Theoretical Computer Science 409 (2008) 417–431

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Minimum-cost delegation in service composition[☆,☆☆]

Cagdas E. Gerede [a,*], Oscar H. Ibarra [a], Bala Ravikumar [b], Jianwen Su [a]

[a] Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

[b] Department of Computer Science, Sonoma State University, Rohnert Park, CA 94928, USA

ARTICLE INFO

Article history:
Received 30 August 2006
Received in revised form 31 March 2008
Accepted 25 August 2008
Communicated by M. Ito

Keywords:
Finite state automata
Service modeling
Automated service composition

ABSTRACT

The paradigm of automated service composition through the integration of existing services promises a fast and efficient development of new services in cooperative service (e.g., business) environments. Although the "why" part of this paradigm is well understood, many key pieces are missing to utilize the available opportunities. Recently "service communities" where service providers with similar interests can register their services are proposed toward realizing this goal. In these communities, requests for services posed by users can be processed by delegating them to existing services, and orchestrating their executions. We use a service framework similar to the "Roman" model departing from it particularly assuming service requirements are specified in a sequence form. We also extend the framework to integrate activity processing costs into the delegation computation and to have services with bounded storage as opposed to finite storage. We investigate the problem of efficient processing of service requests in service communities and develop polynomial time delegation techniques guaranteeing optimality.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

The framework of web services paints a promising picture for future software and applications development. Much of the expected benefits comes from a systematic approach of sharing program executions, a.k.a. services, as well as data. This has been clearly demonstrated by applications built on software development tools and platforms including Microsoft's .NET, Sun's J2EE, IBM's WebSphere, BEA's Web Logic, etc. 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 [25]. 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 [32]. A major challenge thus lies 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 *services*.

Service composition has its similarities with information system integration, workflow system design, and distributed computing [23,28]. 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 [23,28]. Towards this goal, *service communities* have been proposed [4,2] where service providers with similar interests can register their services for the community use. When the user asks for the execution of some activities, if the community

☆ The preliminary results from this paper were presented at the International Conference on Services Computing (SCC '05) [C.E. Gerede, O.H. Ibarra, B. Ravikumar, J. Su, Online and minimum-cost ad hoc delegation in e-service composition, in: Proc. IEEE Int. Conf. on Services Computing, SCC, 2005].

☆☆ Work supported in part by NSF grants IIS-0101134 (Gerede, Ibarra, and Su), CCF-0430945 and CCF-0524136 (Ibarra), and IIS 0415195 (Gerede and Su).

* Corresponding author. Tel.: +1 805 637 3212.
E-mail addresses: cagdas.gerede@gmail.com (C.E. Gerede), ibarra@cs.ucsb.edu (O.H. Ibarra), ravi.kumar@sonoma.edu (B. Ravikumar), su@cs.ucsb.edu (J. Su).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.08.040

does not have a capable service directly, these activities may be delegated to registered services so that the user needs are satisfied.

Service composition, generally, consists of two main steps [4]. The *first* step, 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. 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.

Automated composition synthesis problem was studied in [4,6,17]. Given a set of descriptions of existing services (e.g., from a "UDDI + +" repository) and a desired service, the problem is to construct a "delegator" that will coordinate the activities of those services to achieve the desired service. All three studies use a model, often referred to as the "Roman" model, where services are represented as activity-based finite state machines (FSMs). Extending these studies, in this paper we model services as FSMs augmented with linear counters and we integrate activity processing costs into the model. We then investigate the problem of computing an optimal delegation for a given sequence of service requests.

We depart from the original Roman model in the following ways. First of all, service requirements can be described in many different forms. One form can be a Turing-machine. Another can be a finite transition system like the case in the original Roman model. The forms we study here are sequences. A client requests a sequence of activities, and then the sequence is delegated to available services. Second, we provide a bounded storage to each service and illustrate the cases where this type of storage is required. Third, we integrate the costs of activity processing into the model to capture the cases where the processing cost of an activity varies from one service to another.

In this paper, we study the following problem: given a community of services and a sequence of activities, how to compute an optimal delegation of the sequence, i.e., the cheapest way to process the sequence using the community services in a collaborative manner. We make the following main contributions:

- We show that for finite state services, there is a linear time algorithm computing an optimal delegation, and that for services with linear bounded storage, the optimum delegation problem can be solved in polynomial time and space or in $O(\log^2 n)$ space and slightly super-polynomial time ($O(n^{\log n})$) in terms of n where n is the length of the service request. Note that in the complexity analysis, we consider the service request the only input to the problem. In other words, the size and the number of services are considered constant.
- When we take the size and the number of services into account, our algorithms for both cases have the exponential constant factor c^k where c is the size of services, and k is the number of services. The algorithm proposed in [4] is also linear in the size of services and exponential in the number of services. We show that it is very unlikely to have an algorithm with polynomial time complexity in both the size and the number of services.

The remainder of the paper is organized as follows. In Section 2, we describe our service model and delegation problem with an example. In Section 3, we formalize the model and the problem. We provide a linear time delegation algorithm for a restricted version of the model in Section 4. In Section 5, we study the complexity of delegation for the general model. Related work and conclusions are provided in Sections 6 and 7, respectively.

2. A model for services

In this section, we first summarize the framework we used as a starting point for our model used in this paper. We then explain the extensions introduced in our model. The explanations are mostly informal. The formal discussion will be presented in Section 3.

We base our model on the service framework presented in [4] (often referred to as the Roman model). This initial effort provides a nice formal setting for a precise characterization of automatic composition of services. In this model, a *service* is a software artifact interacting with its clients (humans or other 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. A *service instance* refers to one occurrence of a 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 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 a response. On the basis of the response, she determines her next activity request. When the client doesn't have any more requests, she may explicitly terminate the enactment.

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

An *service community* consists of an *activity alphabet* Σ and a set of services of similar interests. A 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 [2]).

The framework described so far is general and does not refer to any specific forms of service schemas. As formalized in the next section, the study in this paper focuses on the services whose external service schemas are represented as *nondeterministic finite state machines augmented with counters*. In our study, we only focus on external schemas; therefore, from now on we use the word "service" to actually mean an external service schema.

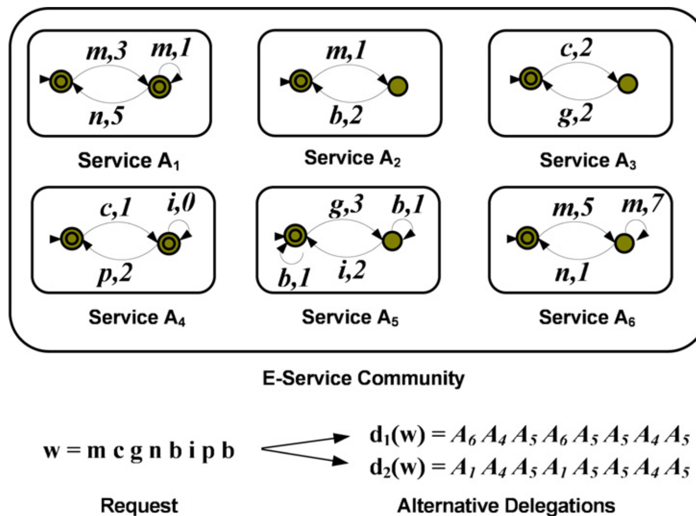

Fig. 1. A community of services.

Being able to use nondeterminism provides the specification designer more flexibility in describing a service. For example, consider a music service where you search songs, listen to them, and finally buy the one you listen to the last. The set of transitions of this service can be described as $\{(q_0, \text{search}, q_0), (q_0, \text{listen}, q_0), (q_0, \text{listen}, q_1), (q_1, \text{buy}, q_2)\}$ using a nondeterministic FSM, or can be described as $\{(q_0, \textit{search}, q_0), (q_0, \textit{listen}, q_1), (q_1, \textit{search}, q_0), (q_1, \textit{listen}, q_1), (q_1, \textit{buy}, q_2)\}$ using a deterministic FSM. This interpretation of nondeterminism is similar to "angelic" nondeterminism [5]. We would like to point out that the nondeterminism can also be interpreted as the result of unknowns in the environment. For example, consider a shopping service with an operation validateCreditCard. When the client requests the validation of her credit card, depending on the outcome of the validation, the service can go to a success or failure state, from which different options would be available for the client. This can be described with two transitions labeled with the same operation validateCreditCard going nondeterministically in two different states. This interpretation of nondeterminism is also called "devilish" nondeterminism [5,19]. In this study, we adopted the former interpretation.

The original framework also does not describe activity processing costs. Most of the practical cases, however, require the modeling of costs because the processing costs may vary from one service to another. These costs can be related, for instance, to the amount of money charged to a client, or the communication time necessary for the processing. In our study we extend service specifications with cost functions to capture the cost information explicitly.

The following example illustrates the described notions.

Example 1. Fig. 1 illustrates a community of genomics services. The community alphabet consists of the following activities: transcription factor binding search (**b**), cluster and principal components analysis (**c**), GenBank sequence retrieval (**g**), promoter identification (**i**), microarray analysis (**m**), NCBI BLAST search (**n**), and promoter model generator (**p**). The community has 6 registered services each with different functionalities and processing costs. For instance, the service A_4 can perform **c**, **i** and **p** with costs 1, 0 and 2, respectively. Note that costs are context dependent meaning a service can process the same activity with different costs depending on the current state of the service, for instance, the activity m in the service A_1.

Each service (schema) is represented by an FSM as illustrated in Fig. 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 experiment 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. We represent this sequence of activities as a word $w = mcgnbipb$. Obviously, none of the 6 services can process w by itself. The services, on the other hand, can be coordinated so that they collaboratively process w.

There are two possible ways to accomplish this. The processing of the activities m, c, g, n, b, i, p, b can be delegated to $A_6, A_4, A_5, A_6, A_5, A_5, A_4, A_5$, respectively (shown as $d_1(w)$ in Fig. 1). In other words, A_6 processes the subsequence *mn*, A_4 processes the subsequence *cp*, etc. An alternative way is to perform the same delegation except the subsequence *mn* is delegated to A_1 (shown as $d_2(w)$ in Fig. 1).

The cost of a delegation of a sequence is the sum of the delegation costs of individual activities. Therefore, the cost of d_1 is 16 $(=5 + 1 + 3 + 1 + 1 + 2 + 2 + 1)$, and the cost of d_2 is 18 $(= 3 + 1 + 3 + 5 + 1 + 2 + 2 + 1)$. We can say that d_1 is the optimal (cheapest) delegation of w to the service community in Fig. 1. □

In some cases, not all delegations are desirable. System designers very often would like to constrain the delegations the system is allowed to perform. These constraints can be of different types. They can be about each service in isolation

such as the number of "payment" activities processed by a shopping service should be equivalent to the number of "order" activities. This type of constraints is especially important to specify service level agreements, where a service requires a specific usage. They can also be about the relationships of a delegator over multiple services such as the number of "order" activities delegated to the shopping service 1 must be more than the number of "order" activities delegated to the shopping service 2, because we have an agreement with service 1 to direct the traffic towards that as much as we can. They can also be about systems generated by single services as a whole. In fact, this type of constraints is widely used in the choreography literature [7,16].

In this study, we consider linear constraints, a subset of the first type of constraints. For example, in the service A_4 in Example 1, we would like to restrict the delegations based on the usage history so that the service A_4 can perform an *i* only if the requester has delegated and paid for enough number of *c*'s and *p*'s so far. In other words, the number of *i*'s performed by A_4 can be at most 4 times the total number of *c*'s and *p*'s A_4 has performed so far. To enable the specification of such kind of constraints, we augment FSMs with linear bounded counters. More details are provided in the next section.

Note that in terms of computing an optimal delegation, a greedy approach such as delegating each activity in a given sequence to the service with the cheapest processing cost does not work. In the optimal delegation in Example 1, *m* is delegated to A_6 even though the cost at A_1 is lower. The delegation of an activity, in fact, may require more complex analysis of the whole sequence. For instance, if the number of *m*'s in w were more than 1, then the first *m* would be assigned to A_1 in the optimal delegation.

We would like to clarify an important departure from the original Roman model of [4,6,5]. In the original Roman model, the delegation is based on a target transition system. Given a target service as a transition system, the goal is to construct a delegator which outputs a delegation for every desired sequence of activities in the target service. For instance, for a target service with the desired activity language $\{mn, cg\}$, we can construct a delegator, because *mn* can be processed by Service A_1 and *cg* can be processed by Service A_3. An important characteristic of the original model is the interactivity of services and clients. Whenever the service performs an activity, it provides the next possible set of activities and the client can request an activity from this set. Because of this interactivity, a delegator built on top of existing services also has to work interactively. In other words, it has to decide the delegation of each activity before the next activity. For example, a target service with the activity language $\{mn, mb\}$ cannot be realized, because the delegation of *m* cannot be determined (either to Service A_1 or to Service A_2) before knowing whether the next activity is *n* or *b* (this will be decided at run time by the client).

A natural question is whether we gain anything if we sacrifice some degree of interactivity. In fact, in [17], the authors studied a different class of delegators, namely lookahead delegators, where the delegator delays the decision of activities a fixed amount of steps. For instance, for the target service with the activity language $\{mn, mb\}$, there is a 1-lookahead delegator which delays the delegation of *m* until it knows whether the next activity is *n* or *b*. Practically, this means that when the client requests the execution of m, the delegator asks whether the client will pick n or b in the next round. It is shown that the more delays we allow the delegator to have, the more target services we can realize [17]. In this paper, we study the case when a delegator can delay a delegation an unbounded number of steps (i.e., the delay amount depends on the input length). Practically, the client gives the complete sequence of activities she requests, and the delegator delegates this sequence from beginning to the end without any other interactions with the client. The batch-style specification of users' requests are also adopted in many real systems. For instance, in high performance computing systems such as MapReduce [12] system and Hadoop system [21], which are used by Google and Yahoo, respectively, for their large scale data analysis jobs emerging in the areas such as web search and online advertisement, users specify the desired computations in-priori. Then, these systems achieve these computations by delegating them among available services.

3. Formalization

In this section, we formally define the notion of a service, and the optimal delegation problem. We assume some familiarity with formal languages and finite state machines.

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). We use λ to denote the *empty word*, i.e., 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) (see [13] for counter machines). 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 stay unchanged. The counters can be tested for zero. Note that these counters are restricted in the sense that there is a constant *c* such that on any input of length *n*, the value stored in each counter during the computation must be at most *cn*; thus, these 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*.)

A counter is formally represented by a push-down stack where the stack can only be checked if it has a top element. More precisely, the stack has only two symbols "*B*" and "0" where *B* can only appear at the bottom of the stack and it is never erased. Thus, the number of 0's on top of *B* represents a number, the content of the counter. Note that there is no direct way to check the value of a counter other than testing whether it is zero or not, i.e., whether the stack is empty or not.

Example 2. For the service A_4 in Example 1, we specify a constraint that the number of *i*'s performed should be at most 4 times the number of *c*'s and *p*'s performed. To specify this constraint, the FSM of A_4 can be augmented with 5 linear bounded

counters. Given a string, each of the 4 counters counts the total number of c's and p's performed (all 4 counters contain the same count). The 5-th counter counts the number of i's. At the end of the computation, the 5-th counter is decremented by the other 4 counters. If the 5-th counter is zero, then the number of i's is at most 4 times the number of c's and p's; therefore, the string is accepted. Any string accepted by A_4, therefore, any delegation assigned to A_4, satisfies the constraint. □

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 1-way $\log n$ space-bounded Turing machines [22] that are equivalent to linear CM's. Roughly, a (possibly nondeterministic) *1-way $\log n$ space-bounded Turing machine (TM) A* has a one-way read-only input tape and k two-way read/write work tapes for some $k \geqslant 0$. The TM A uses no more than $\log n$ cells on each work tape for any input of length n. The following can easily be verified (see, e.g., [22]).

Theorem 1. *Every linear CM can be simulated by a 1-way $\log n$ space-bounded Turing machine, and conversely.*

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

We model each service as a 1-way $\log n$ space-bounded TM with associated cost on transitions.

Definition 1. An *service A* is a tuple $(S, \Sigma, \Gamma, \delta, s_0, B, F, C)$ where

- S is the finite set of states, $s_0 \in S$ is the start state, and $F \subseteq S$ is the set of *accepting states* (where an enactment can be terminated successfully),
- Γ is a finite set of tape symbols including the blank symbol B,
- $\Sigma \subseteq \Gamma - \{B\}$ is the set of input symbols (*activity alphabet*),
- $\delta \subseteq S \times \Sigma \times \Gamma^k \times S \times \Gamma^k \times \{\mathsf{R}, -\} \times \{\mathsf{R}, \mathsf{L}, -\}^k$ is a set of *transitions* (k being the number of work tapes).
  Note that each transition in δ is a tuple $(s, a, b_1, \ldots, b_k, s', b'_1, \ldots, b'_k, m, n_1, \ldots, n_k)$ where $s, s' \in S$, $a \in \Sigma$, $b_i, b'_i \in \Gamma$, $m \in \{\mathsf{R}, -\}$, $n_j \in \{\mathsf{R}, \mathsf{L}, -\}$. The *semantics* of the transition is defined naturally; for instance, the transition $(s, a, b, c, s', b', c', -, \mathsf{R}, \mathsf{L})$ of a service with 2 work tapes has the following effect: if the service is in the state s and the input and work tape heads are scanning symbols a, b, c (respectively), then the service goes to the state s', and the work tape heads replaces b, c with b', c'. In addition, the input head stays stationary ($-$), while the work tape heads move right and left (respectively) (R, L).
- C is a cost function that maps each transition t in δ to the cost $C(t)$ of performing the transition.

A service A defined as above can be viewed as an acceptor. A accepts a word $w = a_1 a_2 \cdots a_n$ if, when started in its start state with input $\#a_1 \cdots a_n\#$ (#'s are the end markers) and blank read/write work tapes, it eventually lands in an accepting state with all work tapes blank. When it is convenient, we may ignore the end markers and assume that two end markers are added to the original input. When the number of work tapes is 0, we call such a service an *Finite State Machine service*, or shortly, an *FSM service*.

When a client submits a request (a sequence of activities) to a service instance, a new enactment is created 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 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 requests accepted by A.

Definition 2. Let A be a service (or a $\log n$ space bounded TM with k work tapes). A (partial) *configuration* of A is a tuple $\text{ID} = (q, y_1, j_1, \ldots, y_k, j_k)$, 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.

The configuration notion defined is partial because it includes neither the position information of the head of the input tape nor the input tape's content. Since the size of each worktape content is at most $\log n$ space, the total number of such configurations is $|S| \times ((|\Gamma|^{\log n}) \times \log n)^k$ which is bounded by n^m for some constant m. For an FSM service, the number of configurations is $|S|$.

A *service community* $\mathbf{C}$ *of size r* is a set $A_1, \ldots, A_r$ of services. Informally, a *delegation of a word* $w = a_1 a_2 \cdots a_n$ 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 nonempty subsequence should end up in an accepting state.

Now we can formalize the notion of a delegation as follows:

Definition 3. For each word w, an *online pre-delegation* of w over $\mathbf{C}$ is a function $d : \{1, 2, \ldots, |w|\} \to \mathbf{C}$. Let $image_A^d(w)$ be the subsequence of w obtained by concatenating the symbols assigned to the service A. Then, an online pre-delegation d is an *online delegation* if d is total on $\{1, 2, \ldots, |w|\}$, and $image_A^d(w)$ is either λ or in $L(A)$ for every service A in $\mathbf{C}$.

Given a delegation d, for a service A, $image_A^d(w)$, unless it is λ, corresponds to a sequence of transitions (or path) in the service A's state space. The *cost* of $image_A^d(w)$ is the sum of the cost of these transitions. If it is λ, then the cost is defined as 0. If A is nondeterministic, $image_A^d(w)$ may correspond to a set of sequences of transitions. In this case, the cost is defined as the cost of the sequence having the minimum cost. Finally, the *cost* of a delegation is defined as the sum of the costs of $image_A^d(w)$ for all $A \in \mathbf{C}$.

Definition 4. A delegation d of w over $\mathbf{C}$ is *optimal* if there are no other delegations of w over $\mathbf{C}$ with some cost less than the cost of d.

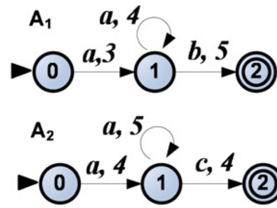

Fig. 2. Services A_1, A_2 used in the product machine construction.

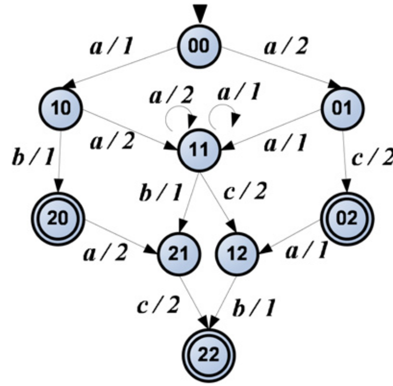

Fig. 3. The product machine of the FSMs shown in Fig. 2.

To make the presentation of our results on the general model clear, we first present our findings on a restricted version. In the restricted version, we remove the linear bounded storage from services. In essence, each service becomes an FSM service.

4. Delegation for FSM services

In this section, we study a restricted version of the model introduced in the previous section such that each service has no worktapes, therefore, has a finite number of partial configurations.

Before we describe our linear time delegation algorithm, we first introduce the notion of a *product machine* that is used in the explanation of the proposed algorithm.

Example 3. Fig. 2 shows two services, A_1 and A_2. Each has three states numbered as 0, 1, and 2. Each transition is also labeled with its cost. For example, the cost of A_1 making the transition $(0, a, 1)$ is 3. The product machine of A_1 and A_2 is shown in Fig. 3. Each (global) state of the product machine represents a "configuration" and it refers to a state for each service. For instance, the configuration 21 represents a global state where A_1 is in its state 2 and A_2 is in its state 1. 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 the service i, i.e., the service i makes an x transition. □

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

Definition 5. Given a set $\{A_1, \ldots, A_r\}$ of services where for each $1 \leqslant i \leqslant r$, $A_i = (S_i, \Sigma, \delta_i, s_i^0, F_i)$, the *product machine* PROD $= (S_P, \Sigma^{in}, \Sigma^{out}, \delta_P, s_P^0, F_P)$ is a *Mealy* FSM where

- Input and output alphabets: $\Sigma^{in} = \Sigma$, and $\Sigma^{out} = \{1, 2, \ldots, r\}$,
- States: $S_P \subseteq (S_1 \times \cdots \times S_r)$,
- Starting state: $s_P^0 = [s_1^0, \ldots, s_r^0]$,
- Accepting states[1]: $F_P = \{[q_1, \ldots, q_r] \mid \forall i (q_i \in F_i)\}$
- The transition mapping $\delta : S_P \times \Sigma^{in} \to S_P \times \Sigma^{out}$ is defined as follows: Let $[q_1, \ldots, q_r]$ 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 $\delta([q_1, \ldots, q_r], a)$ in the product machine such that $\delta([q_1, \ldots, q_r], a)$ includes $([p_1, \ldots, p_r], i)$ where $p_i = \delta_i(q_i, a)$, and $p_j = q_j$ if $j \neq i\}$.
  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 merely 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.

[1] When defining the accepting states, for the simplicity of the discussion, we assume that the initial states are accepting.

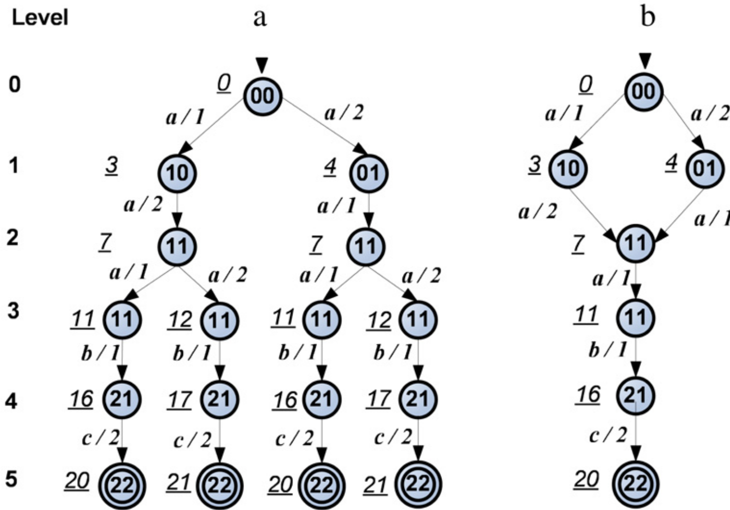

Fig. 4. (a) The full computation tree for $w = aaabc$ generated by unfolding the product machine in Fig. 3, and (b) the final reduced directed acyclic graph.

Note that since a configuration consists of the state information of each service, the number of states in the product of r services can be at most c^r where c is the number of states of the biggest service.

We are now ready to discuss the algorithm for the delegation problem and related analysis.

Let PROD denote the product of a set $\{A_1, \ldots, A_r\}$ of services. Given a word $w = a_1 \cdots a_n$, 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 $a_1 a_2 \cdots a_n$ as follows. Starting from the initial state of PROD, we can construct a tree, in which at level i, we will have all states of PROD that are reachable from the initial state on the prefix $a_1 \cdots a_i$ 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 Fig. 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 cost of the path 00-10-11 is 7.

It is important to note that if the structure is constructed as described, the number of nodes can be exponential in the length of the input word w, because the tree shows all possible paths. Instead of constructing the structure as a tree, we construct it as a directed acyclic graph by merging the identical states at the same level. For example, in Fig. 4(a), when the second level of the tree is constructed, two copies of the state 11 appear. In this case, we merge them (Fig. 4(b)). Since we look for the shortest path, when we merge the states, for each state, we only keep the paths which have the minimum path cost from the root to the state. For instance, when the two 11 nodes in the second level are merged, there are two paths, 00-10-11 and 00-01-11. Since they both cost 7, both transition $a/2$ and $a/1$ are kept. In the next level, again 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 paths using $a/1$ cost 11 while the paths using $a/2$ cost 12. The final reduced directed acyclic graph is shown in Fig. 4(b).

This idea of trimming paths in the tree is formalized in the following algorithm DELEGATE_FSM. The algorithm starts from the root level of the tree and at each iteration it selects the paths that are shortest to the next level. Note that Step 5 in the algorithm implements the selection.

DELEGATE_FSM($a_1, \ldots, a_n$) {
/∗ $c(x)$ *denotes the cost of transition* x
    $c(x)$ *also denotes the shortest path cost arriving at configuration* x
    $t(x, y)$ *denotes a transition from configuration* x *to configuration* y ∗/
1: E := { initial configuration };   /∗ E *contains frontier configurations* ∗/
2: For each a_i from a_1 to a_n
3:     Let G be the set of reachable configurations from E with a_i
4:     $\forall g \in G$
5:         record each transition $t(e, g)$ where $e \in E$ such that
            $\forall e' \in E \; \forall t' \; t'(e', g)$ implies $c(e) + c(t) \leqslant c(e') + c(t')$
6:         $c(g) := c(e) + c(t)$
7:     $E := G$
8: Find an accepting configuration $x \in E$ with minimum $c(x)$
9: Backtrack the path from x to the initial configuration and
    output the corresponding delegation }

We can prove the following Lemma.

Lemma 1. *Let $A_1, \ldots, A_r$ be a community of services, $a_1 \cdots a_n$ be the input sequence of activities, and T be the complete computation tree constructed by unfolding the product machine of $A_1, \ldots, A_r$, with respect to $a_1 \cdots a_n$. For each $1 \leqslant j \leqslant n$, if p is a path from the root of T to the j-th level of T with the lowest cost, then every configuration on the path p is assigned to the frontier set E by the algorithm* DELEGATE_FSM.

Proof. Let f_0 denote the initial configuration. Each node is labeled with a configuration. The root of the tree T is labeled f_0. Each edge a_k/z represents the delegation of the activity a_k to a service z where $z \in \{A_1, \ldots, A_r\}$ and $1 \leqslant k \leqslant n$. The cost of an edge a_k/z is the cost of the corresponding transition in service z.

The tree T has n levels where the root level is 0. Each path from the root to an accepting configuration at level n corresponds to a valid delegation of $a_1 \cdots a_n$. We call such a path an *accepting path*. The cost of each (accepting) path is the sum of the cost of the edges and represented by $c(p)$ for a path p. Let $p = f_0\ a_1/i_1\ f_1\ a_2/i_2 \cdots a_j/i_j\ f_j$ be a path with the smallest cost for $a_1 \cdots a_j$ where f_k represents the configuration the path visits at level k. We prove the claim by induction on j.

- *Basis*: Let g be a configuration at level 1 and $t_{y_1}, \ldots, t_{y_q}$ be the transitions from f_0 to g. Also assume that $c(t_{y_1}) \leqslant c(t_{y_2}) \leqslant \cdots \leqslant c(t_{y_q})$. In Line 5, the algorithm removes all transitions except t_{y_1}.
  . *Case* 1: $g \neq f_1$
  The path p is not removed.
  . *Case* 2: $g = f_1$ and $t_{y_1} = a_1/i_1$
  The path p is not removed.
  . *Case* 3: $g = f_1$ and $t_{y_1} \neq a_1/i_1$
  For the path $p' = f_0 t_{y_1} f_1 a_2/i_2 f_2 \cdots a_j/i_j f_j$, it must be that $c(p') = c(p)$, otherwise the cost of p cannot be the lowest; therefore, both paths p, p' have the lowest cost and they are not removed.
- *Inductive Step:* Assume the reduction in Line 5 doesn't remove all optimal paths up to level m (i.e., the prefix $a_1 \cdots a_m$ is delegated). We prove that the reduction at level $m+1$ also keeps all paths with the lowest cost at that level.
  Let h be a configuration at level $m+1$, and let G be the configurations at level m each having a transition to h. Suppose that $p = f_0\ a_1/i_1\ f_1 \cdots\ a_{m-1}/i_{m-1}\ f_m a_m/i_m\ f_{m+1}$ is a path with the lowest cost in the current reduced tree. For some $g \in G$, let $c(g)$ represent the cost of the paths reaching g (Note that there is only one such cost because of the reductions applied up to level m). Let $X(G, h) = \{x \mid x$ is a transition from g to h for some $g \in G\}$ and $X(g, h) = \{x \mid x$ is a transition from g to $h\}$.
  Let's say, for a transition t from g_1 to h, the cost $c(g_1) + c(t)$ is the minimum of $\{c(g_i) + c(t_j) \mid g_i \in G$ and t_j is in $X(g_i, h)\}$. The reduction thus removes all transitions in $X(G, h)$ except for t. We have the following cases:
  . *Case* 1 : $h \neq f_m$.
  The path p with the lowest cost is not removed.
  . *Case* 2 : $h = f_m$.
  Subcase 2.1 : $g_1 = f_{m-1}$ and $t \neq a_{m-1}/i_{m-1}$
  The path where t is replaced by a_{m-1}/t_{m-1} in p is also an optimal path, since $c(t)$ must be equal to $c(a_{m-1}/i_{m-1})$, otherwise, p is not an optimal path.
  Subcase 2.2 : $g_1 = f_{m-1}$ and $t = a_{m-1}/i_{m-1}$
  The path p is not removed.
  Subcase 2.3 : $g_1 \neq f_{m-1}$
  The path reaching to g_1 followed by $t\ f_m\ a_m/i_m\ f_{m+1} \ldots a_n/i_n\ f_n$ is an optimal path because $c(g_1) + c(t) \leqslant c(f_{m-1}) + c(a_{m-1}/i_{m-1})$.
  In all cases, all paths with the least cost remain in the reduced tree (i.e., the end node of the path is assigned in the frontier set). □

The following follows immediately.

Lemma 2. *The reduction of the transitions done at Line 5 in Algorithm* DELEGATE_FSM *does not remove any optimal delegations.*

From this lemma, we can conclude the following:

Theorem 2. *Algorithm* DELEGATE_FSM *correctly computes an optimal delegation (nondeterministically).*

Proof. Lemma 2 implies that the final structure contains at least one optimal path. It is clear that the cost of each path reaching to a configuration is computed correctly. At the end, in the final level, an accepting configuration labeled with the minimum cost represents the final configuration reached with the delegation of the activities. Backtracking the path from this configuration to the root gives the optimal delegation of the sequence. □

Theorem 3. *An optimal delegation of a word w of length n over a community of r FSM services each with at most c states can be computed in $O(c^r * n)$ time.*

Proof. The time complexity of the proposed technique can be analyzed as follows: At each level of the reduced tree 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 tree equals the number of states in the final level which is at most c^r. Since the length of the string w is n, there are c^r paths of length n. Therefore, the construction of the reduced tree 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 DELEGATE_FSM takes $O(c^r * n)$ time. □

In our case the community of services is not part of the input to the problem, therefore, we conclude the following.

Theorem 4. *An optimal delegation of a word w can be computed in time linear in the length of w (the community of services is considered fixed).*

Proof. Follows from Theorem 3 and the fact that the community of services is not an input to the problem. □

It can be shown that the algorithm 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 that contains the input string $a_1 a_2 \cdots a_n$ including endmarkers 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.

Note that the algorithm DELEGATE_FSM has an exponential constant factor c^r where r is the number of services each with at most c states. One natural question is whether we can improve this result by reducing this factor to a bound that is polynomial in c and r. Such an algorithm would be preferable. The following result, however, shows that this is unlikely since such an algorithm would imply $P = NP$.

Theorem 5. *For the optimal delegation problem, suppose there is an algorithm of time complexity $f(c, r) * n$ where f is a function of c and r, r is the number of services each has at most c states, n is the length of the input word. If $f(c, r)$ is bounded by a polynomial in c and r, then $P = NP$.*

Proof. Suppose such an algorithm A exists. Consider the following problem: Given a finite collection of $r + 1$ strings $x, y_1, \ldots, y_r$ (over a fixed alphabet of length at least 3), determine if x can be written as a shuffle of strings $y_1, \ldots, y_r$. [36] shows that this problem is NP-complete. We can now use the algorithm A to solve this problem as follows: For each i, let A_i be a deterministic FSM (DFSM) that accepts the string y_i, and assign cost 0 for all the edges for all DFSMs. Apply the algorithm A and output "yes" if and only if the optimal cost is 0. Note that $|A_i| = |y_i| + 1$. This gives an algorithm of polynomial time complexity for the shuffle problem and it follows that $P = NP$. □

Note that this theorem essentially proves that the optimal delegation problem is NP-hard. While the complexity of the problem could be higher, in Theorem 3 we provide an EXPTIME upper-bound.

5. Delegation for linear counter services

In this section, we generalize Theorem 4 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). Let $\{A_1, \ldots, A_r\}$ 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 similar to the FSM case. Rather than defining 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 is also a $\log n$ space bounded TM and 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.

More formally, the configuration of the product machine *PROD* will be of the form $< a_1 a_2 \ldots a_n, S, j, T_1, T_2, \ldots, T_r >$. Here, the input string is $a_1 a_2 \ldots a_n$, $S \subset S_1 \times \ldots \times S_r$ (S_i is the state set of A_i), $1 \leqslant j \leqslant n$, and T_k ($1 \leqslant k \leqslant r$) is a partial configuration of A_k. This configuration represents the following scenario: at this point, the first $j - 1$ symbols have been collectively processed by the services $\{A_k\}$, and as a result, they have advanced from their respective starting configurations to the tape configurations $\{T_k\}$.

5.1. Polynomial-time optimal delegation algorithm

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.

Next we propose an algorithm which given a sequence of requests, computes a delegation over a community of log n-space bounded TMs.

```
DELEGATE_LC(a_1, ..., 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
6:        E = set of configurations reachable from current configuration
             with a_i in PROD
7:        if i < n
8:           Find a configuration, say ID_p, in E such that
                a_{i+1} ... a_n is accepted by PROD_{ID_p}
9:        else
10:            Find an accepting configuration, say ID_p, in E
11:       Output A_j that corresponds to the transition
              from current configuration to ID_p (thus a_i is delegated to A_j)
10:       current = ID_p }
```

Lemma 3. *Let $a_1 \ldots a_n$ be a word accepted by* PROD. *Let c_i represent the configuration assigned to the variable current at the end of ith iteration of the loop starting in Line 5. For all $i < n$, there exists a path of length $n - i$ from c_i to an accepting configuration in* PROD *with the word $a_{i+1} \ldots a_n$, and c_n is always an accepting configuration.*

Proof. Let (ID_1, w, ID_2) represent the fact that the configuration ID_2 is reachable from ID_1 with the word w. We prove the first part by induction on the number of iterations.

- *Basis* ($i = 1$): Since $a_1 \ldots a_n$ is accepted by PROD (otherwise the iteration will not be entered), there must exist a configuration x such that (ID_0, a_1, x) and $(x, a_2 \ldots a_n, ID_f)$ for some accepting configuration ID_f in PROD. Line 8 assigns such a configuration x to the variable *current*. Therefore, at the end of the first iteration, *current* holds x. Since $(x, a_2 \ldots a_n, ID_f)$ is true, $(c_1, a_2 \ldots a_n, ID_f)$ is also true. Therefore, there exists a path of length $n-1$ from c_1 to an accepting configuration.
- *Inductive Step*: Let's assume for all $i < k < n$, it is true that there exists a path of length $n - i$ from c_i to an accepting configuration in PROD. By this assumption, $(c_{k-1}, a_k \ldots a_n, ID_f)$ is true for some accepting configuration ID_f. This implies that there exists a configuration x and an accepting configuration $ID_{f'}$ such that (c_{k-1}, a_k, x) and $(x, a_{k+1} \ldots a_n, ID_{f'})$ hold. Line 8 assigns such a configuration x to the variable *current*. Therefore, at the end of the kth iteration *current* holds x. Since $(x, a_{k+1} \ldots a_n, ID_{f'})$ is true, $(c_k, a_{k+1} \ldots a_n, ID_{f'})$ is also true. Therefore, there exists a path of length $n - k$ from c_k to an accepting configuration.

This further implies that there must exist a path of length 1 from c_{n-1} to an accepting configuration x and in the kth iteration, Line 10 assigns such a configuration x to the variable *current*. Therefore, c_n is always an accepting configuration. □

Lemma 4. *For an input word $a_1, \ldots, a_n$,* DELEGATE_LC *computes a delegation, if there exists one.*

Proof. By construction of the product machine there exists an accepting path in the product machine with a word w if and only if w can be delegated to the services successfully. In addition, the outputs generated by traveling such a path gives one possible delegation of w. We prove that the algorithm "travels" such an accepting path for every word which can be successfully delegated (let's call such words "delegable").

Let c_i represent the configuration assigned to the variable *current* at the end of ith iteration of the loop starting in Line 5. For a delegable word $a_1 \ldots a_n$, the path the algorithm travels is $ID_0 a_1 c_1 a_2 c_2 \ldots a_n c_n$. According to Lemma 3, c_n is an accepting configuration; therefore, the path is an accepting path which concludes the proof. □

Lemma 5. DELEGATE_LC *terminates in time polynomial in the length of the input word (the community of services is considered fixed).*

Proof. A $\log n$ space-bounded nondeterministic TM can be simulated by a polynomial time-bounded deterministic TM [22]; therefore, Line 1 can be implemented in polynomial time. Briefly, we can do a breadth first search and mark 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 also takes a polynomial number of steps, because, there is a polynomial number of configurations and for each configuration the reachability can be checked in polynomial time similar to Line 1. The same analysis applies to Line 8. Therefore, the algorithm is polynomial-time bounded. □

Theorem 6. *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 computed in polynomial time in the length of w (the community of services is considered fixed).*

Proof. Follows from Lemmas 4 and 5. □

Corollary 1. *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 (the community of services is considered fixed).*

Proof. Briefly, the algorithm DELEGATE_LC can be adapted to take processing costs into consideration. The technique is very similar to the technique we described previously in FSM services. First, at each iteration, a set of frontier configurations and their costs are recorded. When multiple versions of one configuration emerges, only the one with the minimum cost is kept and the rest is eliminated from the frontier configurations for the next iteration. Since the number of configurations in a log n space-bounded TMs is polynomial, each iteration takes polynomial time. At the end of the nth iteration, the configuration with the minimum cost can be found in polynomial time. Then, a path from this configuration to the initial configuration is backtracked in linear time to produce the delegation. □

5.2. $\log^2 n$-space Algorithm for Optimal Delegation

The algorithm proposed in the previous section uses polynomial space. In this section, we propose a $\log^2 n$-space algorithm for the optimal delegation problem by sacrificing time.

First we need a space efficient way to check whether one configuration can reach to another in s steps with a given input string on the tape. The next algorithms are due to Savitch [22]. More precisely, we start with two base cases to deal with a single input symbol a on the input tape. TEST1(a, ID_1, ID_2, s) represents the case in which the input head is scanning a at the start as well as at the end of the computation, with ID_1 and ID_2 representing the worktape configurations at the start and at the end. TEST2(a, ID_1, ID_2, s) represents the case in which the input head is scanning a at the start and has just moved to the right of a at the end, with ID_1 and ID_2 representing the worktape configurations at the start and at the end. In both cases, s represents the number of computation steps.

```
TEST1(a, ID1, ID2, s) { //Reachability with a symbol a in s steps with
//input head stationary
1:  if s = 1
2:    if ID2 is reachable from ID1 in one step with symbol a
3:        on the input tape, with input head not moving
4:      return true
5:    else
6:      return false
7:  else
8:    for each possible configuration ID3
9:      if (TEST1(a, ID1, ID3, ⌊s/2⌋)) and
10:        (TEST1(a, ID3, ID2, ⌈s/2⌉))
11:        return true
12:   return false
}
TEST2(a, ID1, ID2, s) { //Reachability with a symbol a in s steps with
//input head moving right one step
1:  if s = 1
2:    if ID2 is reachable from ID1 in one step with symbol a
3:        on the input tape, with the input head moving to the right
4:      return true
5:    else
6:      return false
7:  else
8:    for each possible configuration ID3
9:      if [(TEST1(a, ID1, ID3, ⌊s/2⌋)) and
10:        (TEST2(a, ID3, ID2, ⌈s/2⌉))] OR
11:        [(TEST2(a, ID1, ID3, ⌊s/2⌋)) and
12:        (TEST1(a, ID3, ID2, ⌈s/2⌉))]
13:        return true
14:   return false
}
```

The next algorithm checks the reachability between two configurations with a string of length greater than one on the input tape. The procedure REACHABLE takes i, j, ID_1, ID_2 and t as parameters and determines if the following is true: The Turing machine, starting with the input head scanning the leftmost symbol of the input string $a_i a_{i+1} \ldots a_j$ with ID_1 on the worktapes, can reach the configuration ID_2 on the worktapes, and with the input head just moving past the rightmost symbol a_j in t steps.

```
REACHABLE(i, j, ID_1, ID_2, t) { //Reachability with the substring a_i ... a_j
1:  if i = j
2:    if TEST2(a_i, ID_1, ID_2, t)
3:      return true
4:    else
5:      return false
6:  else
7:    for each possible configuration ID_3
8:      for each k from 1 to t − 1
8:      if REACHABLE(i, ⌊(i+j)/2⌋, ID_1, ID_3, k) and
9:        REACHABLE(⌈(i+j)/2⌉, j, ID_3, ID_2, t − k)
10:       return true
11:   return false
}
```

The space complexity analysis of the algorithm provides the following result:

Lemma 6. REACHABLE$(i, j, \text{ID}_1, \text{ID}_2, t)$ *takes* $O(\log^2 n)$ *space where* ID_1, ID_2 *are two configurations of a* $\log n$ *space-bounded TM,* $0 \leqslant j - i \leqslant n$ *and* $t = O(n^m)$ *for some constant m.*

Proof. Since the worktapes are bounded by $O(\log n)$, the number of steps in a nonlooping computation sequence is bounded by $O(n^m)$ and so, $t = O(n^m)$. The algorithm REACHABLE consists of two phases. In the first phase, recursive calls are made by dividing the substring by 2 each time. When the substring length is 1, the second phase starts where the algorithm TEST2 is used to check if two configurations are reachable in $t = O(n^m)$ steps (which is an upper-bound on the number of all possible configurations). The height of the call stack (the depth of the recursion) in the first phase is at most $\log n$ where n is the input length. Then, because of the second phase, $\log n^m$ more frames are added on top of $\log n$ frames. Therefore, the stack height is $O(\log n^m) = O(\log n)$. Each stack frame keeps 2 substring index positions, 3 configurations and the number of steps left. It can easily be seen that the size of one frame is $O(\log n)$. As a result, the stack size is at most $O(\log n * \log n) = O(\log^2 n)$ which is the space complexity of the algorithm REACHABLE. □

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. The algorithm DELEGATE_LC_SPACEEFFICIENT describes how to determine the delegation of symbols to the A_i's of a string $w = a_1 a_2 \cdots a_n$.

```
DELEGATE_LC_SPACEEFFICIENT(a_1, ..., a_n) {
1:  if REACHABLE (1, n, ID_0, ID_f, n^m) is false
2:    output error
3:  else
4:    current = ID_0
5:    i = 1
6:      for each possible configuration ID
7:        if REACHABLE (i, i, current, ID, n^m) and
            REACHABLE (i + 1, n, ID, ID_f, n^m)
8:            Output A_j that corresponds to the transition
                from current configuration to ID (thus a_i is assigned to A_j)
9:            i + +
10:           current = ID
}
```

Theorem 7. *For a set of services modeled as* $\log n$ *space-bounded TM's where n is the input length, a delegation of a word* w*, can be computed in* $O(\log^2 n)$ *space.*

Proof. The algorithm DELEGATE_LC_SPACEEFFICIENT basically determines the first move that moves the head past i on the input tape of the PROD machine and assigns the i-th symbol to the appropriate service. It can be implemented on a $\log^2 n$-space bounded TM. For each input symbol, the algorithm REACHABLE is executed which requires $\log^2 n$ space by Lemma 6. Each substring position and configurations require $\log n$ space. As a result, the total space complexity of is $\log^2 n$. The correctness of the algorithm is obvious since the algorithm exhaustively searches all possible paths through which ID_f is reachable from ID_0. □

We informally describe how to extend this algorithm for finding an optimal delegation. Line 2 of REACHABLE can be changed to return the cost of the transitions with a_i. Also, The costs returned from Line 8 and 9 can be summed and returned. Then, DELEGATE_LC_SPACEEFFICIENT has two passes. In the first pass, it computes the costs of all paths, and keep the minimum cost value. The cost of a path can be stored in $O(\log n)$ space (the cost value can be at most $n^m \times c \times n$ for some constant c representing the cost of a step). In the second pass, the costs returned with Line 7 can be summed and check whether it is equal to the minimum cost value. If it is, then the corresponding delegation is output. Therefore, we have the following result.

Corollary 2. *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* $O(\log^2 n)$ *space.*

Note that the algorithm above takes time $O(n^{\log n})$ and its main advantage is the reduced space requirement.

6. Related work

The preliminary results from this paper were presented in [18]. This paper is related to the work presented in a series of papers [1–4,6,11,10,17,20,31]. [4] defines a service framework and studies the problem of automated composition synthesis. One input is a set of descriptions of 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, is proven to be computable in EXPTIME. It has been brought to our attention by one of the referees that this problem has been shown to be EXPTIME-complete by Muscholl and Walukiewicz [29]. [11,10] study generalizations to automata with unbounded storage where decidability and undecidability of the composability problem are shown. [6] extends the framework to allow interactions among existing services, which provides more flexibility to the services to achieve the desired service behavior. In [17], 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 a part of the input and a composite service is created through composition of existing services. This construction happens offline. In this study, instead of a desired service, an instance of an execution is given. As a result, we don't have any knowledge to use for preprocessing before run time. In addition, the earlier work in [4,6,17] were based only on standard finite state machines and did not have cost functions associated with the activities.

In [1], the notion of "parallel composition" is introduced and some results are obtained regarding the degree of parallelism that can be achieved in the composition of services. The authors define the notion of speedup to measure the amount of parallelism. It is proven that the speedup problem has the same complexity lower bound as the limitedness for distance automata, which lies between PSPACE and EXPTIME. Also, a parallel PTIME transducer for the composition problem is shown which runs with maximal parallelism. In [20], Roman model is extended with a merge operator allowing for simultaneous execution of actions. At each step, multiple services are allowed to process an input symbol. In other words, one or more symbols are processed by multiple services simultaneously at each step. Forward simulation technique is used for the computation of a delegator.

Recently, the Roman model of [4] was extended to include messages among services [3] and the impacts of activities on the real world are modeled as a relational database. Two types of composition models are studied namely, composition synthesis problem and choreography synthesis problem. The desired behavior is achieved via a central mediator in the former, while in the latter, the coordination of existing services is done in a distributed manner. This work also considers devilish nondeterminism.

The idea of service communities we used in this study has a similar flavor with the ones studied in [4,2]. Our notion of service communities is closer to the one of [4]. An important difference between [2] and [4] is that in [2], 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.

The batch-style specification of users' requests has similarities with a choreography specification. A choreography specification is a collaboration protocol among multiple services describing their interactions to achieve a desired goal [24]. There are studies such as [8,14,24] in the literature that focus on verifying whether services can interact in a way that is not conformant to the desired choreography. There are two main differences between these studies and ours. First, we model the desired sequence of actions performed by services while they focus on the sequence of messages exchanged by services. Second, our goal is to compute a way to delegate desired actions to available services while theirs is to verify if service interactions can result in an undesired state.

Service composition is also closely related to planning [27], where existing tasks are put together for a given goal. An approach to automated composition [30] has been developed for the OWL-S model [9]. 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. [31] uses planning via symbolic model checking and proposes a promising technique to generate an executable BPEL process from a set of abstract BPEL descriptions.

The problem was also considered in the context of workflows. In [35], the global dependencies are given as a tree, with "optional" and "choices" on some dependencies, resembling the event algebra [34]. 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, [26] 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 [7,15] 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 our model is activity-based.

In terms of service modeling, there are three main approaches: Activity-centric modeling (such as [4,6,17]), message-centric modeling (such as [7,15]), and hybrid models (such as [33,3]). In this paper, we adopted the activity-centric approach where the focal point of the modeling is service activities (excluding message sends and receives). Compared to this approach, in message-centric models [7,15], exchanged messages are the center of attention. In [33,3], hybrid models are proposed where both performed activities and exchanged messages are specified. Each approach gives a different perspective on the composition problem and has different complexity characterizations.

We also would like to point out the tutorial by De Giacomo and Mecella [28] which categorizes service composition studies in 4 dimensions the dimensions being the statics of the system, the dynamics of the target service, the dynamics of the component services, and the degree of completeness. According to this categorization, our study lies in the high parts of the dynamics of target services and component services dimensions, and in the low parts of the statics of the system dimension. For more information on the categorization and the categorization of other service composition studies, we refer the interested reader to this tutorial.

7. Conclusion and future work

We investigated online and minimum-cost delegation problem in service communities where services are modeled as FSMs augmented with linear counters. We formally analyzed the problem and give complexity bounds. We hope to see software implementations inspired from the techniques presented in this paper. In future, we plan to extend the model in various ways. For instance, the current model doesn't capture data requirements and data dependencies among services. It also doesn't address the possible impacts of activities on the real world such as what data each service creates/updates/deletes. The first results on this matter can be found in [3].

Acknowledgements

We are grateful to anonymous referees for their truly helpful comments that improve the presentation of our results.

References

[1] T. Ahmed, G. Grahne, Parallel composition of finite state activity automata, in: Proc. Descriptional Complexity of Formal Systems, DCFS, Las Cruces, NM, USA, 2006.
[2] B. Benatallah, M. Dumas, Q.Z. Sheng, A. Ngu, Declarative composition and peer-to-peer provisioning of dynamic web services, in: Proc. Int. Conference on Data Engineering, ICDE, 2002.
[3] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, M. Mecella, Automatic composition of transition-based semantic web services with messaging, in: Proc. Very Large Databases, VLDB, 2005.
[4] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, Automatic composition of e-services that export their behavior, in: Proc. Int. Conf. on Service Oriented Computing, ICSOC, in: LNCS, vol. 2910, 2003.
[5] D. Berardi, D. Calvanese, G. De Giacomo, M. Mecella, Composition of services with nondeterministic observable behavior, in: Proc. Int. Conf. on Service Oriented Computing, ICSOC, 2005.
[6] D. Berardi, G. De Giacomo, M. Lenzerini, M. Mecella, D. Calvanese, Synthesis of underspecified composite e-services based on automated reasoning, in: Proc. Int. Conf. on Service Oriented Computing, ICSOC, 2004.
[7] T. Bultan, X. Fu, R. Hull, J. Su, Conversation specification: A new approach to design and analysis of e-service composition, in: Proc. Int. World Wide Web Conf., WWW, 2003.
[8] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, G. Zavattaro, Choreography and orchestration conformance for system design, in: Proc. Int. Conf. on Coordination Models and Languages, ICCML, 2006.
[9] OWL Services Coalition. OWL-S: Semantic markup for web services, 2003.
[10] Z. Dang, O.H. Ibarra, J. Su, On composition and lookahead delegation of e-services modeled by automata, Theoretical Computer Science 341 (2005) 344–363.
[11] Z. Dang, O.H. Ibarra, J. Su, Composability of infinite-state activity automata, in: Proc. Int. Symp. on Algorithms and Computation, ISAAC, Hong Kong, 2004.
[12] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Communications in ACM 51 (1) (2008) 107–113.
[13] P.C. Fischer, A.R. Meyer, A.L. Rosenberg, Counter machines and counter languages, Mathematical Systems Theory 2 (1968) 265–283.
[14] H. Foster, J. Kramer, S. Uchitel, J. Magee, Ws-engineer: A rigorous approach to engineering web service compositions and choreography, in: Proc. XML 2006 Conference, 2006.
[15] X. Fu, T. Bultan, 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.
[16] X. Fu, T. Bultan, J. Su, Analysis of interacting BPEL web services, in: Proc. Int. World Wide Web Conf., WWW, May 2004.
[17] C.E. Gerede, R. Hull, O.H. Ibarra, J. Su, Automated composition of e-services: Lookaheads, in: Proc. of Int. Conf. on Service Oriented Computing, ICSOC, 2004.
[18] C.E. Gerede, O.H. Ibarra, B. Ravikumar, J. Su, Online and minimum-cost ad hoc delegation in e-service composition, in: Proc. IEEE Int. Conf. on Services Computing, SCC, 2005.
[19] G. De Giacomo, S. Sardina, Automatic synthesis of new behaviors from a library of available behaviors, in: Proc. Int. Joint Conf. on Artificial Intelligence, IJCAI, 2007.
[20] G. Grahne, V. Kricenko, Process mediation in an extended roman model, in: Proc. Mediation in Semantic Web Services, MEDIATE, 2005.
[21] Hadoop home page. http://hadoop.apache.org/core/.
[22] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, 1979.
[23] R. Hull, J. Su, Tools for design of composite web services, in: Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004.

[24] R. Kazhamiakin, M. Pistore, Choreography conformance analysis: Asynchronous communications and information alignment, in: Proc. Web Services and Formal Methods, WSFM, 2006.
[25] J.O. Kephart, D.M. Chess, The vision of autonomic computing, IEEE Computer 1 (2003) 41–50.
[26] S. Lu, Semantic Correctness of transactions and workflows, Ph.D. Thesis, SUNY at Stony Brook, 2002.
[27] S.A. McIlraith, T.C. Son, Adapting Golog for composition of semantic web services, in: Proc. Int. Conference on Principles and Knowledge Representation and Reasoning, KR-02, 2002.
[28] M. Mecella, G. De Giacomo, Automatic web service composition, in: Proc. IEEE Int. Conf. on Web Services, ICWS, 2006.
[29] A. Muscholl, I. Walukiewicz, A lower bound on web services composition, in: Proc. Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS, 4423, 2007.
[30] S. Narayanan, S. McIlraith, Simulation, verification and automated composition of web services, in: Proc. Int. World Wide Web Conf., WWW, 2002.
[31] M. Pistore, P. Traverso, P. Bertolli, A. Marconi, Automated synthesis of composite bpel4ws web services, in: Proc. IEEE Int. Conf. on Web Services, ICWS, 2005.
[32] R. Ramakrishnan, et al., Science of design for information systems, ACM SIGMOD Record 33 (1) (2004) 133–137.
[33] Z. Shen, J. Su, Web service discovery based on behavior signatures, in: Proc. Int. Conf. on Service Oriented Computing, ICSOC, 2005.
[34] M. Singh, Semantical considerations on workflows: An algebra for intertask dependencies, in: Proc. Workshop on Database Programming Languages, DBPL, 1995.
[35] W.M.P. van der Aalst, On the automatic generation of workflow processes based on product structures, Computer in Industry 39 (2) (1999) 97–111.
[36] M.K. Warmuth, D. Haussler, On the complexity of iterated shuffle, Journal of Computer System Sciences 28 (3) (1984) 345–358.