Online and Minimum-Cost Ad Hoc Delegation in e-Service Composition *

Cagdas E. Gerede Oscar H. Ibarra
Dept. of Computer Science
University of California
Santa Barbara, CA, 93106, USA

Bala Ravikumar
Dept. of Computer Science
Sonoma State University
Rohnert Park, CA 94928, USA

Jianwen Su
Dept. of Computer Science
University of California
Santa Barbara, CA, 93106, USA

Abstract

The paradigm of automated e-service composition through the integration of existing services promises a fast and efficient development of new services in cooperative business environments. Although the "why" part of this paradigm is well understood, many key pieces are missing to utilize the available opportunities. Recently "e-service communities" where service providers with similar interests can register their services are proposed towards realizing this goal. In these communities, requests for services posed by users can be processed by delegating them to already registered services, and orchestrating their executions. We use the service framework of the "Roman" model and further extend it to integrate activity processing costs into the "ad hoc" delegation computation. We investigate the problem of efficient processing of service requests in service communities and develop polynomial time ad hoc delegation techniques guaranteeing optimality.

1 Introduction

The framework of web services paints a bright picture for future software and applications development (see recent conferences such as [19, 2, 1, 9]). Much of the expected benefits come from a systematic approach of sharing program executions, a.k.a. services, as well as data (Microsoft's .NET, Sun's J2EE, IBM's WebSphere). Clearly, the ability of providing services is becoming critical in many business applications where the integration and performance management of business processes ultimately determine the success [14]. In spite of the demand from applications and practice, software system design still lacks the fundamental principles governing rigorous technical analysis in terms of functional adequacy and/or performance metrics [20]. A major challenge lies thus in the technical advancement in the areas of both formalisms for software design and analysis techniques. The goal of the present paper is to develop techniques and algorithms for automated composition of *e-services* (or equivalently, *web services*, *services*).

Service composition has its similarities with information system integration, workflow system design, and distributed computing [13, 17]. The main task is to assemble existing pieces in a way that the autonomous pieces will cooperate with each other. The goal is to facilitate a fast and efficient development of new services in cooperative business environments [13, 17]. Towards this goal, *e-service communities* are proposed [4, 3] where service providers with similar interests can register their services for the community use. When a user asks for the execution of some activities, the community may not have a capable service directly; however, the activities may be delegated to registered services so that the user need is satisfied.

Service composition, generally, consists of two main steps [4]. The first one, sometimes called *composition synthesis*, describes the process of (manually or automatically) computing a *specification* of how to coordinate the components to answer a service request. On the other hand, the second step, often referred to as *orchestration*, defines the actual run-time coordination of service executions, considering also data and control flow among services. This paper focuses on the synthesis part.

Recently, automated composition synthesis problem was studied in [4, 5, 11]. Given a set of descriptions of existing e-services (e.g., from a "UDDI++" repository) and a desired e-service, the problem is to construct a "delegator" that will coordinate the activities of those services to achieve the desired e-service. Also, all three studies use a model, often referred to as the "Roman" model, where e-services are represented as activity-based FSMs. Compared to these studies, one important difference in our approach is that we look at the problem of computing an optimum "on-the-fly" delegation for a given specific sequence of service requests instead of the problem of offline construction of a delegator achieving the behavior of a desired nonexisting e-service. Also, in our case, e-services are modelled as FSMs augmented with linear counters and we integrate activity processing costs to the model.

*Work by Gerede, Ibarra, and Su was supported in part by NSF grant IIS-0101134. Ibarra's research was also supported by NSF grants CCR-0208595 and CCF-0430945.

This paper makes the following contributions:

- We extend the "Roman" model in two ways: First, we model e-services as FSMs augmented with linear counters (i.e., linear storage) and illustrate the usefulness of the extended model. Secondly, We integrate activity processing costs into the model to capture the cases where different services process the same activities with different costs.
- We define the ad hoc delegation problem in e-service communities. Given a community of e-services, and a service request (sequence of activities), we study the problem of how to compute an optimal delegation of requests as the cheapest way to proces the requests using the community services in a collaborative manner.
- We show that for FSM only e-services, there is a linear time algorithm computing an optimal delegation, and that for linear counter e-services, the optimum ad hoc delegation problem can be solved in $O(log^2 n)$ space and in polynomial time in terms of n, where n is the length of the service request.

The remainder of the paper is organized as follows. In Section 2, we define our service model and delegation problem. In Section 3, we present the linear time delegation algorithm. In Section 4, we study the complexity of delegation for linear counter e-services and propose a polynomial time algorithm. Section 5 discusses related work and Section 6 concludes the paper.

2 A Model for E-services

We adopt the e-service framework presented in [4] (often referred to as the "Roman" model), because the model clearly defines e-services and provides a nice formal setting for a precise characterization of automatic composition of e-services. In this model, an *e-service* is a software artifact interacting with its clients (humans or other e-services). An *external service schema* describes the published service behavior represented as sequences of activities with constraints on their execution order, whereas an *internal schema* specifies the internal logic of the service meaning how the activities are actually executed. An *e-service instance* refers to one occurrence of an e-service among several independently running instances. Each instance conforms to its schema during its execution. We can informally describe the semantics of a service execution as follows: When a client first invokes a service instance, an "enactment" is created for the conversation. Then, the client interacts with the service instance by sending an activity request and waiting for the return information. On the basis of the returned information, she determines her next activity request. When the client doesn't have any more requests, she may explicitly terminate the enactment.

When an e-service is invoked by a client, each requested task can be performed by either executing certain actions on its own, or interacting with other e-services to *delegate* the processing to them. A *simple e-service* processes all requests from its client on its own, while a *composite e-service* invokes other e-services to answer the client requests.

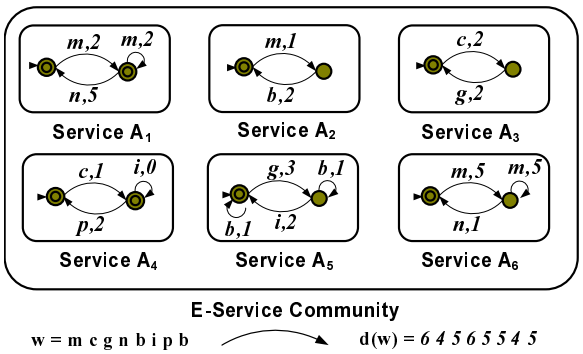

Figure 1. A Community of Genomics Services

An *e-service community* consists of an *activity alphabet* Σ and a set of e-services of similar interests. An e-service joins a community by registering its external schema in terms of the alphabet of the community (which can be done by defining a mapping from the service alphabet to the community alphabet, e.g., as described in [3]).

So far the model doesn't refer to any specific form of service schemas. As formalized in the next section, this study focuses on the services whose external service schemas are represented as *nondeterministic FSMs augmented with counters*. We mainly focus on external schemas; therefore, from now on we use the word "service" to actually mean an external service schema.

We also extend the model with processing costs. A *cost function* of a service describes how much it costs to perform an activity and these costs can be related, for instance, to the amount of money charged to a client or the communication time necessary for the processing.

Before the formal discussion of the service model, let's illustrate the described notions with an example.

Example 2.1 Figure 1 illustrates a community of genomics services. The community alphabet consists of the following activities: transcription factor binding search ($\mathbf{b}$), cluster and principal components analysis ($\mathbf{c}$), GenBank sequence retrieval ($\mathbf{g}$), promoter identification ($\mathbf{i}$), microarray analysis ($\mathbf{m}$), NCBI BLAST search ($\mathbf{n}$), and promoter model generator ($\mathbf{p}$). The community has 6 registered services each with different functionalities and processing costs. For instance, service A_4 can perform $\mathbf{c}$, $\mathbf{i}$ and $\mathbf{p}$ with costs $1, 0$ and 2 respectively. Each service (schema) is represented by an FSM as illustrated in Figure 1. In each FSM, the state directed by an arrow head without a tail represents the start state of the service when the enactment is created. Also, each double circle represents an accepting state where an enactment can be terminated successfully.

Suppose that a scientist would like to make an experi-

ment on promoter identification by performing the following activities in that order: microarray analysis, cluster and principal components analysis, Genbank sequence retrieval, NCBI BLAST search, transcription factor binding search, promoter identification, promoter model generator and transcription factor binding search (which can be described as a word $\mathbf{w} = \mathbf{mcgnbipb}$). Obviously, there is no single service that can process $\mathbf{w}$. However, it can be shown that services can be coordinated so that they can collaboratively process $\mathbf{w}$. Figure 1 shows the optimum delegation $\mathbf{d}(\mathbf{w})$ where, for instance, micro array analysis and NCBI BLAST is delegated to service 6.

Note that the delegation of an activity can be affected by the other activities in the sequence. For instance, in $\mathbf{d}(\mathbf{w})$, the activity $\mathbf{c}$ is delegated to A_4 instead of A_3 since $\mathbf{w}$ has the activity $\mathbf{p}$. In addition, the optimum delegation cannot simply be computed by selecting the service with the cheapest processing cost at each step. For example, the activity $\mathbf{m}$ should be delegated to A_6 instead of A_1, even though A_1's cost is much lower. The reason is that the activity $\mathbf{n}$'s lower cost in A_6 compensates the higher cost of $\mathbf{m}$. However, that wouldn't be the case if $\mathbf{w}$ had more than one $\mathbf{m}$.

One final point we would like to mention with this example is about the counters. Suppose that service A_4 doesn't want to let its users abuse the system by requesting many $\mathbf{i}$'s for free and would like to put the following constraint on the executions: the number of $\mathbf{i}$'s executed must be at most 4 times the total number of $\mathbf{c}$'s and $\mathbf{p}$'s. This kind of linear constraints can be easily checked if FSMs are augmented with linear bounded counters. For instance, in this case, the constraint can be checked if A_4 is augmented with 5 linear bounded counters, where 4 of them counts the total number of $\mathbf{c}$'s and $\mathbf{p}$'s and the last one is for the number of $\mathbf{i}$'s. ∎

Next, we formally define the notion of an e-service, and the optimal delegation problem.

Let Σ be a finite alphabet of symbols, each of which represents an *activity*. A *service request (or word) of length* $k \in \mathbb{N}$ over Σ is a sequence of k activities requested by a client. λ denotes the word of length 0. Let Σ^* be the set of all words over Σ. A *language* is a subset of Σ^*.

A *linear(-bounded) counter machine* (or simply, *linear CM*) is an FSM augmented with a finite number of counters (or integer variables). While making a transition from a state to another state, each counter (which can only have nonnegative values) can be incremented or decremented by 1 or unchanged, and can be tested for zero. The counters are restricted in that there is a constant c such that on any input of length n, the value stored in each counter during the computation is at most cn; thus, such counters are called *linear bounded* counters (Note that our restriction on the counters being linear can easily be relaxed to counters with values bounded by cn^k for constants c and k).

We will show that for a set of services modeled as linear CM's, the (optimal) delegation problem is solvable in polynomial time. To obtain our results, it is convenient to work with a model equivalent to a linear CM. The following can easily be verified (see, e.g., [12]):

Theorem 2.2 *Every linear CM can be simulated by a* 1*-way* $\log\ n$ *space-bounded TM, and converse is also true.*

In view of the theorem above, for convenience, we will state our results in terms of 1*-way* $\log\ n$ *space-bounded Turing Machines*[1], while the same results can also be obtained in terms of linear CM's.

Next, we define an e-service modeled as a 1-way $\log\ n$ space-bounded TM.

Definition 2.3 An *e-service* is a (possibly nondeterministic) 1-way $\log\ n$ space-bounded TM A where A has a one-way read-only input tape and a finite number, say k, of two-way read/write work tapes (i.e., A uses no more than $\log\ n$ cells on each work tape for any input of length n). More formally, the service A is defined as $(S, \Sigma, \Gamma, \delta, s^0, B, F, cost)$ where

- S is the finite set of states, s^0 in S is the start state, $F \subseteq S$ is the set of *accepting states* where an enactment can be terminated successfully,
- Γ is the finite set of tape symbols and B, a symbol in Γ, is the blank symbol,
- Σ, a subset of Γ not including B, is the set of input symbols (*activity alphabet*),
- δ is the next move function, a mapping from $S \times \Gamma^{k+1}$ to $2^{S \times \Gamma^k \times \{+1,0\} \times \{+1,-1,0\}^k}$ (k is the number of work tapes). For instance, a transition $(s', b', c', 0, +1, -1) \in \delta(s, a, b, c)$ in a service with 2 work tapes represents the move the move: if the machine is in the state s and the input and work tape heads are scanning symbols a, b, c respectively, then it can go to the state s', and the work tape heads can replace b, c with b', c'. In addition, while the input head stays stationary (0), the work tapes move right and left respectively $(+1, -1)$.
- $cost$ is the cost function where $cost(q, a, p)$ defines the cost of processing an activity a when A makes a transition from a state q to a state p (If the transition is not defined in δ, then the cost is infinite). ∎

A service A defined as above is viewed as an acceptor. A accepts a word $w = a_1 a_2 ... a_n$ if, when started in its start state with input $\#a_1 ... a_n\#$ (#'s are the end markers) and blank read/write work tapes, it eventually lands in a unique accepting state with all work tapes blank (From now on we ignore the end markers and assume that two end markers are added to the original input).

When a client submits a service request (sequence of activities) to a service instance, a new enactment is created

[1] A 1*-way* $\log\ n$ *space-bounded Turing Machine (TM)* is an FSM augmented with a finite number of two-way read-write worktapes which are $\log\ n$ space-bounded in the sense that on any input of length n, each worktape uses no more than $\log\ n$ space.

and it is terminated when the processing finishes. Since the termination of an enactment is correct only when it happens at one of the accepting states, an accepted word intuitively represents a service request that can be processed by A successfully. Then, the *(activity) language* of a service A, denoted as $L(A)$, is the set of service requests accepted by A.

Let $\mathbf{C}$ denote a service community and $\mathbf{A_1}, ..., \mathbf{A_r}$ be the registered services in $\mathbf{C}$ where $|\mathbf{C}| = r$. Informally, a *delegation of a word* $w = a_1 a_2 ... a_n$ over $\mathbf{C}$ is a mapping that specifies which service in the community should process each activity. Intuitively, it defines a subsequence for each service and each service processing a non-empty subsequence should end up in an accepting state.

Now, we can formalize the notion of an optimal delegation as follows:

Definition 2.4 For a word w, a *predelegation* over $\mathbf{C}$ is a one-to-one function $d : [1..|w|] \rightarrow [1..|C|]$ and $image_j^d(w)$ is the subsequence of w obtained by concatenating the symbols assigned to the service j. Then, d is a *delegation* if it is a predelegation and $image_j^d(w)$ is either λ or in $L(A_j)$ for every service A_j in $\mathbf{C}$. Finally, a delegation d of w over $\mathbf{C}$ is an *optimal delegation* of w if there is no other delegation of w such that its cost is lower than the cost of d.

In the next section we give an analysis for a special case of this model where each service is an FSM (i.e., no counters). Then, we extend the results to the general model, namely FSMs with linear counters.

3 Delegation for FSM e-Services

In this section, we study a special case of the model introduced in the previous section such that each service is an FSM. Before we describe the linear time delegation algorithm, we first introduce the notion of product machine that is used in the explanation of the proposed algorithm.

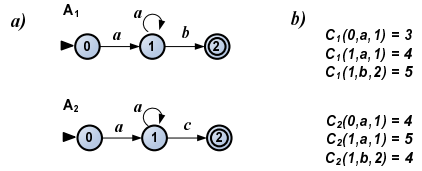

Figure 2. *a)* FSMs A_1, A_2 used in the product construction, and *b)* their cost functions, C_1 and C_2.

Example 3.1 Fig. 2.a shows two services, A_1 and A_2. Each has three states numbered as $0, 1$ and 2. Fig. 2.b shows the cost functions of A_1 and A_2. For example, $\mathcal{C}_2(0, a, 1) = 4$ means the cost of making an a transition from 0 to 1 costs 4 in A_2. The product machine of A_1 and A_2 is shown in Fig. 3. Each state of this machine represents a *configuration* of the system, i.e., it refers to a state for each service. For instance, the configuration 21 says that under this configuration, A_1 is in its state 2 and A_2 is in its state 1.

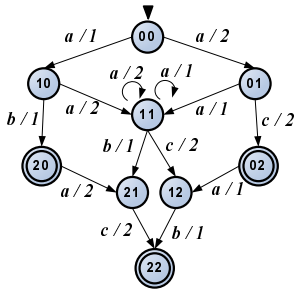

Figure 3. The product machine of the FSMs shown in Fig. 2

Transitions show how the system proceeds when an activity is processed. For example, the $a/1$ transition from 00 to 10 shows that if A_1 processes the activity a, then A_1 goes to its state 1, while A_2 stays at the state 0. In general, a transition x/i denotes the fact that x is delegated to service i. ∎

Next we formally define the product machine of a set of services.

Definition 3.2 Given a set $\{A_1, ..., A_r\}$ of services where $A_i = (S_i, \Sigma, \delta_i, s_i^0, F_i)$, the *product machine* PROD = $(S_\mathcal{P}, \Sigma^{in}, \Sigma^{out}, \delta_\mathcal{P}, s_\mathcal{P}^0, F_\mathcal{P})$ is a *Mealy* FSM[2] where

- Input and output alphabets: $\Sigma^{in} = \Sigma$, and $\Sigma^{out} = \{1, 2, ..., r\}$,
- States: $S_\mathcal{P} \subseteq (S_1 \times ... \times S_r)$,
- Starting state: $s_\mathcal{P}^0 = [s_1^0, ..., s_r^0]$,
- Accepting states: $F_\mathcal{P} = \{[q_1, ..., q_r] \mid \forall i \ (q_i \in F_i \vee q_i = s_i^0) \wedge \exists j \ q_j \in F_j$ where $i, j \in [1, r]\}$,
- The transition mapping $\delta : S_\mathcal{P} \times \Sigma^{in} \rightarrow S_\mathcal{P} \times \Sigma^{out}$ is defined as follows: Let $[q_1, ..., q_r]$ be a state in PROD. For each activity $a \in \Sigma$ and for each $i \in [1, r]$, if $\delta_i(q_i, a)$ is defined, then, there exists a transition $\delta([q_1, ..., q_r], a)$ in the product machine such that $\delta([q_1, ..., q_r], a) = \{ \ ([p_1, ..., p_r], i) \mid \ \ p_i = \delta_i(q_i, a) \wedge p_j = q_j$ if $j \neq i$ where $1 \leqslant j \leqslant r\}$.

  Note that the transition above represents the fact that a is assigned to the service i and therefore, the system moves to a new configuration where each machine stays in the same state except A_i. ∎

We note that the product machine is not only an acceptor, but an acceptor with outputs, i.e., a transducer. Also, in the description of accepting states, for the simplicity of the discussion, we assume that once a service processes an activity, it cannot return back to its initial state. In fact, every FSM can be converted to an equivalent FSM satisfying this property.

Note that since a configuration consists of the state information of each service, the number of states in the product

[2]Each transition produces an output.

of r services can be at most c^r where c is the number of states of the biggest service among r services.

We are now ready to move on to the algorithm we propose for the delegation problem and its analysis.

3.1 A Linear Time Delegation Algorithm

In this section, we informally describe the algorithm we propose that solves the delegation problem in linear time.

Let PROD denote the product of a set of services $\{A_1, ..., A_r\}$. Given a word $w = a_1...a_n$, we build a graph that simulates PROD on w and adds costs to the nodes to indicate the cost of every path. The goal is to find the shortest path in this graph.

More precisely, the graph is built by unfolding the PROD on the input string $a_1a_2...a_n$ as follows. Starting from the initial state of PROD, we build the computation tree, in which at level i, we will have all states of PROD that are reachable from the initial state on the prefix $a_1...a_i$ and we keep track of the total cost of each path from the initial state to the level i. For instance, Fig. 4.a shows the computation tree generated for the word $w = aaabc$ using the services in 2. This tree shows all possible paths that can be taken with $aaabc$ in the product machine shown in Fig. 3. The underlined numbers next to each node shows the cost of the path from the root to that node. For instance, the 00-10-11 path costs 7.

It is important to realize that if the structure is built as described, the number of nodes can be exponential in the length of w, because the tree shows all possible paths. Instead of building structure as a tree, we construct it as a DAG (directed acyclic graph) by merging the identical states at the same level. For example, in Fig. 4, when the second level of the tree is constructed, two copies of the state 11 appear. In this case, we merge them. Since we look for the shortest path, when we merge the states, we only keep the path which is the shortest one from the root. For instance, when the 11's in the second level are merged, there are two paths, 00-10-11 and 00-01-11. The path 00-10-11 is kept and the transition from 01 to 11 is ignored, which means we ignore the path 00-01-11. Similarly, in the next level, two copies of the state 11 appear and when they are merged, the path with $a/1$ transition is kept and the path with $a/2$ is ignored by removing the transition $a/2$ since the first path costs 11 while the second costs 12. The final DAG is shown in Fig. 4.b.

At each level of the DAG construction, we merge the identical states; therefore, there are at most c^r states at each level (where c = maximum number of states in the A_i's) which is the maximum number of states in PROD as explained in the previous section. Merging the states means merging some paths. When a merge happens, only the shortest path among those paths is kept and the others are ignored. Therefore, in each level, the number of paths arriving to a node is 1 (the shortest one) and because of that, the number of paths in the final DAG equals to the number of states in the final level which is at most c^r. Since the length

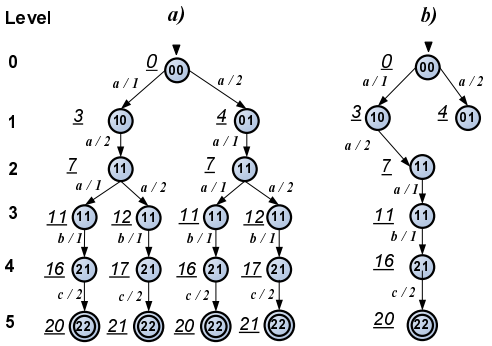

Figure 4. *a)* The full computation tree for $w = aaabc$ generated by unfolding the product machine in Fig. 3, and *b)* the computation graph considering only the shortest paths.

of the string w is n, there are c^r paths of length n. Therefore, the construction of the DAG takes $O(nc^r)$ time. When the final level is reached, by backtracking from the leaf level to the root level, the shortest path with the delegations can be extracted which takes $O(n)$ since there are n levels. As a result, the algorithm[3] takes $O(n)$ time since c and r are constants independent of w (since the set of services is fixed). Therefore, an optimal delegation can be computed in time linear in the length n of w.

It can be shown that the algorithm above can be implemented on a deterministic linear-time two-way Turing Machine (TM) transducer T (i.e., T has a two-way read-only input tape with endmarkers to contain the input string $a_1a_2...a_n$ to be processed, read-write work tapes, and a one-way write-only output tape to write the assignments). The TM essentially makes two passes on the input. On the first pass, it carries out the computation of the costs of the paths. After it has found a shortest path, the TM makes another pass on the input to output the delegation for each symbol in the input string.

Theorem 3.3 *An optimal delegation of a word w of length n over a community of services of FSMs can be computed in $O(n)$ time.*

One natural question regarding the improvement of the above algorithm is to whether there is an algorithm of time complexity $O(p(c, r) \times n)$ where p is a polynomial, c is $\max\{|A_i|\}$ and r is the number of services. Such an algorithm would be preferable. The following result, however, shows that this is unlikely since such an algorithm would imply $P = NP$.

Theorem 3.4 *If there is an algorithm of time complexity $O(p(c, r)\times n)$ for the optimal delegation problem described above, then $P = NP$.*

Proof: Suppose such an algorithm A exists. Consider the following problem: Given a finite collection of $r+1$ strings

[3]This algorithm has an exponential constant factor. Thm 3.4 proves that this factor cannot be reduced to a polynomial

x, y_1, ..., y_r (over a fixed alphabet of length at least 3), determine if x can be written as a shuffle of strings y_1, ..., y_r. [23] shows that this problem is NP-complete. We can now use algorithm A to solve this problem as follows: Let A_i be a deterministic FSM (DFSM) that accepts the string y_i for all i and assign cost 0 for all the edges for all DFSMs. Apply algorithm A and output "yes" if and only if the optimal cost is 0. Note that $|A_i| = |y_i|$. This gives an algorithm of polynomial time complexity for the shuffle problem from which it follows that $P = NP$. ∎

4 Delegation for Linear Counter e-Services

In this section, we generalize Theorem 3.3 to FSMs with *storage*. More specifically, each service is modeled as an FSM augmented with linear counters (i.e., an FSM having a finite number of counters, each of which can have value at most linear in the length of the input). We show that the (optimal) delegation problem is solvable in polynomial time.

Recall that for convenience we obtain the results using $\log\ n$ space bounded TMs (see section 2).

Definition 4.1 Let A be $\log\ n$ space bounded TM with k work tapes. A (partial) *configuration* of A is a tuple $ID = (q, y_1, j_1, ..., y_k, j_k)$, where q is the current state, y_i is the content of the worktape i, and j_i is the position of the read-write head within the worktape i. ∎

Let $\{A_1, ..., A_r\}$ be a community of services, where each A_i is a $\log\ n$ space bounded TM (recall that the input is 1-way). The product machine PROD is defined in the same was as for FSMs. Rather than define it formally, we just give a brief informal description. Like in the FSMs case, PROD simulates the computations of the A_i's faithfully by keeping track of the state changes and the changes in the worktapes (PROD, thus, will have as many worktapes as the A_i's). When a new input symbol has to be processed, PROD nondeterministically simulates the move of exactly one of the A_i's that requests (i.e., reads a new symbol). Note that the A_i's are not operating in real-time (i.e., their input heads do not move right at every step) and, hence, their input heads are not synchronized. So, e.g., A_1 might be requesting to read a symbol at time t_1 while A_2 might request to read the (same) symbol at a later time t_2. PROD may then delegate the symbol to A_1 at time t_1 (hence simulating the move of A_1 on the input symbol) or guess that A_2 will read at a later time and delegates the symbol to A_2 at time t_2.

Next, we analyze the space complexity and time complexity of computing an optimal delegation.

4.1 Space Complexity

Let's first define a recursive algorithm which is used later for our delegation algorithm.

Let A be $\log\ n$ space-bounded TM. Given two configurations ID_1, ID_2 of A and an input $w = a_1a_2...a_n$, Algorithm 1 checks whether ID_2 is reachable from ID_1 with the word $a_i...a_j$.

Algorithm 1 $CHECK(i, j, ID_1, ID_2)$

```
1: if i = j
2:    if ID_1 can reach ID_2 in n^k steps by processing symbol a_i
3:       return true
4:    else
5:       return false
6: else
7:    for each configuration ID_3 do
8:       if CHECK(i, (i+j)/2, ID_1, ID_3) and
         CHECK((i+j)/2+1, j, ID_3, ID_2)
9:          return true
10:      return false
```

The check in line 2 of Algorithm 1 can be done by calling Algorithm 2 due to Savitch[12]. Algorithm 2 checks if the machine can change its configuration from ID_1 to ID_2 in m steps while processing an input symbol. Since the size of a configuration in this case is $\log\ n$, the number of all possible configurations is at most at most n^k for some fixed k. Therefore, for any ID_1 and ID_2, if ID_1 can reach ID_2 by processing a single input symbol, say a, then this will happens in no more than n^k steps. In fact, for convenience, we can assume, that the number of steps is always exactly n^k (by repeating ID_2). Therefore, Algorithm 1 can check the reachability from ID_1 to ID_2 in n^k steps ($m = n^k$).

Algorithm 2 $TEST(ID_1, ID_2, m)$

```
1: if m = 1
2:    if (ID_1 = ID_2) or ID_2 is reachable from ID_1 in one
      step
3:       return true
4:    else
5:       return false
6: else
7:    for each ID_3 do
8:       if (TEST(ID_1, ID_3, ⌊m/2⌋)) and
         (TEST(ID_3, ID_2, ⌈m/2⌉))
9:          return true
10:   return false
```

Lemma 4.2 $CHECK(i, j, ID_1, ID_2)$ takes $O(\log^2 n)$ space where ID_1,ID_2 are two configurations of a $\log\ n$ space-bounded TM and $j - i \leqslant n$.

Now we are ready to define our delegation algorithm. Let PROD be the product machine of a set of services. Let ID_0 be the initial configuration of PROD. Without loss of generality, assume that PROD has only one (i.e., unique) accepting ID, say ID_f. Algorithm 3 describes how to determine the delegation of symbols to the A_i's of a string $w = a_1a_2...a_n$.

This algorithm can be implemented on a $\log^2 n$ TM space transducer. For each input symbol, Algorithm 1 is executed which requires $\log^2 n$ space by Lemma 4.2. Each substring position and configurations require $\log\ n$ space. As a result, the total space complexity of Algorithm 3 is $\log^2 n$.

Algorithm 3 $DELEGATE(a_1, a_2, ..., a_n)$

1: **if** $CHECK(1, n, ID_0, ID_f)$ is false
2: output(*error*);
3: **else**
4: $current = ID_0$;
5: **for each** i from 1 to n **do**
6: $N = \{ID \mid CHECK(i, i, current, ID)\}$; /* set of next configurations from $current$ configuration with a_i */
7: Find a configuration, say ID_p, in N such that $CHECK(i+1, n, ID_p, ID_f)$ is true;
8: Output index of machine, say j, that corresponds to the transition from $current$ configuration to ID_p (thus a_i is assigned to A_j).
9: $current = ID_p$;

Theorem 4.3 For a set of services modeled as $\log\ n$ space-bounded TMs where n is the input length, the delegation of a word w, can be done in $O(\log^2 n)$ space.

4.2 Time Complexity

Let PROD denote the product of a set of services and ID_0 be the initial configuration of PROD. For every possible configuration ID, PROD_{ID} represents the same machine with PROD except the initial configuration of PROD_{ID} is ID instead of ID_0. Algorithm 4 computes a delegation in polynomial time.

Algorithm 4 $DELEGATE(a_1, a_2, ..., a_n)$

1: **if** $a_1 a_2 ... a_n$ is not accepted by $PROD_{ID_0}$
2: output *error*;
3: **else**
4: $current = ID_0$
5: **for each** i from 1 to n **do**
6: $N = \{ID \mid CHECK(i, i, current, ID)\}$; /* set of next configurations from $current$ configuration with a_i */
7: Find a configuration, say ID_p, in N such that $a_{i+1} ... a_n$ is accepted by $PROD_{ID_p}$
8: Output the index of machine, say j, that corresponds to the transition(s) from $current$ configuration to ID_p (thus a_i is assigned to A_j).
9: $current = ID_p$;

Since a $\log\ n$ space-bounded nondeterministic TM can be simulated by a polynomial time-bounded deterministic TM [12], Line 1 can be implemented in polynomial time doing a breadth first search and marking reached configurations. Since there is a polynomial number of configurations, at each step, marking takes polynomial time. The process continues n times, therefore, the total complexity of Line 1 is polynomial. Line 6 takes a polynomial number of steps, because, there is a polynomial number of configurations and for each configuration, Algorithm 1 calls Algorithm 2 which also takes polynomial time. Line 7 is similar to Line 1. It follows that the entire algorithm can be implemented on a deterministic polynomial time-bounded two-way TM transducer. Therefore, we have the following:

Theorem 4.4 For a set of services of $\log\ n$ space-bounded TMs where n is the input length, a delegation of an input word w, can be done in polynomial time in the length of w.

Clearly, the technique above can be changed to take processing costs into consideration with the DAG idea used in the FSM case. In FSM case, the nodes of the DAG are the states of FSMs. In this case, they are configurations of space-bounded TMs. Since the number of configurations in a $\log\ n$ space-bounded TMs is polynomial, there is a polynomial number of paths in the DAG. Therefore, we get the following result:

Corollary 4.5 For a set of services of $\log\ n$ space-bounded TMs where n is the input length, an optimal delegation of an input word w can be computed in polynomial time in the length of w.

5 Related Work

This paper is most relevant to the work [4, 5, 11, 8, 3, 8]. [4] defines an e-service framework and study the problem of automated composition synthesis. One input is a set of descriptions of e-services, each given as an automaton. The second input is a desired global behavior, also specified as an automaton, which describes the possible sequences of activities. The output is a subset of the atomic web services, and a delegator that will coordinate the activities of those services, through a form of delegation. Finding a delegator, if it exists, can be done in EXPTIME. [8] studies generalizations to automata with unbounded storage where decidability and undecidability of the composability problem are shown. [5] extends the framework to allow interactions among existing services, which provides more flexibility to the services to achieve the desired service behavior. In [11], the notion of delegator was extended to have "look ahead", i.e., the delegation of an activity is determined by looking at the future activities. In all three approaches, a desired service is specified and a composite service is created through composition of existing services. The construction happens before run time. In this study, instead of a desired service, an instance of an execution is given and we are only interested in performing the given instance at run time. In addition, the earlier work in [4, 5, 11] were based only on standard finite state machines and did not have cost functions associated with the activities.

The idea of service communities is similar to the ones studied in [4] and [3]. Our notion of service communities is closer to the one of [4]. An important difference between [3] and [4] is that in [3], service communities define services they would like to have, and service providers register their services if they think they can provide the desired service.

In [4], service providers export their services with respect to a community alphabet and the community figures out what services it can provide using the registered services.

Service composition is also closely related to planning [16], where existing tasks are put together for a given goal. An approach to automated composition [18] has been developed for the OWL-S model [7]. The basic question in that work is whether a given collection of atomic services can be combined, using the OWL-S constructors, to form a composite service that accomplishes a stated goal. The approach taken is to encode the underlying situation calculus world view, the desired goal, the individual services (or more specifically, their pre-conditions and effects), and the OWL-S constructors into a Petri net model. This reduces the problem of composability to the problem of reachability in the Petri net. The main difference in our approach is that we use finite state machines to represent services and the desired goal is represented as a sequence of activities.

The problem was also considered in the context of workflows. In [22], the global dependencies are given as a tree, with "optional" and "choices" on some dependencies, resembling the event algebra [21]. An algorithm was given to map to a Petri net that generates the root of the tree without violating the dependencies. In a simpler model, [15] starts from a pair of pre- and post-conditions and assembles the workflow by selecting tasks from a library.

Another approach on the automated composition problem is considered in [6, 10] where the desired global behavior described as a *conversation* (a family of permitted message sequences) specified as a finite state automaton. These studies use message-based models whereas we focus on an activity-based model.

6 Conclusion

In conclusion, this paper focused on the optimal ad hoc delegation problem in dynamic e-service communities where services are modeled as FSMs augmented with linear counters. We formally analyzed the problem and give complexity bounds. In future, we plan to build software implementations inspired from the techniques presented in this paper.

References

[1] *Proceedings of the IEEE International Conference on Services Computing (SCC'04), September 15-18, 2004, Shanghai, China*. IEEE Computer Society, 2004.

[2] *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*. IEEE Computer Society, 2004.

[3] B. Benatallah, M. Dumas, Q. Sheng, and A. Ngu. Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, 2002.

[4] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. 1st Int. Conf. on Service Oriented Computing (ICSOC)*, volume 2910 of *LNCS*, pages 43–58, 2003.

[5] D. Berardi, G. D. Giacomo, M. Lenzerini, M. Mecella, and D. Calvanese. Synthesis of underspecified composite e-services based on automated reasoning. In *Proc. Int. Conf. on Service Oriented Computing (ICSOC)*, 2004.

[6] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. Int. World Wide Web Conf. (WWW)*, May 2003.

[7] O. S. Coalition. OWL-S: Semantic markup for web services, November 2003.

[8] Z. Dang, O. Ibarra, and J. Su. Composability of infinite-state activity automata. In *Proc. 15th Annual Int. Symp. on Algorithms and Computation, Hong Kong*, 2004.

[9] S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors. *Proceedings of the 13th international conference on World Wide Web, (WWW), New York, NY, USA, May 17-20, 2004*. ACM, 2004.

[10] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proc. Int. Conf. on Implementation and Application of Automata (CIAA)*, 2003.

[11] C. E. Gerede, R. Hull, O. H. Ibarra, and J. Su. Automated composition of e-services: Lookaheads. In *Proc. of 2nd Int. Conf. on Service Oriented Computing (ICSOC)*, pages 252 – 262, New York, NY, November 2004.

[12] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[13] R. Hull and J. Su. Tools for design of composite web services. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 958–961, 2004.

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, (1):41–50, Jan. 2003.

[15] S. Lu. *Semantic Correctness of Transactions and Workflows*. PhD thesis, SUNY at Stony Brook, 2002.

[16] S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25*, pages 482–496, 2002.

[17] M. Mecella and G. D. Giacomo. Service composition: Technologies, methods and tools for synthesis and orchestration of composite services and processes. In *Proc. Int. Conf. on Service Oriented Computing (ICSOC)*, 2004.

[18] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proc. Int. World Wide Web Conf. (WWW)*, 2002.

[19] M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang, editors. *Service-Oriented Computing - ICSOC 2003, First International Conference, Trento, Italy, December 15-18, 2003, Proceedings*, volume 2910 of *Lecture Notes in Computer Science*. Springer, 2003.

[20] R. Ramakrishnan et al. Science of design for information systems. *ACM SIGMOD Record*, 33(1):133–137, Mar. 2004.

[21] M. Singh. Semantical considerations on workflows: An algebra for intertask dependencies. In *Proc. Workshop on Database Programming Languages (DBPL)*, 1995.

[22] W. M. P. van der Aalst. On the automatic generation of workflow processes based on product structures. *Computer in Industry*, 39(2):97–111, 1999.

[23] M. K. Warmuth and D. Haussler. On the complexity of iterated shuffle. *J. Comput. Syst. Sci.*, 28(3):345–358, 1984.