community in the form a Table for the γ function.

The service A allows the user to search for and buy computer towers, while A' provides the same options for computer monitors. Now, what would be the semantics of this community? Intuitively, the executions of the individual web services can be interleaved in any possible order. In addition, with respect to the defined γ searches can be done in parallel, and simultaneously buying a tower and a monitor amounts to buying a computer. The merge of the A and A', with respect to the given γ is illustrated in Figure 6, □



Action	Action	Compound action
search	*search*	*search*
buy_tower	*buy_monitor*	*buy_computer*

Fig. 5. Community of FSM's

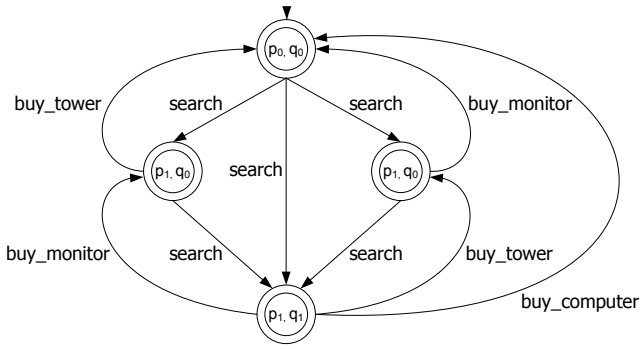


Fig. 6. Merge of community FSM's

5 Semantics of Web Service Communities

As the meaning of Web Service processes is given in terms of process trees, we need to define the community merge in terms of trees. We do this by extending the language-theoretic *shuffle* [15] with the γ-function. (Note that in classical process algebra the merge is defined axiomatically [10].)

Let $T : dom(T) \to Q \times \Sigma$, and $T' : dom(T) \to Q' \times \Sigma$, be trees, with final states $F \subseteq Q \subseteq \{1, \ldots, k\}^*$, and $F' \subseteq Q' \subseteq \{1, \ldots, m\}^*$. To define the *merge*, $T \| T'$, of trees T and T' we need to specify $dom(T \| T')$. For this, let $w \in dom(T)$ and $u \in dom(T')$. We set:

1. $\epsilon \| w = \{w\}$, $\epsilon \| u = \{u\}$, and $\epsilon \| \epsilon = \{\epsilon\}$
2. if $w = w'.i$ and $u = u'.j$, for some $i \in \{1, \ldots, k\}$, $j \in \{1, \ldots, m\}$ then $w \| u = (w' \| u)i \cup (w \| u')k + j$.
   Furthermore, if $\gamma(t(w), t'(u))$ is defined, then add the element $(w' \| u')k + m + 2^i \cdot 3^j$ to $w \| u$.

We can now define

$$dom(T \| T') = \{w \| u : w \in dom(T), u \in dom(T')\}.$$

We regard $dom(T \| T')$ as a subset of $\{1, \ldots, k + m + 2^k \cdot 3^m\}^*$.

Lemma 2. *$dom(T \| T')$ is a tree domain.*

For an arbitrary string w over $dom(T||T')$, we define w_l as the projection of w to the subsequence of symbols $\leq k$, and w_r as the projection of w onto the symbols $> k$ and $\leq m$.

By $T_Q(w)$ we mean p, where $T(w) = (p, \sigma)$, and by $T_\Sigma(w)$ we mean σ. Similarly for T'.

Now we can define $T \| T'(\epsilon)$ as $(\langle T_Q(\epsilon), {T'}_Q(\epsilon)\rangle, \lambda)$, and for all $wi \in dom(T \| T')$,

$$T \| T'(wi) = \begin{cases} (\langle T_Q(w_l i), {T'}_Q(w_r)\rangle, T_\Sigma(w_l i)), & i \leq k \\ (\langle T_Q(w_l), {T'}_Q(w_r i)\rangle, {T'}_\Sigma(w_r i)), & k < i \leq k + m \\ (\langle T_Q(w_l i), {T'}_Q(w_r i)\rangle, \gamma(T_\Sigma(w_l i), {T'}_\Sigma(w_r i)), & i \geq k + m + 6. \end{cases}$$

Example 5. Let us return to the community of the two services in Examples 4. Two execution trees T and T', corresponding to A and A' are given in Figure 7. The semantic merge of these two trees are shown in in Figure 8.

The semantic merge has the following properties:

Theorem 1. *$T \| T' = T' \| T$, up to isomorphism.*

Theorem 2. *$(T \| T') \| T'' = T \| (T' \| T'')$, up to isomorphism.*

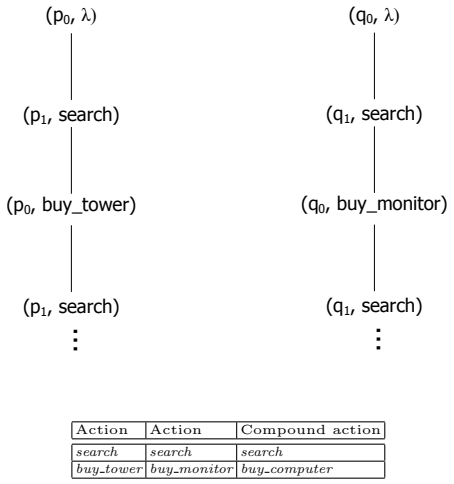
Given a web service community $(T_1, \ldots, T_k)$, we can now define its behavior as $T_1 \| \ldots \| T_k$

Theorem 3. *The semantics $T_1 \| \ldots \| T_k$ is uniquely defined, up to isomorphism.*

When the web services in a community are defined as finite state machines $A = (Q, \Sigma, \delta, p_0, F)$, and $A' = (Q', \Sigma', \delta', q_0, F')$, we can construct an FSM $A \| A' = (Q_\|, \Sigma_\|, \delta_\|, s_\|, F_\|)$, where $Q_\| = Q \times Q'$, $\Sigma_\| = \Sigma \cup \Sigma'$, $s_\| = (p_0, q_0)$, $F_\| = F \times F'$, and $\delta_\| \subseteq (Q \times Q') \times (\Sigma \cup \Sigma') \times (Q \times Q')$ is defined as follows: $\forall\ (p, a, q) \in \delta$, $\forall\ (q, b, q') \in \delta'$, we have the transitions $(\langle p, q\rangle, a, \langle p', q\rangle)$ and $(\langle p, q\rangle, b, \langle p, q'\rangle)$ in $\delta_\|$. Furthermore, if $\gamma(a, b) = c$ then $\delta_\|$ also contains $(\langle p, q\rangle, c, \langle p', q'\rangle)$.

By an encoding similar to those used in the proofs of Theorems 1 and 2 the following can be shown.

Theorem 4. *$T_{A \| A'} = T_A \| T_{A'}$* □



Action	Action	Compound action
search	*search*	*search*
buy_tower	*buy_monitor*	*buy_computer*

Fig. 7. Web service community

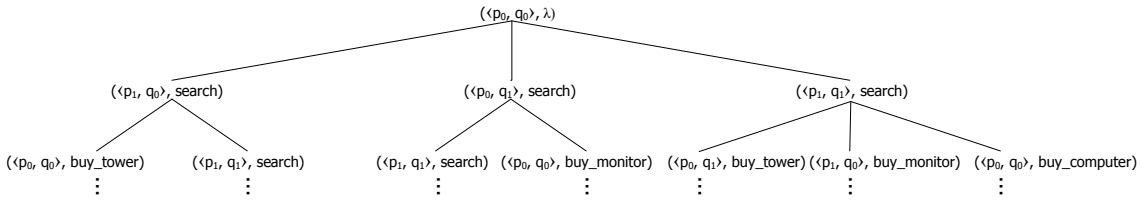


Fig. 8. Web service community execution tree

6 Synthesizing and Orchestrating Client Requests

The *raison d'être* of a Web Service community is to provide services to their clients. When a client requests a certain service there might be no individual Web Service in the community that can deliver it directly. However, as we already discussed in the previous section, the actions of individual Web Services in the community can be interleaved in any order and also executed in parallel. Therefore, it is possible that a *composite* Web Service can satisfy the client's request.

Service composition involves two main issues. The first, often called *composition synthesis*, is concerned with synthesizing a new composite Web Service, thus producing a specification of how to coordinate the Web services available in the community to provide a specified service. (This is the "compile time" phase.) The second, referred to as *orchestration*, is concerned with coordinating the correct execution of of an instance of a client request in the synthesized composition. (That is, the "run time" phase.)

Let us first formally define a service requested by a client, and what it means for Web Service community to be able to satisfy the request.

A *client request* is an execution tree $T_0\colon\ dom(T_0) \to Q_0 \times \Sigma$. We say that a web service community $\{T_0, \ldots, T_k\}$ can *satisfy* the client if $T_0 \preceq\ T_1 \| T_2 \| \ldots \| T_k$.

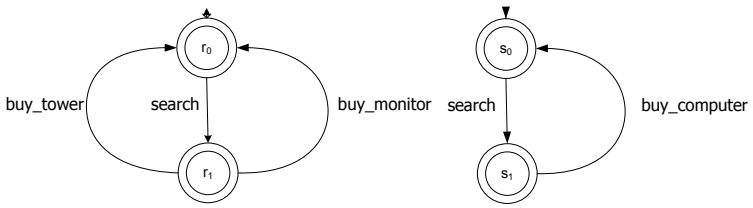


Fig. 9. Two web service clients

In practice each subservice T_i is specified by an FSM A_i, and the client by an FSM A_0. We are thus interested in whether $T_{A_0} \preceq\ T_{A_1} \| T_{A_2} \| \ldots \| T_{A_k}$. For this we need

Theorem 5. *$T_{A_0} \preceq T_{A_1} \| T_{A_2} \| \ldots \| T_{A_k}$ if and only if $A_0 \preceq\ A_1 \| A_2 \| \ldots \| A_k$.*

Proof. By Lemma 1, we have $A_0 \preceq\ A_1 \| A_2 \| \ldots \| A_k$ iff $T_{A_0} \preceq\ T_{A_1 \| A_2 \| \ldots \| A_k}$. By repeatedly applying Theorem 4 we have that $T_{A_1 \| A_2 \| \ldots \| A_k}$ is isomorphic to $T_{A_1} \| T_{A_2} \| \ldots \| T_{A_k}$. The claim now follows. □

Let the client FSM be $A_0 = (Q_0, \Sigma, \delta_0, p_0, F_0)$ and let the community be $\mathscr{A} = \{A_1, \ldots, A_k\}$, where $A_i = (Q_i, \Sigma_i, \delta_i, q_{0_i}, F_i)$. We denote by $\overrightarrow{q_0}$ the tuple $\langle q_{0_1}, q_{0_2}, \ldots, q_{0_k} \rangle$, where $q_{0_i} \in Q_i$ is the start state of A_i. Similarly, $\overrightarrow{q}$ denotes a tuple of states (not necessarily initial) from $A_1, A_2, \ldots A_n$.

The following algorithm computes the simulation relation $A \preceq \mathscr{A}$.

Forward-Sim$(A, \mathscr{A})$

```
1  U ← ∅
   ▷ pairs of states (p, q) such that p cannot be simulated by q.
2  repeat
3          R ← ∅
           ▷ simulation relation
4          W ← ∅
           ▷ pairs of states (p, q) visited more than once
5          S ← {(p_0, q_0→)}
           ▷ pairs of states (p, q) in the currently checked sequence
           Reliability ← TRUE
6          Result ← Check(p_0, q_0→)
7      until Result = FALSE ∨ Reliability = TRUE
8  if Result = FALSE
9      then R ← ∅
10 return R
```

```
CHECK(p, q⃗)
 1  if p ∈ F ∧ ∃q_{i_j} ∈ q⃗ such that q_{i_j} ∉ F_j
 2     then U ← U ∪ {(p, q⃗)}
 3          S ← S \ {(p, q⃗)}
 4          return FALSE
 5  if {(p, a, p′) : (p, a, p′) ∈ δ, a ∈ Σ, p′ ∈ Q} = ∅
 6     then R ← R ∪ {(p, q⃗)}
 7          S ← S \ {(p, q⃗)}
 8          return TRUE
 9  for each (p, a, p′) ∈ δ
10        do V ← {(q_{m_o}, ..., q_{l_j}, ..., q_{n_k}) :
                        q⃗ = (q_{m_o}, ..., q_{i_j}, ..., q_{n_k}) ∧ (q_{i_j}, a, q_{l_j}) ∈ δ_j}
           ∪
                 {(q_{m_o}, ..., q_{r_j}, ..., q_{s_h}, ..., q_{n_k}) :
                        q⃗ = (q_{m_o}, ..., q_{i_j}, ..., q_{l_h}, ..., q_{n_k})∧
                        (q_{i_j}, b, q_{r_j}) ∈ δ_j ∧ (q_{l_h}, c, q_{s_h}) ∈ δ_h ∧ γ(b, c) = a}
11           if V = ∅
12              then U ← U ∪ {(p, q⃗)}
13                   S ← S \ {(p, q⃗)}
14                   if (p, q⃗) ∈ W
15                      then Reliability = FALSE
16                   return FALSE
17           Flag ← FALSE
18           while V ≠ ∅
19                do
20                   V ← V \ {q⃗′}
21                   if (p′, q⃗′) ∉ U
22                      then if (p′, q⃗′) ∈ R
23                                then Flag ← TRUE
24                                else if (p′, q⃗′) ∉ S
25                                        then S ← S ∪ {(p′, q⃗′)}
26                                             Flag ← Flag ∨CHECK(p′, q⃗′)
27                                        else W ← W ∪ {(p′, q⃗′)}
28                                             Flag ← TRUE
29  if Flag = FALSE
30     then S ← S \ {(p, q⃗)}
31          U ← U ∪ {(p, q⃗)}
32          if (p, q⃗) ∈ W
33             then Reliability = FALSE
34          return FALSE
35  R ← R ∪ {(p, q⃗)}
36  return TRUE
```

The algorithm is based on the partial depth first search approach used for "on-the-fly" verification of behavioral equivalences introduced in [13].

The procedure CHECK starts with a pair of start states and traverses the client FSM constructing "on-the-fly," as needed, part of the community FSM in depth first order. The construction of the next state of $\mathscr{A}$ is based on the transitions of the component FSMs $A_1, \ldots A_k$, and on the γ-table of the community.

First the procedure checks that if the state of A_0 is a final state then the corresponding state of $\mathscr{A}$ is also a final state, that is, all of the component states are final. Next, the procedure checks whether the state of A_0 has any outgoing transitions, and, if this is not the case, then the corresponding state of $\mathscr{A}$ can definitely simulate it. After that, it picks one after another the transitions going out of the state of A_0 and checks if there is at least one corresponding transition in $\mathscr{A}$. If the algorithm finds such transition, it checks the pair of states that these transitions are incident upon. If no corresponding transition exists in $\mathscr{A}$, then, obviously, $\mathscr{A}$ cannot simulate A_0

The list S of transitions that are considered in the current traversal is maintained in order not to go in cycles. When the algorithm sees the same pair of states again it does not visit them again, but starts returning and constructing the simulation relation. Thus, the pairs of states are visited in prefix order, while the conditions for the simulation relation are checked in postfix order. Consequently it is possible to reach a pair of states $(p, \overrightarrow{q})$ which have already been visited, but not yet determined to be in the simulation relation. In this case the algorithm makes an optimistic assumption that $(p, \overrightarrow{q})$ will be determined to be in $\mathscr{R}$ at a later time. When this pair is eventually analyzed by the algorithm (the algorithm maintains a set W of such pairs), if it is determined not to belong to the simulation relation, then the procedure sets the flag *Reliability* to FALSE, which means that the simulation relation that the algorithm has constructed is not guaranteed to be correct as the optimistic assumption used by the algorithm was wrong.

If the algorithm decided that A_0 cannot be simulated by $\mathscr{A}$ this decision is always reliable because not assumptions are used to decide the FALSE. However, if the procedure CHECK returned TRUE the *Reliability* has to be TRUE as well, otherwise the simulation relation constructed by the procedure has to be discarded and CHECK has to be called again. In order to omit repeating the same dead-end traversals the algorithm maintains global list U of pairs that are determined to be not in the simulation relation by previous calls to CHECK.

We note that Shukla et al. [20] previously have given a decision procedure for testing whether one FSM can be forward simulated by another. This is achieved by a reduction to a variant of Horn clause satisfiablity. However, their procedure does not construct an actual simulation relation.

Theorem 6. *Algorithm* FORWARD-SIM$(A_0, \mathscr{A})$ *runs in time* $\mathcal{O}(|\delta_0| \times |\delta|)$*, where* $|\delta_0|$ *and* $|\delta|$ *denote the sizes of the transition relations in* A_0 *and* $\mathscr{A}$*, respectively.*

Since the size of the community FSM $\mathscr{A}$ is at most exponential in the size of the component FSM's $A_1, \ldots, A_k$, the following holds.

Corollary 1. *The algorithm* FORWARD-SIM *runs in EXPTIME.*

Client Action	Community Action
$r_0 \xrightarrow{search} r_1$	$p_0 \xrightarrow{search} p_1$ $q_0 \xrightarrow{search} q_1$
$r_1 \xrightarrow{by\_tow} r_0$	$p_1 \xrightarrow{by\_tow} p_0$
$r_1 \xrightarrow{by\_mon} r_0$	$q_1 \xrightarrow{by\_mon} q_0$

Fig. 10. Orchestration of left client

Fig. 11. from Figure 9

Client Action	Community Action
$s_0 \xrightarrow{search} s_1$	$p_0 \xrightarrow{search} p_1$ $q_0 \xrightarrow{search} q_1$
$s_1 \xrightarrow{by\_comp} s_0$	$p_1 \xrightarrow{by\_tow} p_0$ $q_1 \xrightarrow{by\_mon} q_0$

Fig. 12. Orchestration of right client from Figure 9

Theorem 7. *The algorithm* FORWARD-SIM *is sound and complete.*

Example 6. Consider a client given by the FSM on the left in the Figure 9, and the Web Service community $\{A_1, A_2\}$ from Figures 5 and 6.

Since we have $\gamma(search, search) = search$, allowing the searches to be executed in parallel on many services, it is easy to see that there indeed is a simulation relation from A_0 to $A_1 \| A_2$, namely $\mathscr{R}\{(r_0, \langle p_0, q_0\rangle), (r_1, \langle p_1, q_1\rangle)\}$. From this an orchestration engine (a Mealy-machine) can straightforwardly be constructed. For simplicity we illustrate the orchestration engine informally in the table in the Figure 10.

The need for γ can be easily seen, as this client cannot be simulated by the community in Figure 6, unless we use the look-ahead mechanism from [14]. However, look-ahead is not always possible, e.g. postponing the payment transaction in on-line shopping. □

In the full paper we show how a process algebra description can be obtained from the simulation relation computed by our algorithm. Then, the process algebra description can be automatically translated to executable BPEL-code, using the two way mapping between process algebra and BPEL given in [12]. The following example illustrates this, and points out important BPEL constructs that can be handled by our extended model, and which were not present in the *ESC* implementation of the classical Roman model [4].

Example 7. Consider the right client Figure 9. We can satisfy this client, as we have $\gamma(buy\_tower, buy\_monitor) = buy\_computer$. In this case the simulation relation from the client to $A_1 \| A_2$ is $\mathscr{R} = \{(s_0, \langle p_0, q_0\rangle), (s_1, \langle p_1, q_1\rangle)\}$

Note that γ allows composite services to be merged in the external action alphabet of a web service community, thereby also achieving ease of integration of new services into the community. The simulation relation for the right client is given informally as the table in the Figure 11.

The BPEL pseudo-code for the actual implementation of this service is given in Figure 13. Note that there are two `<flow>` constructs in this pseudo-code. The first one results from $\gamma(search, search) = search$ The two participating services are invoked to perform *search* in parallel. The second corresponds to $\gamma(buy\_tower, buy\_monitor) = buy\_computer$. □

7 Conclusions and Future Directions

We have develop a rigorous extension to the Roman model and given a formal semantics using a process algebraic approach [18, 10]. Process algebra works well at the stages of design and formal verification of web services [12, 19]. In our work we show that this approach gives advantages if used at the process mediation stage as well. Our extension allows for fuller coverage of the standard languages used for description and execution of Web Services in practice and unifies the modeling and mediating aspects of Web Services. Moreover, our extension allows for formal verification of mediated Web Services though the use of the numerous tools available for process algebra.

We gave an algorithm for mediating a requested Web Service from a resource pool of more basic services. The algorithm constructs the required composition "on the fly" without constructing the FSM for the entire Web Service community. The produced composition is complete in the sense that it covers all alternatives and the final decisions can be made at run time, based on the availability of the component services, network bandwidth, or some cost model. The algorithm runs in the exponential time which is the same as in the classical Roman model.

To see that the proposed solution is in fact practical consider the complete mediation procedure, which starts with the executable web services, produces the abstract descriptions (here is a possibility to formally verify some properties of the available services, if required), composes the abstract descriptions to produce a mediated service (at this stage also formally verifiable) and, finally, translates the mediated service to an actual executable web service.

We are currently working on the implementation of a prototype system where the mediator extracts algebraic descriptions from existing web services and the abstract specification of the produced mediated web service is translated into executable BPEL code, using the techniques in [12]. We are also working on further extensions of the formal model.

References

1. Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou, editors. *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings*. ACM, 2004.
2. Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic composition of e-services that export their behavior. In Maria E. Orlowska, Sanjiva Weerawarana, Mike P. Papazoglou, and Jian Yang, editors, *ICSOC*, volume 2910 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2003.
3. Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. A foundational vision of e-services. In Christoph Bussler, Dieter Fensel, Maria E. Orlowska, and Jian Yang, editors, *WES*, volume 3095 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 2003.
4. Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Esc: A tool for automatic composition of services based on logics

of programs. In Ming-Chien Shan, Umeshwar Dayal, and Meichun Hsu, editors, *TES*, volume 3324 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2004.
5. Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Massimo Mecella. Automatic composition of transition-based semantic web services with messaging. In Böhm et al. [8], pages 613–624.
6. Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW*, pages 403–410, 2003.
7. Daniela Berardi, Giuseppe De Giacomo, Maurizio Lenzerini, Massimo Mecella, and Diego Calvanese. Synthesis of underspecified composite -services based on automated reasoning. In Aiello et al. [1], pages 105–114.
8. Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors. *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 2005.
9. BPEL. Business process execution language for web services (version 1.1), May 2003.
10. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, 2001.
11. Zhe Dang, Oscar H. Ibarra, and Jianwen Su. Composability of infinite-state activity automata. In Rudolf Fleischer and Gerhard Trippen, editors, *ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 377–388. Springer, 2004.
12. Andrea Ferrara. Web services: a process algebra approach. In Aiello et al. [1], pages 242–251.
13. Jean-Claude Fernandez and Laurent Mounier. "on the fly" verification of behavioural equivalences and preorders. In Kim Guldstrand Larsen and Arne Skou, editors, *CAV*, volume 575 of *Lecture Notes in Computer Science*, pages 181–191. Springer, 1991.
14. Cagdas Evren Gerede, Richard Hull, Oscar H. Ibarra, and Jianwen Su. Automated composition of e-services: lookaheads. In Aiello et al. [1], pages 252–262.
15. J. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Language, and Computation*. Addison–Wesley, Reading, MA, 1979.
16. ISO. Lotos: a formal description technique based on the temporal ordering of observational behaviour. Technical Report 8807, International Standards Organisation, 1989.
17. Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.
18. R. Milner. A calculus on communicating systems. *Lecture Notes in Computer Science*, 92, 1980.
19. Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *ICWS*, pages 43–. IEEE Computer Society, 2004.
20. Sandeep K. Shukla, Harry B. Hunt III, Daniel J. Rosenkrantz, and Richard Edwin Stearns. On the complexity of relational problems for finite state processes (extended abstract). In Friedhelm Meyer auf der Heide and Burkhard Monien, editors, *ICALP*, volume 1099 of *Lecture Notes in Computer Science*, pages 466–477. Springer, 1996.
21. WSMX. Web service execution environment, June 2005.

```
<process name = "ComputerPurchase" ...>
   ...
   <partnerLinks>
      <partnerLink name = "customer" .../>
      <partnerLink name = "towerStore" .../>
      <partnerLink name = "monitorStore" .../>
      ...
   </partnerLinks>
   <variables>
      <variable name = "searchRequest" .../>
      ...
   </variables>
   ...
   <pick>
      <onMessage partnerLink = customer
         portType = ...
         operation = "searchRequest"
         variable = ...>
         <sequence>
            ...
            <flow>
               <invoke partnerLink = "towerStore"
                  portType = ...
                  operation = "searchRequest"
                  ... />
               <invoke partnerLink = "monitorStore"
                  portType = ...
                  operation = "searchRequest"
                  ... />
            </flow>
            ...
            <pick>
               <onMessage partnerLink = customer
                  portType = ...
                  operation = " buy_computerRequest"
                  variable = ...>
                  <sequence>
                     ...
                     <flow>
                        <invoke partnerLink = "towerStore"
                           portType = ...
                           operation = "buy_towerRequest"
                           ... />
                        <invoke partnerLink = "monitorStore"
                           portType = ...
                           operation = "buy_monitorRequest"
                           ... />
                     </flow>
                     ...
                  </sequence>
                  ...
               </onMessage>
               ...
            </pick>
            ...
         </sequence>
         ...
      </onMessage>
      ...
   </pick>
   ...
</process>
```

Fig. 13. BPEL pseudo-code for the orchestration of the right client in Figure 9