Journal of Automata, Languages and Combinatorics **u** (v) w, x–y
© Otto-von-Guericke-Universität Magdeburg

# QUALITY-AWARE SERVICE DELEGATION IN AUTOMATED WEB SERVICE COMPOSITION: AN AUTOMATA-THEORETIC APPROACH [1]

OSCAR H. IBARRA
*Computer Science Dept., University of California, Santa Barbara, Santa Barbara, CA 93106, USA*
*e-mail:* `ibarra@cs.ucsb.edu`

BALA RAVIKUMAR
*Computer Science Dept., Sonoma State University, Rohnert Park, CA 94928, USA*
*e-mail:* `ravi.kumar@sonoma.edu`

and

CAGDAS EVREN GEREDE
*Computer Science Dept., University of California, Santa Barbara, Santa Barbara, CA 93106, USA*
*e-mail:* `gerede@cs.ucsb.edu`

ABSTRACT

*Automated Web Service Composition* has gained a significant momentum in facilitating fast and efficient formation of business-to-business collaborations where an important objective is the utilization of existing services to respond to new business requirements in a timely manner. In this context, the service delegation problem can be formulated as follows: When a user poses a sequence of requests to a "service community", how to delegate the requests to the available services registered in the community so that the user requests are satisfied via a collaboration of these services. Here, we present a formal analysis of the *constrained service delegation* problem where users also provides a set of quality constraints about the delegation of their requests. We follow the "Roman" service composition framework and extend it with a QoS model. We use the Presburger arithmetic to specify constraints. We show that there exists a linear time algorithm for the service delegation problem. In fact, this algorithm is a finite memory algorithm that solves two variations of the service delegation problem by reading the activity sequence in two or multiple passes. We also show that these results are tight in the sense that the number of passes can't be further reduced. We also prove that the constrained service delegation problem can be solved in polynomial time in the number of service requests and delegation constraints.

*Keywords:* Automated Service Composition, Quality of Service, Deterministic Finite Transducers, Presburger Arithmetic, Suffix Problem

---

[1]The research of Oscar H. Ibarra and Cagdas Evren Gerede was supported in part by NSF Grants CCR-0208595 and CCF-0430945.

## 1. Introduction

The Web services paradigm is about the fact that the benefits of the Internet as a platform can apply not only to information but to services as well. This paradigm promises to facilitate inter-operation of highly distributed and heterogeneous services which are loosely-coupled, reusable and network-enabled applications. A significant progress towards this goal has been made by emerging standards (e.g., SOAP [14], WSDL [16], BPEL [4]), industrial technology (e.g., IBM's WebSphere, Microsoft's .NET, Sun's J2EE) and several research efforts. Our effort here is on the analysis and understanding of automated service composition.

Automated service composition has similarities with information system integration, workflows, automated planning, and distributed computing ([10], [12]). The main objective is to respond to new business requirements automatically by leveraging the existing assets. This objective is achieved by putting the existing pieces together, and facilitating a collaboration among these autonomous pieces. One proposal toward this goal is *service communities* ([1], [2]). A service community is a group of service providers and service users of similar interests. Service providers register their services for the community use. When a service user asks for the execution of some activities, there may not be a stand-alone service that is capable enough to satisfy the request. There may be, however, a way to coordinate the services in the community to perform the job in a collaborative manner. The goal of this paper is to study the automatic generation of this type of collaborations.

The next example illustrates the studied problem. In this example and the rest of the paper, the "Roman" service framework ([2], [3], [6], [7]) is used as the service model. Finite state automata (FA) are used to model service execution logics where each labelled transition represents an execution of an activity (see Section 3 for more details).

**Example 1** Figure 1 illustrates the specifications of 5 services desired by a travel service community. For instance, $S1$ is a book-air-travel service which can be used to book a flight and a limo. The final (initial) state is represented by a double circle (a circle with a triangle). Each transition label describes the activity performed and the quality of the transition. For instance, let's say, the execution cost of book-plane activity by $S1$ should be between \$10 and \$20 dollars, and the execution time should be between 5 and 10 seconds (the units are assumed to be defined in the community ontology). This can be represented as $Q_{11} = \{$(price, $\langle 10, 20 \rangle$, \$), (duration, $\langle 5, 10 \rangle$, sec)$\}$. The service providers joining the community register their service instances with the appropriate service specifications. The registration of a service instance under a service specification implies that the registered service instance can satisfy the specified service behavior with some quality lying in the specified quality ranges. For instance, a service instance registering for $S1$ has to execute the book-plane activity between 5 and 10 seconds.

In this setting, let's say, a community user asks for the execution of the sequence of the activities book-air-plane, book-shuttle, register-event, book-hotel and book-limo. Also, the user wants the execution time to be less than 10 seconds and the execution cost to be less than \$20. Unfortunately, none of the specifications describes such a service. On the other hand, the activities can be delegated to any 3 services each of which satisfies $S1$, $S2$ and $S4$ respectively, and these services can be coordinated to perform the execution successfully.

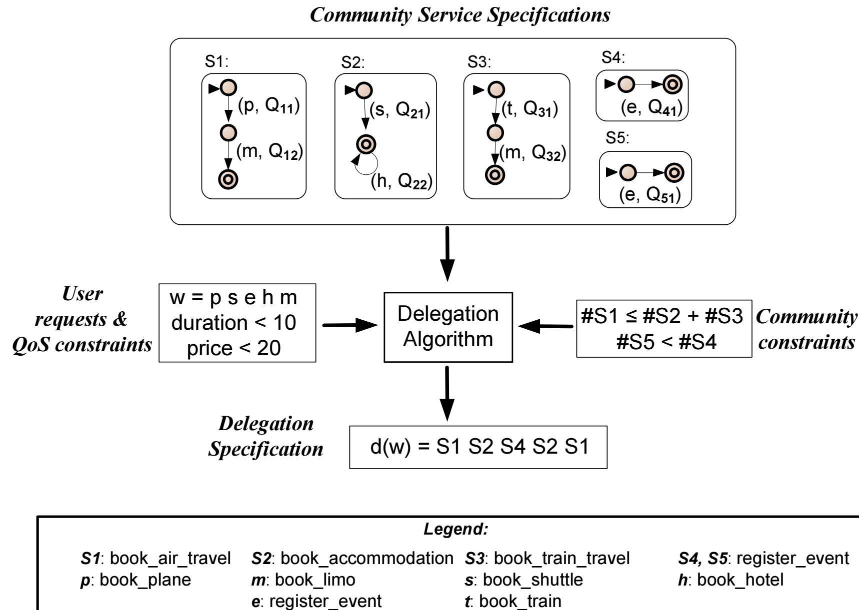


Figure 1: Problem Overview: A travel service community

Figure 1 shows one such delegation described as d(w) = $12421$.[2]

The community also can have some constraints about a delegation. This kind of constraints give the community the power to control the execution of services. For instance, the community may favor the usage of $S2$ and $S3$ rather than $S1$ which can be represented as the constraint $\#S1 < \#S2 + \#S3$ (the number of activities executed by $S1$ should be less than the sum of the ones by $S2$ and $S3$). These constraints can also be used to affect the performance characteristics of the community. For example, the constraint $\#S5 < \#S4 < 2{*}\#S5$ specifies that the work load on $S4$ should be about within a factor two of the workload on $S5$.

Our goal in this paper is to develop a delegation algorithm which takes user requests and delegation constraints as the input and produces a specification describing a satisfactory delegation of the requests to the service specifications published by the service community.

□

This paper makes the following contributions:

- We extend the “Roman” model [2] with a quality of service model to capture the cases where different services process the same activities with different qualities.
- We define the constrained service delegation problem. Given a community of services, service requests (a sequence of activities), and a set of quality and community constraints, we study the problem of how to compute a delegation of requests which sat-

[2]Note that for the clarity of the discussion, here, we explain a simpler version of a delegation specification. In fact, as described in Section 3, the specification includes not only which activity is delegated to which service, but also which transitions are taken by each service.

isfies the constraints and describes a way to process the requests using the community of services in a collaborative manner.

- We show that there is a linear time algorithm for the service delegation problem (no constraints). In fact, this algorithm is a finite memory algorithm that solves two variants of the problem by reading the activity sequence in two or multiple passes. We also show that these results are tight in the sense that the number of passes cannot be reduced.
- We prove that the constrained service delegation problem can be solved in polynomial time in the number of the service requests and the delegation constraints.

The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 defines our service model and the constrained delegation problem. In Section 4, we present the technique used in the study of delegation problem in a more abstract formal language theory framework. Section 5 and Section 6 present our main results on the delegation problems. In Section 7, a related problem of multiplicity of delegations is studied. Section 8 concludes the paper.

## 2. Related Work

Our work is most closely related to the works in [1], [2], [3], [5], [6], [7], and [17]. [2] defines a service composition framework, often referred to as "Roman" model, and studies the following problem: Given a number of existing service descriptions and a desired service, how to find a delegator (a specialized kind of mediator) that will achieve the desired behavior by coordinating the activities of the existing services.

In [2], [3] and [6], the desired service is given as a finite state automaton; and the problem is to construct a delegator that can output the delegation of any sequence of requests acceptable by the desired service. This problem is more service provider-oriented in that the provider would like to construct her desired service. In other words, she specifies the service she would like to have and the algorithm computes a delegator so that the new composite service can serve its clients successfully by delegating the tasks to the existing services. Depending on the given system of services, the delegator varies from being a simple FA to a universal model such as a (polynomial time bounded) Turing machine.

The problem of computing a delegator, if there is one, is shown to be solvable in EXPTIME [2]. In [3], the model was extended to allow interaction among existing services, and therefore the delegation decisions are partly given to the services involved which provides more flexibility to the services in order to achieve the desired behavior. In [6], the delegator notion was extended to have "look ahead", i.e., the delegation decisions may depend on future activities. [5] studied generalizations of the model of [2] to automata with unbounded storage where decidability and undecidability of the composability were shown. Compared to these studies, we are given a desired execution, instead of a desired service. In that sense, our work is more service user-oriented.

[7] extends the model with processing costs attached to each activity and studies the problem of computing the cheapest service delegation. It has been shown that the delegation can be computed in time linear in the length of any input. In contrast to [7], here we don't consider the optimality of a delegation; on the other hand, the model defined here is more general and can model several quality metrics including execution costs.

Service communities are also studied in [1] and [2]. In [2], service providers export their services to a community, and what additional services can be provided from the registered services is computed automatically. On the other hand, in [1], service communities describe the services they would like to provide, and service providers register their services which can satisfy the communities' specifications. Our notion of service communities is closer to that of [1].

[17] proposes a QoS-aware middleware for service composition. Compared to our work, in [17], each quality metric represents the exact quality of an operation or a service, whereas in our case the quality metrics are ranges. Also, [17] studies how to select services to form a given composite service to satisfy a set of quality constraints. In their case, a composite service schema (represented as a state chart) specifies which activity is performed by which type of service. Therefore, by construction, the delegation information is included in the composite service schemas. In our case, however, the given execution sequence doesn't contain this information.

Note that the task of service composition consists of two main steps, namely *composition synthesis* and *orchestration* [2]. Composition synthesis refers to the process of computing a specification of how to delegate user requests to the composition components to answer users' needs. On the other hand, orchestration refers to the actual run-time coordination of service executions, considering data and control flow among the components. In this paper, we study the synthesis problem.

## 3. Preliminaries

### *3.1. Service Model and Service Communities*

This section presents a formal setting for services in order to get a precise characterization of the service delegation problem. For this purpose, we follow the model presented in [2] (often referred to as the "Roman" model).

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 (activities performed and their execution order). It hides some of the details about the internals. An *internal schema*, on the other hand, specifies the full logic of how the activities are actually executed. In other words, while an internal schema provides the complete service specification, an external schema contains partial information and hides some of the business logic from service users. This study focuses on external service schemas. In the remainder of the paper, we, therefore, use 'schema' to mean 'external schema'.

A *service instance* refers to one occurrence of a service among several 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 a client sends requests to a service, the service may perform each requested activity on its own or it can *delegate* the processing of them to other services. A *simple service*

processes all requests on its own, while a *composite service* may delegate some or all to others.

A *service community* consists of a set of services of similar interests and defines a community ontology (a common language agreed by the community). Based on its requirements, the community defines (possibly composite) external service schemas. These schemas describe specifications of desired services. Each such schema is realized by actual services which join to the community by registering themselves with appropriate external schemas. This process is facilitated by means of the community ontology ([2], [1], [17]).

The model introduced so far doesn't refer to any specific form of service schemas. As formalized next, this study focuses on the services whose external service schemas are represented as *nondeterministic finite state automata (NFAs).*

**Definition 1** *A (possibly composite) service is a nondeterministic finite state automaton (NFA)* $A = (S, \Sigma, \delta, s^0, F)$ *where* $S$ *is the finite set of states,* $\Sigma$ *is the input (activity) alphabet,* $\delta$ *is a mapping from* $S \times \Sigma$ *to* $2^S$*,* $s^0 \in S$ *is the starting state,* $F \subseteq S$ *is the set of accepting states.*

Intuitively, a service $A$ is viewed as an acceptor. Each labeled transition corresponds to the processing of the corresponding activity. The notion of a word *accepted* by $A$ is defined in the standard manner. If a word is accepted by $A$, it means the word represents an executable sequence of activities conforming to the external service schema of $A$.

### *3.2. Quality Model*

We extend the model so that each external service schema has associated quality of service attributes. Here for illustration purposes, we use 5 generic quality metrics, which are execution duration, execution price, availability, reputation, and reliability, but our results can be extended to include other quantifiable quality of service metrics.

#### *3.2.1. Service Quality for an Activity:*

We model the QoS attributes as activity level attributes. In other words, each transition of an external schema has a QoS vector. Each attribute in such a vector is represented by a numeric range $\langle i, j\rangle$ ($\langle i, i\rangle$ means 'exactly $i$'). When a community defines an external service schema, the specified QoS ranges describe the expectation of the community from the service instances registering themselves to realize the schema. A schema, therefore, is an "umbrella" specification, representing a set of services which are behaviorally the same, but qualitatively varying in a specific range. For example, $\langle 10, 20\rangle$ for the execution price of a transition on a schema shows the service instance realizing the schema has to charge between 10 and 20 units when serving the transition (the unit details should be defined in the community ontology). Note that we assume the greater the value of a quality attribute is, the better quality it represents. If that is not the case, then the attributes can be normalized as described in [17]. For instance, the execution price is such an attribute; but the provided value can be treated as negative, which is a simple way to normalize such attributes.

We can represent the quality vector of a transition $t$ as

$$Q_t = \{\langle t_l^{av}, t_h^{av}\rangle, \langle t_l^{dur}, t_h^{dur}\rangle, \langle t_l^{pr}, t_h^{pr}\rangle, \langle t_l^{rel}, t_h^{rel}\rangle, \langle t_l^{rep}, t_h^{rep}\rangle\}$$

where each tuple represents low and high quality values of this transition for availability, duration, price, reliability and reputation metrics. Given two quality vectors $Q_t, Q_u$, the sum $Q_t + Q_u$ is defined simply as a quality vector consists of the sum of the ranges of $Q_t$ and $Q_u$. For example, the price range of $Q_t + Q_u$ is $\langle t_l^{pr} + u_l^{pr}, t_h^{pr} + u_h^{pr}\rangle$.

*3.2.2. Service Quality on a Path:*

An NFA service schema can be seen as a directed graph and a path beginning from the starting state represents how the service changes its state as it processes a sequence of activities. We define the quality on such a path as the sum of qualities of the processed activities. More formally, the quality on a path $P$ is defined as $Q_P = \sum_{t \in P_T} Q_t$ where $P_T$ is the set of transitions in $P$.

Next, we define the delegation of a word and then we define the quality of a delegation based on the quality model.

### *3.3. Service Delegation and Quality of Delegation*

*3.3.1. Service Delegation:*

When a user requests are received by the community, these requests are delegated to the appropriate service schemas and the requests are processed by the service instances realizing the schemas. In other words, the processing of the whole sequence of requested activities are realized by the collaboration of multiple services. The concept of delegating each activity to a service schema is called a service delegation [6]. Next we formalize the semantics of a service delegation.

Let's first define $\mathcal{C}, w, w'$ and $\text{image}_j^{w'}(w)$:

- $\mathcal{C} = (A_1, \ldots, A_r)$ denotes a community having $r$ service schemas.
- A request word $w = a_1 a_2 \ldots a_n$ is a sequence of activities requested from $\mathcal{C}$.
- An assignment word $w' = (x_1 b_1 y_1)(x_2 b_2 y_2) \ldots (x_n b_n y_n)$ represents each activity $a_i$ is assigned to the service schema $b_i$, and $b_i$ processes $a_i$ making a transition from the state $x_i$ to the state $y_i$, where $1 \leq i \leq n$, $1 \leq b_i \leq r$, and $x_i$, $y_i$ are two states of $A_{b_i}$
- $\text{image}_{w'}^j(w)$ ($j$-image of $w$ under $w'$) is the path beginning from the start state on service schema $A_j$ constructed from the sequence of transitions $A_j$ makes implied by the assignment word $w'$. If the number of transitions in the path is 0, then we say $\text{image}_{w'}^j(w)$ is $\lambda$. Note that if such a path cannot be constructed, we say the assignment word $w'$ is not *valid*.

Now we can define a service delegation of a word as follows:

**Definition 2** *A delegation of a word $w$ on a community $\mathcal{C}$ is a valid assignment word $w'$ where for every $A_j$ in $\mathcal{C}$, the path* $\text{image}_j^{w'}(w)$ *is either $\lambda$ or ends in an accepting state of $A_j$.*

**Example 2** Let $w'$ be $(p1q)(r2s)(s2t)(q1n)(k3r)(t2z)$ for $w = a_1a_2a_3a_4a_5a_6$. Then, $w'$ implies that $a_1, a_4$ are assigned to the service 1; $a_2, a_3, a_6$ are assigned to the service 2, and $a_5$ is assigned to the service 3. The transition path the services take are represented with the $\text{image}_j^{w'}(w)$ concept. For instance, $\text{image}_1^{w'}(w)$ shows that the service 2 follows the path $r \rightarrow^{a_2}\rightarrow s \rightarrow^{a_3}\rightarrow t \rightarrow^{a_6}\rightarrow z$.

**Definition 3** *If a word $w$ has a service delegation $w'$ on a community $\mathcal{C}$, then we say that $w$ is composable with respect to $\mathcal{C}$.*

*3.3.2. Quality of Delegation:*

Given a delegation $w'$ of a word $w$ on a community $\mathcal{C}$, the quality of $w'$ is $Q_{w'} = \sum Q_p$ where $p \in \{ \text{image}_j^{w'}(w) \mid A_j \in \mathcal{C}\}$ (the sum of qualities of the paths taken by the service schemas). Note that the quality of a path and the sum of two quality vectors are defined in the previous section.

Now we can define the constrained delegation problem.

**Definition 4** *Given a word $w$ of activities, a set of constraints $C$, and a service community $\mathcal{C}$, computing a delegation $w'$ on $\mathcal{C}$ such that $\langle w', Q_{w'}\rangle$ satisfies $C$ is called the constrained service delegation problem.*

What we mean by "$\langle w', Q_{w'}\rangle$ satisfies $C$" and the forms of constraints we model are described later in Section 6. In Section 5 we study a simpler problem of how to compute a service delegation without any constraints (i.e., $C$ is empty). Then, in Section 6, we examine the general problem and take quality and community constraints into consideration.

In the remainder of the paper, when the intent is clear from the context, we use the word "service" to refer to an external service schema.

## 4. Finite-state Transducers for the Suffix Problem

In this section we present two types of transducers we consider in this paper and study the *suffix problem*, which will be defined below. The results and the techniques presented in this section will be used in Section 5 to show some results about forward delegation.

- A ***1-way DFT*** (1-DFT for short) is a deterministic finite automaton $M$ with a finite control, a 1-way read-only input tape and a 1-way write-only output tape. Figure 2.a illustrates one such $M$. $M$ has a printing head on the output tape that moves from left to right printing symbols. At each step, as a function $\delta$ of its current state and the input symbol scanned on the (read-only) input tape, $M$ changes its state, moves the input head to the right and writes a word on the output tape and moves its output head to the left-most blank cell. (Initially, all the cells in the output tape are blank.) The transducer halts when it enters a special (halting) state. For convenience, we assume that there are left and right end-markers on the input tape. Note that $M$ need not print on every move. For any word $w \in \Sigma^*$ where $\Sigma$ is the input alphabet, let $y \in \Delta^*$ be the output where $\Delta$ is the output alphabet. The function $T(M) = \{(w, y) \mid y$ is the output

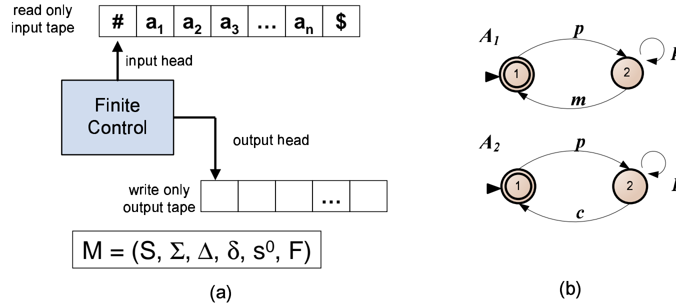


Figure 2: *(a)* A finite state transducer $M$; *(b)* A community $\mathcal{C} = \{A_1, A_2\}$

of $M$ on input $w$ } is the transduction realized by $M$.

- A ***2-DFT*** is a deterministic finite automaton $M$ with a finite control, a 2-way read-only input head and a 1-way output head. The output head can print a word (including empty word) over a finite output alphabet $\Delta$ at each step of the computation. The input head can move left, right or remain stationary based on the current state and the symbol read by the input head except at the end-markers (#, $) where the movement is more restricted. (Thus, on the left (right) end-marker, it can only move right (left), or remain stationary.) The output head always moves left to right (or remains stationary if the output printed at the current step is $\lambda$), and after printing a word on the output tape, it moves to the leftmost blank square to the right of the word it just wrote. The transducer halts when it reaches a configuration $\langle q, a\rangle$ such that $\delta(q, a)$ is not defined where $q, a$ are the current state and the scanned input symbol respectively. Note that a 2-DFT may not halt on all the inputs since it could loop forever on some inputs. The transduction realized by $M$ is $T(M) = \{(w, y) \mid y$ is the output on input $w$ where $M$ halts on $w$ when $M$ starts in its initial configuration [3] }.

Let $T_1$ and $T_2$ be two transductions $T_1 : \Sigma^* \to \Delta^*$, and $T_2 : \Delta^* \to \Gamma^*$. The composition $\oplus$ of transductions $T_1$ and $T_2$ is a relation on $\Sigma^* \to \Gamma^*$ defined in the usual way:
$T_1 \oplus T_2 = \{(w_1, w_2)|$ for some $w_3$, $(w_1, w_3) \in T_1$, $(w_3, w_2) \in T_2\}$.

In this section, we consider a problem that we call the *suffix* problem for regular languages defined as follows. Given a regular language $L$, the *suffix computation* involves the transduction

$T_L = \{$ $(a_1...a_n$ , $b_1...b_n) \mid b_i = 1$ if the suffix $a_i...a_n$ is in $L$, and $b_i = 0$ otherwise. }

The problem we address is the following: For a regular language $L$, can $T_L$ be accepted by a finite-state transducer? The main result of this section is that for any regular language $L$, $T_L$ can be realized by a 2-DFT, but for most of the regular languages $L$, $T_L$ cannot be realized by a 1-DFT. We also show that there is a regular language $L$ for which $T_L$ can be realized by a 2-DFT, but no 2-DFT with a bounded number of head reversals (independent of

[3]For an input $w$, a DFT $M$ is in its initial configuration when *i)* the finite control is in the start state, *ii)* the input tape content is $\#w\$$, *iii)* the input head scanning the left end-marker (#), and *iv)* the output tape is blank).

the input length) can realize it. The results and the techniques presented in this section will be used in the Section 5 to show some results about forward delegation.

**Theorem 1** Let $L$ be $(0+1)^*11(0+1)^*$. $T_L$ can be realized by a 2-DFT, but not by any 1-DFT.

*Proof.* First we describe the 2-DFT that realizes $T_L$ informally. (Although this result is a consequence of the general result we show later, the general result is not intuitive so we present a direct construction here.) On input string $\#w\$$, the 2-DFT $M$ starts scanning the input tape until it sees the first occurrence of a 11 or reaches the right end-marker without seeing a 11. Latter case is easy to deal with: In this case, the output is $0^n$ where $n = |w|$. $M$ generates the output 0 on a reverse sweep for each input symbol until it reaches the left end-marker and halts. In the former case, it moves its input head until it reaches the left end-marker and for each input symbol scanned it writes a 1 on the output tape. Now it moves the input head to the right until it reaches the first occurrence of 11, continues until it sees a second set of consecutive 1's or it reaches the right end-marker without seeing another 11. In the latter case, it reverses the input head and writes a 0 on the output tape for each input symbol scanned until it reaches the previous 11 and halts. In the former case, it reverses the input head and writes a 1 on the output tape for each input symbol scanned until it reaches the previous 11. Now it repeats the process. The correctness of this construction is obvious from this informal description.

Next we show that there is no 1-DFT that realizes $T_L$. Suppose such a transducer $M$ exists. Consider the sequence of states $q_0$, $q_1$, ..., where $q_{i+1}$ is the state reached from $q_i$ on input 0. Let $j$ be the smallest integer such that $q_j = q_i$ for some $i < j$. Let the outputs on the transitions $\delta(q_k, 0)$ be $T_k$ for all $k$ = 1, ..., $j$. If all the $T_k$'s are $\epsilon$ for $k$ = 1, 2, ..., $j$, then it is obvious that $M$ produces the same output on $0^i\$$ and $0^{2j-i}\$$, which is clearly wrong. Thus, $T_k \neq \epsilon$ for some $k$. This means that $M$ produces an output while processing a prefix of 0's. Suppose the output produced includes a 1. Then, on input $0^j\$$, it produces an output string one bit of which is 1, a contradiction. Similarly, if the output produced includes a 0, the input $0^r11\$$ (for some $r \geq j$) would produce a contradiction.

This concludes the proof. □

In fact, it seems that no 2-DFT with bounded reversals can realize $T_L$ for the above language $L$. However, it can be seen that there is a 1-NFT that realizes $T_L$.

Next we consider a regular language $L$ for which there is no 1-DFT but there is a 2-DFT that makes only two reversals on the input tape to realize $T_L$.

**Theorem 2** Let $zero(x)$ denote the number of 0's in a binary string $x$. Consider the well-known parity language $L = \{x \mid zero(x) \text{ is odd}\}$. $T_L$ can be realized by a 2-DFT that reverses the input head only twice, but not by a 1-DFT.

*Proof.* We describe informally the 2-DFT $M$. On input $\#x\$$, $M$ makes an initial pass over input string $x$ determining if the parity of $zero(x)$ is odd or even. During the second pass, it uses this information to determine the parity of each suffix. In fact, the first output bit $b_1$ = 1 (0) if the parity of $zero(x)$ is odd (even). If the output $b_i$ is 1, then $M$ uses the following simple rule to determine the output $b_{i+1}$. (It need only the remember the previous output bit

in order to implement this computation.) $b_{i+1} = 1$ if the next input is 0, else it is complement of $b_i$.

The following is a simple proof that $L$ can't be realized by a 1-DFT: $L$ has the property that for any $x \in \{0,1\}^*$, there exist $y$ and $z$ such that $xy \in L$, and $xz \notin L$. From this it follows that the 1-DFT can never outputs its first bit until it reads the entire input string. From this fact, a contradiction can be readily obtained. □

Next we show that, for any regular language $L$, the transduction $T_L$ can be realized by a 2-DFT. In order to show this result, we need the following lemma from Chytil and Jakl [13].

**Lemma 1** *The class of transductions computed by 2-DFT is closed under composition.*

We use the above lemma to show the following:

**Theorem 3** For any regular language $L$, there is a 2-DFT that realizes $T_L$.

*Proof.* Let $T'_L = \{(x_1x_2...x_n, y_ny_{n-1}...y_1) \mid y_i = 1$ (0) if and only if $x_ix_{i+1}...x_n$ is (not) in $L$ $\}$. We will show that $T'_L$ can be realized by a 2-DFT $M_L$ as follows. Let $M$ be a DFA that accepts $L$. Create an NFA $M'$ by reversing the transitions of $M$, and convert the NFA $M'$ into a DFA $M_1$ using the subset construction. The set $F$ of accepting states of $M$ will be the starting state of $M_1$. The DFT $M_L$, on input $\#x_1x_2...x_n\$$, moves its head to the right until it reaches the right end-marker. Then, it starts reading the input from right to left, and maintains in finite control a state of $M_1$. Initially, this state is $F$. After reading each input, it updates the state by simulating $M_1$ from the maintained state on the input scanned. If this current state contains $q_0$, then it writes 1 on the output tape, else it writes 0 on the output tape. It repeats this until the input head reaches the left end-marker. Then it halts. It is easy to see that $M_L$ realizes $T'_L$.

Now define the transduction $T_1$ as: $T_1 = \{(x, x^{rev}) | x \in \{1, 2, ..., r\}^*\}$. It is clear that $T_1$ can be realized by a 2-DFT that simply moves the head to the right until it reaches the right end-marker, then reads the input tape from right to left and copies the symbols read on the output tape. It is obvious that $T_L = T'_L \oplus T_1$. By the above lemma, there exists a 2-DFT $M'$ that realizes $T_L$ since it is the composition of transductions realized by two 2-DFT's. □

It would be interesting to characterize the regular languages for which the suffix problem can be realized by a bounded reversal DFT.

## 5. Computing a Service Delegation

Our aim here is to propose an algorithm for the service delegation problem and to make a formal analysis of the algorithm. To do that, first, we need a model of the machinery computing a delegation. We call this machinery a *delegator*.

### *5.1. Delegator Model*

We model a delegator as a *deterministic finite-state transducer* (DFT). There are some advantages of designing a delegator as a finite-state transducer. In addition to the fact that such

delegators work in linear time, they also have other desirable properties such as testability for conformance to specification.

Next, we ask a natural question whether a 1-DFT or a 2-DFT is more appropriate to model a delegator. Essentially, a 1-DFT delegator should be able to compute a delegation "online" (or "real time"), i.e., it makes one pass on the given sequence of activities and as it sees the activity, it assigns the activity to a service schema. In other words, the delegator doesn't have to know the complete sequence in order to start the delegation. The following result, however, shows that in general a 1-DFT is not strong enough to model a delegator and the delegation computation cannot be done "online" in general.

**Theorem 4** There exists a community of services $\mathcal{C} = (A_1, ..., A_r)$ such that there is no 1-DFT delegator that computes a service delegation $w'$ for every word $w$ being composable with respect to $\mathcal{C}$.

*Proof.* Consider the community $\mathcal{C} = (A_1, A_2)$ of services in Figure 2.(b). We claim that there is no 1-DFT $M$ that can compute a delegation for any word composable with respect to $\mathcal{C}$. Suppose that such a 1-DFT $M$ exists. Assume $M$ has $k$ states. Consider an input $x = p^n m$, where $n >> k$ (we do not show the end markers that go with the input as required by the definition of 1-DFT). By definition of a 1-DFT, the transducer need not write on the output tape at every step. However, it is clear that $M$ has to write a non-blank word after reading at most $k$ $p$'s on the input; otherwise we can pump $i$ $p$'s to $x$ for some $i$ and obtain a word $y = p^{n+i}m$, and the output for $y$ will be the same as the output for $x$. This is a contradiction, since the output for $y$ must be longer by $i$ bits. Hence, on input $x$, $M$ outputs a word starting, let's say, with 1 (since $x$ ends with an $m$) before processing all of the $p$'s. Now since $M$ is deterministic, it will also output 1 as the first bit for the input $x' = p^n c$. But this is wrong, since $x'$ ends with a $c$ and the first bit to be output should be a 2. It follows that $M$ does not exist. □

Our next results show that 2-DFT delegators are capable enough to solve service delegation problem.

### *5.2. Delegation Algorithm*

Here we show that there exists a delegator which computes a service delegation for an input of sequence of activities in time linear in the length of the input. Essentially, the delegator works on the input tape, makes bidirectional moves on the input tape and uses its finite number of states to decide the assignment of each activity. Note that this style of computation is not "real-time" because the delegator doesn't decide the fate of an activity as it sees the activity; instead, the decision happens after examining a window of activities.

To prove the existence of such a delegator, we first construct a 2-DFT which computes the *reverse* of the service delegation. Note that the delegation is a word as defined before and the reverse of a word $x_1x_2 \ldots x_{n-1}x_n$ is defined as $x_nx_{n-1} \ldots x_2x_1$ in the standard manner. This 2-DFT makes a right sweep followed by left sweep and outputs one delegation on each word.

**Theorem 5** There exists a 2-DFT that computes the reverse of a delegation of a word $w$ of length $n$ on a community $\mathcal{C}$ in $2n$ steps.

*Proof.* Let $S = (A_T; A_1, ..., A_r)$ and $\Sigma$ be the common input alphabet of the NFAs. Without loss of generality, assume that the language accepted by each machine does not contain the null word $\epsilon$. Construct the product NFA $B$ of the system $S$ whose states are $r+1$-tuples $\langle q, q_1, q_2, ..., q_r\rangle$ where $q$ is a state of $A$ and $q_i$ is the state of $A_i$ and the transitions are defined as follows: $\langle q, q_1, q_2, ..., q_r\rangle \in \delta(\langle p, p_1, p_2, ..., p_r\rangle, a)$ whenever $q \in \delta_T(p, a)$ and there is a $j$ such that $q_j \in \delta_j(p_j, a)$ and $p_k = q_k$ for all $k \neq j$. (Note that this product construction is different from the standard product construction used for the intersection.) We introduce a new symbol $\&$ and create a new state in the product machine $B$ and direct all accepting states to the new state via input $\&$. We make the new state the only (unique) accepting state. Then the language accepted by the modified product machine is a subset of $\Sigma^+\&$. For notational convenience, we also call this modified machine $B$ and continue to refer to it as a product NFA (or machine).

Let $B_d$ be the DFA constructed from $B$ using the subset construction [9]. Let $q_1, ..., q_k$ be the states of $B$, where $q_1$ is the initial state and $q_k$ is the unique accepting state. Note that each state $q_i$ is an $(r+1)$-tuple representing the states of $A_T, A_1, ..., A_r$. For every $q_i$ $(1 \leq i \leq k)$, let $B_{q_i}$ be the NFA $B$ with start state $q_i$. Note that each such NFA has the same accepting state $q_k$. For every $q_i$ $(1 \leq i \leq k)$, let $B^r_{q_i}$ be the NFA derived from $B_{q_i}$ by reversing the directions of the edges, and making $q_k$ the start state and $q_i$ the accepting state. Thus $B^r_{q_i}$ accepts the reverse of the language accepted by $B_{q_i}$.

For each $q_i$ $(1 \leq i \leq k)$, we use the subset construction to construct the DFA $C_{q_i}$ equivalent to $B^r_{q_i}$. Note that the start state of $C_{q_i}$ is the singleton subset $\{q_k\}$ and every accepting state (which is a set of states) contains the state $q_i$. Now we construct a 2DFA $D$ which, when given $\#w\$ = \#a_1...a_n\$$ (where $\#, \$$ are distinguished input delimiters or endmarkers), will output the delegation of $a_n, ..., a_1$ (in this order).

$D$ has in its finite control the specification (i.e., the transition function) of each $C_{q_i}$. $D$ moves its input head from the left endmarker to the right endmarker and simulates $B_d$ and computes the state reached by $B_d$ after reading $w$. Note that the state is a set of $(r+1)$-tuples. If the state reached is not accepting, then $D$ outputs 'invalid' and halts; if the state reached is accepting but there is no tuple in the state for which all components are accepting, then $D$ outputs 'error and halts; else $D$ goes to the next step.

$D$ now moves left and for each symbol $a_{n-i}$ $(i = 0, ..., n-1)$, computes as follows: $D$ maintains in its finite control a vector of the form $\langle P_1, ..., P_k\rangle$, where each $P_i$ is a state of $C_{q_i}$, which is a subset of a state of $B_{q_i}$. Initially, the vector is $\langle\{q_k\}, ..., \{q_k\}\rangle$. Note that $\{q_k\}$ is the initial state of each $C_{q_i}$.

$D$ reads $a_{n-i}$ and computes from $\langle P_1, ..., P_k\rangle$ the vector $\langle P'_1, ..., P'_k\rangle$, where $P'_j$ is the next state of $C_{q_j}$ from $P_j$ on input (i.e., after reading) $a_{n-i}$. Also, if $P'_1$ contains $\{q_{t_1}, ..., q_{t_s}\}$ where $t_1 < ... < t_s$, $D$ picks the first $t_m$ such that $P'_{t_m}$ is an accepting state of $C_{q_{t_m}}$, and then outputs $1 \leq l \leq r$ that corresponds to the delegation to $A_l$ that resulted in state $q_{t_m}$.

It is clear that $D$ is a 2-DFT that performs its computation in exactly $2n$ steps and solves the *backward delegation* problem. This completes the proof. □

Next, we prove the existence of a linear-time delegator for the delegation problem.

**Theorem 6** There exists a delegator that computes a delegation of a word $w$ of length $n$ on a community $\mathcal{C}$ in $O(n)$ steps.

*Proof.* Let $T(D)$ be the transduction associated with the reverse delegation problem. We have shown in the previous theorem that $D$ can be realized by a 2-DFT. Define the transduction $T_1$ as: $T_1 = \{(x, x^R) | x \in \{1, 2, ..., r\}^*$and $x^R$ is the reverse of $x\}$. It is clear that $T_1$ can be realized by a 2-DFT that simply moves the head to the right until it reaches the right end-marker, then reads the input tape from right to left and copies the symbols read on the output tape. Clearly, the composition $T' = T(D) \oplus T_1$ gives the solution to the delegation problem (first use $T_1$ and then $T(D)$). By Lemma 1 of Section 4 from Chytil and Jakl [13], the class of transductions computed by a 2-DFT is closed under composition. Therefore, there exists a 2-DFT $D'$ that realizes $T'$. It follows from the construction of the above lemma that $D'$ halts on all inputs. Thus the total number of time steps on any input length $n$ must be $O(n)$ since the number of visits on any tape square of the input tape can't be visited more than a constant number of times without getting into an infinite loop [9]. $\square$

The delegator described above goes back and forth on the input tape, and decides the delegation of the given activities. In general, the number of reversals (of direction of the input head) can be unbounded (i.e., it can grow with the length of the input). Next, we study the question of whether it is possible to construct a delegator when there is a bound on the number of reversals the delegator can make (a reversal happens when the input head changes its direction from forward to backward or vice versa). We prove the following result:

**Theorem 7** There exists a community $\mathcal{C} = (A_1, ..., A_r)$ of services such that there is no 2-DFT delegator $M$ whose input head reverses a bounded number of times that realizes the forward delegation for any composable word.

*Proof.* Consider the community $\mathcal{C} = \{A_1, A_2\}$ in Figure 2.(b). Let $M$ be a 2-DFT with $s$ states whose input head reverses at most $k$ times for some $k$ that is independent of the input length $n$. Let the input be $x = p^{n_1} q_1 p^{n_2} q_2 ... p^{n_m} q_m$, where $m$ is much greater than $k$ ($k$ is the number of states of $M$), $n_1, ..., n_m$ are much greater than $s$, and $q_i \in \{m, c\}$ for $1 \leq i \leq m$. We ignore the right end-marker since it is not relevant to the proof. We assume (as is customary) that $M$ starts with its head reading the leftmost symbol of its input and that when it halts, its head is reading the right end-marker. We call the subword $p^{n_j} q_j$ as a *block*. Associated with each block $B$ is the number $n(B)$ = total number of reversals performed by the input head while the input head is in that block. Since the number of blocks $m$ is much larger than $k$, there is at least one block $B_j = p^{n_j} q_j$ (for some specific $j$) with the property that the input head of $M$ never reverses while scanning any input symbol of this block. Let $t$ be the first time at which the input head enters the block $B_j$ and let $t'$ be the first time step at which the input head is reading the symbol $q_j$ of block $B_j$. Since $t' - t \geq n_j$ is much greater than $s$, it is clear that during the time between $t$ and $t'$, $M$ must write a non-null word on the output tape. If not, as in Proof 5.1, we can produce two words of different lengths for which $M$'s output words will have the same length. Suppose a symbol printed by $M$ during this time is 1, then by changing $q_j$ (from $m$ to $c$, or vice-versa), $M$ would still produce the same 1 on the output tape, which is an erroneous output for this new input. This completes the proof. $\square$

Next we consider the constrained delegation case.

## 6. Constrained Delegation

In this section, we first define a fairly general class of constraints, namely Presburger formulas, and we show that when a Presburger formula is enforced on a composite service, delegations can be computed in polynomial time in the length of the input.

Let $X = \{x_1, ..., x_r\}$ be a finite set of nonnegative integer variables. An *atomic Presburger relation* on $X$ is either an atomic linear relation

$$\Sigma_{x \in X} a_x x < b,$$

or a mod constraint $x \equiv_d c$, where $a_x, b, c$ and $d$ are integers with $0 \leq c < d$. A Presburger formula can always be constructed from atomic Presburger relations using $\neg$ and $\wedge$. Note that a Presburger formula $F(x_1, ..., x_r)$ defines a relation $P \subseteq N^r$, i.e., the relation $P = \{(k_1, ..., k_r) \mid F(k_1, ..., k_r)$ is true $\}$. Note also that in our definition, Presburger formulas are quantifier-free (i.e., there are no existential and universal quantifiers).

The following lemma is well-known:

**Lemma 2** *Let $R = \{F\#k_1\#...\#k_r \mid r \geq 1, F$ is a Presburger formula over $r$ variables, $x_1, ..., x_r$ are nonnegative integers, and $F(k_1, .., k_r)$ is true $\}$. There is an algorithm (deterministic Turing machine) to accept $R$ in time polynomial in $(m + n)$, where $m$ is the length of the binary representation of $F$ and $n$ is the length of $k_1\#...\#k_r$, with each $k_i$ represented in binary.*

Let $\mathcal{C} = (A_1, ..., A_r)$ be a service community. Suppose we want to restrict the delegation of symbols in $w$ such that if $w_i$ is the subsequence delegated to $A_i$, then not only is $w_i$ accepted by $A_i$ for $1 \leq i \leq r$, but $(k_1, ..., k_r)$ must also be in a specified relation $P$, where $k_i$ is the number of symbols in $w_i$. So, e.g., if $r = 3$, $P$ might be the set of triples $(k_1, k_2, k_3)$ such that $k_1 \leq k_2 + 2k_3$ and $k_2 > k_3 + 5$.

We can prove the following result. Note that $F$ represents a Presburger formula over $r$ variables.

**Theorem 8** Let $\mathcal{C} = (A_1, ..., A_r)$ be a service community. We can construct a two-way deterministic Turing machine transducer (DTMT) $M$ (i.e., $M$ is a 2-DFT augmented with read/write worktapes) which, when given a two-way read-only input $F\#a_1...a_n$, outputs a delegation of all the symbols in $w = a_1....a_n$ (to the $A_i$'s) such that if $k_i$ is the number of symbols of $w$ delegated to $A_i$, then $F(k_1, ..., k_r)$ is true. Moreover, $M$ runs in time polynomial in $L$, where $L$ is the length of $F\#a_1...a_n$.

*Proof.* The main idea behind the proof is to use "breadth-first search" and Lemma 2. The overall idea is as follows: construct a tree recording all possible ways to delegate the given input word. During the construction, at each level of the tree, make the duplicate configurations collapse to one. Each configuration consists of the state and a number of counters (which can contain quality attributes like price, reputation etc., or they can be, for instance, the number of a's assigned to service A1 and the number of b's assigned to service A2 ); therefore, there

is a polynomial number of configurations totally. At the end of the construction, for each configuration, run the given constraint specific Turing machine. Since there is a polynomial number of configurations and to check one formula takes polynomial time, the total time is polynomial in the size of the input word, and linear in the size of constraints and the number of FSMs. □

We can generalize the theorem above to NFAs with *storage*. More specifically, each service is modeled as an NFA augmented with polynomial-bounded counters (i.e., an NFA having a finite number of counters, each which can have value at most $p(n)$ for some given polynomial $p$). We will show that for a set of services under this model, the constrained delegation problem is solvable in polynomial time.

A polynomial-bounded counter machine (or simply, linear NCM) is an NFA 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 are constants $c, k$ such that on any input of length $n$, the value stored in each counter during the computation is at most $p(n) = cn^k$; thus, such counters are called *polynomial-bounded* counters). Note that in automata theory, a counter is 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 ask about the value of the counter other than testing whether it is zero or not, i.e., whether the stack is empty or not.

Our condition on the counters being polynomial-bounded can be strengthened to counters with linear values. This is because a counter that has value bounded by $cn^k$ (for some $k$) can be simulated by $k$ linear counters without loss of time.

Clearly, the proof of the above theorem still holds, even if each $A_i$ has polynomial-bounded counters, since the total number of configurations is still polynomial. Hence, we have:

**Theorem 9** Let $\mathcal{C} = (A_1, ..., A_r)$ be a service community, where each $A_i$ an NCM with polynomial-bounded counters. We can construct a DTMT $M$ which, when given a two-way read-only input $F\#a_1...a_n$, outputs a delegation of all the symbols in $w = a_1....a_n$ (to the $A_i$'s) such that if $k_i$ is the number of symbols of $w$ delegated to $A_i$, then $F(k_1, ..., k_r)$ is true. Moreover, $M$ runs in time polynomial in $L$, where $L$ is the length of $F\#a_1...a_n$.

The Presburger constraints on the delegation can be generalized in various ways. For example, suppose that the community has $r$ services each has $O(k)$ transitions for some number $k$. Also, suppose that the quality vector of each transition contains execution costs. In this setting, Let's say a user wants to specify that the cost of a delegation should be less than some value. Then, a Presburger formula can be formed over $O(kr)$ variables where each variable represents the number of times the corresponding transition is taken during the delegation. More specifically the formula will be "the sum of the variables times the transition costs being less than the user desired cost value".

## 7. Ambiguous Composability

In this section, we study the multiplicity of delegations. We describe the problem and the results in a more abstract language theory framework. The results can easily be tailored to the service delegation problem we discussed in the previous sections.

We define the notion of $k$-ambiguous composability as follows: Given a system $S = (A; A_1, ..., A_r)$ of NFAs, we say that $S$ is unambiguously composable if there is a unique delegation for every string in $L(A)$. More generally, we say that $S$ is $k$-ambiguously composable if there are at most $k$ ways to delegate any string in $L(A)$. If there is no $k$ such that $S$ is $k$-ambiguously composable, then we say that $S$ has unbounded composability.

We can show the following results regarding ambiguous composability.

**Theorem 10** Given a system $S = (A; A_1, ..., A_r)$ of NFAs and a fixed $k$, it can be decided if $S$ is $k$-ambiguously composable.

*Proof.* Create an NFA $M$ that guesses simultaneously $k+1$ distinct compositions and verifies that each of them is a valid decomposition. More precisely, define the language $L = \{[a_1, b_{1,1}..., b_{1,k+1}]\ [a_2, b_{2,1}..., b_{2,k+1}]\ ...\ [a_n, b_{n,1}..., b_{1,k+1}]$ — $b_{j,1}...b_{j,n}$ is a delegation of $a_1...a_n$ for each $j$ = 1, ..., $k+1\}$. An NFA $M$ can be designed to accept this language. $M$ guesses $k+1$ distinct delegations and verify by keeping track of all the k+1 sets of $r+1$-tuples of states and accept if and only if each of the guesses is a valid delegation. Now, $S$ is $k$-ambiguously decomposable if and only if $L$ is empty. The claim follows from the fact that emptiness of a regular language is decidable. □

We can also show that, given a system $S = (A; A_1, ..., A_r)$ of NFAs, it can be decided if $S$ has unbounded composability.

We need the following result from [11]. Let $M$ be a 1-NFT (nondeterministic finite state transducer). We say that $M$ is finite valued if there is an integer $k > 0$ such that for any string $w$ accepted by $M$, there are at most $k$ outputs produced by $M$. We have the following result from [11] that characterizes NFT's that are not finite valued.

**Theorem 11** Let $M$ be a NFA with $n$ states in which all the useless states have been removed. $M$ is of unbounded ambiguity if one of the following is true:

1. There exists a state $q$ and a string $w$ (of length at most $n$) such that there are two paths labeled $w$ that take $q$ to itself producing two different outputs.
2. There exist two states $p$ and $q$, and a string $w$ of length at most $n$ such that $p \in \delta(p, w)$, $q \in \delta(p, w)$ and $q \in \delta(q, w)$ and the outputs produced on the loops (around $p$ and $q$) are different.

We can use the above theorem to show the following:

**Theorem 12** Given a system $S = (A; A_1, ..., A_r)$ of NFAs, it can be decided *in polynomial time* if $S$ is $k$-ambiguously composable for some $k$.

*Proof.* (sketch) It was shown in [11] based on the above theorem that finite valuedness of a NFT is decidable and this result was strengthened in [15] where a polynomial time algorithm was presented for this problem. Given the system $S = (A; A_1, ..., A_r)$ of NFAs, we define a 1-NFT $M_S$ that is similar to the NFA of Theorem 3.1, except that it also outputs the ID number of the NFA that it currently simulates. It is clear that this NFT will output all possible delegations when given a composable string as input. It is also obvious that $S$ is $k$-ambiguously composable if and only if $M_S$ is finite-valued. Now we can apply the algorithm of [15] to determine if this NFT is finite valued. This concludes the proof. □

Finally, we consider the problem of determining the equivalence of delegators. We say that two unambiguous delegators $S_1 = (A; A_1, ..., A_r)$ and $S_2 = (B; B_1, ..., B_r)$ are equivalent if there is a bijective mapping $\pi$ between $A_i$'s and $B_i$'s such that on any input $w \in L(A)$, if $w$ is composable and if the (unique) delegation for $w$ in $S_1$ is $b_1...b_n$, then $w$ is also composable in $S_2$ and the delegation for $w$ in $S_2$ is $\pi(b_1...b_n)$.

We can show

**Theorem 13** It is decidable to determine whether the delegators for two unambiguous delegators $S_1 = (A; A_1, ..., A_r)$ and $S_2 = (B; B_1, ..., B_r)$ are equivalent.

*Proof.* Recall the 2-DFT that was used for the reverse delegation in Section 3. We build the DFT's for $S_1$ and $S_2$. Call them $M_1$ and $M_2$. The delegators for $S_1$ and $S_2$ are equivalent if and only if $M_1$ and $M_2$ are equivalent. The decidability now follows from the fact that equivalence problem for single valued 2-DFT is decidable [8]. □

## 8. Conclusion

Automated Web service composition is a central problem in the area of Web services in order to achieve flexibility and scalability. The delegation problem that forms the central theme of this work involves a user request for service that can be delegated to a set of services that forms a service community so that the composition of these services meets the user's request. We believe that our constructions provide a foundation for efficient implementation of tools for the delegation problem and we hope that the ideas presented here would lead to the development of such software tools.

## 9. Acknowledgements

We would like to thank the reviewers for their careful study of the manuscript and their constructive suggestions for ensuring its accuracy and comprehensiveness.

## References

[1] B. Benatallah, M. Dumas, Q.Z. 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.

[2] D. Berardi, D. Calvanese, G. De 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.

[3] D. Berardi, G. De 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.

[4] Business Process Execution Language for Web Services (BPEL), Version 1.1. `http://www.ibm.com/developerworks/library/ws-bpel`, May 2003.

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

[6] 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.

[7] C. E. Gerede, O. H. Ibarra, B. Ravikumar, and J. Su. Online and minimum-cost ad hoc delegation in e-service composition. In *Proceedings of IEEE International Conference on Services Computing (SCC)*, 2005.

[8] E. Gurari. The equivalence problem for deterministic two-way sequential transducers is decidable. In *Proceedings of the Twenty-First Annual IEEE Symposium on Foundations of Computer Science*, pages 83–85, 1980.

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

[10] 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.

[11] O. H. Ibarra and B. Ravikumar. On sparseness, ambiguity and other decision problems for acceptors and transducers. In *STACS*, Lecture Notes in Computer Science, pages 171–179. Springer, 1986.

[12] 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.

[13] M.P.Chytil and V. Jakl. Serial composition of 2-way finite-state transducers and simple programs on strings. In *ICALP*, volume 52 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 1977.

[14] Simple Object Access Protocol (SOAP) 1.1. W3C Note 08, May 2000. `http://www.w3.org/TR/SOAP/`.

[15] A. Weber and H. Seidl. On the degree of ambiguity of finite automata. In *MFCS*, Lecture Notes in Computer Science, pages 620–629. Springer, 1986.

[16] Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.

[17] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004.