Automated Composition of E-services: Lookaheads

Çağdaş Evren Gerede[*]
Department of Computer Science,
University of California,
Santa Barbara, CA 93106, USA.

Richard Hull
Bell Laboratories/Lucent Technologies,
700 Mountain Avenue,
Murray Hill, NJ 07974, USA.

Oscar H. Ibarra [†] Jianwen Su
Department of Computer Science,
University of California,
Santa Barbara, CA 93106, USA.

ABSTRACT

The e-services paradigm promises to enable rich, flexible, and dynamic inter-operation of highly distributed, heterogeneous network-enabled services. Among the challenges, a fundamental question concerns the design and analysis of composite e-services. This paper proposes techniques towards automated design of composite e-services. We consider the Roman model which represents e-services as activity-based finite state automata. For a given set of existing e-services and a desired e-service, does there exist a "mediator" which delegates activities in the desired e-service to existing e-services? The question was raised in an early study by Berardi et. al. for a restricted subclass of delegators which does not take into consideration of future activities. In this paper, we define a more general class of delegators called "lookahead" delegators and we show that the hierarchy based on the amount of lookahead is strict. We, then, study the complexity of constructing such delegators. We prove that in the case of deterministic e-services, a k-lookahead delegator can be computed in time polynomial in the size of target and subcontractor e-services, and exponential in k and the number of subcontractor e-services. We also present *W*ozart, an automated mediator construction tool implemented to realize our approaches.

Categories and Subject Descriptors

H.1.m [**Models and Principles**]: Miscellaneous; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State diagrams*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and Scheduling*

General Terms

Algorithms, Design, Theory

Keywords

E-services, Automated Composition, E-service Modelling, Service Oriented Computing, Service Representation, Service Composition, Roman Model, Automated Mediator Construction, Lookahead, Delegator, Finite State Automata

[*]Contact Author: `gerede@cs.ucsb.edu`

[†]Research supported in part by NSF Grants CCR-0208595 and CCF-0430945.

1. INTRODUCTION

The *e-services* paradigm promises to enable rich, flexible, and dynamic inter-operation of highly distributed, heterogeneous network-enabled services. Emerging standards such as SOAP [23], WSDL [25], BPEL [4], and research efforts that are building on or taking advantage of the paradigm such as the OWL-S (formerly DAML-S) program [19], the Semantic eWallets project [13], ActiveXML [2], have made a substantial progress toward this goal. However, given the ostensible long-term goal of enabling the automated discovery, composition, enactment, and monitoring of collections of e-services (we also use *web services* interchangeably) working to achieve a specified objective, key pieces are missing to make the goal a reality. Among the challenges, a fundamental question concerns the design and analysis of composite e-services. The focus of this paper is on automated design of composite e-services.

Automated composition was studied in [24, 16] in the context of workflows. In [24], the global dependencies are given as a tree with optional and choices on some dependencies, resembling the event algebra [22]. 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, [16] starts from a pair of pre- and post-conditions and assembles the workflow by selecting tasks from a given library. The synthesis problem for finite state specifications has been studied intensely within the automata theory and verification community [5, 1]. Consider synthesis of a collection of finite automata interacting via bounded queues. The synthesis problem has a variant for open and closed systems. In the closed case, a "folkloric" result is that synthesis from a formula can be decided by linear reduction to the satisfiability test for the logic hence it can be done in PSPACE for LTL and in PTIME for ω-regular sets represented explicitly by an automaton. The open case is undecidable [20] in general, but decidable when e-services are connected in a linear topology [15].

Automated e-service composition has been the focus of several recent studies. One approach was developed in the context of OWL-S [18]. The basic question there 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. Their approach is to encode the underlying situation calculus world view, the desired goal, the individual services in terms of 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. In another approach [6, 10, 11], the desired global behavior is a "conversation" (i.e., family of permitted message sequences) specified using a finite state automaton. Under certain conditions, the automaton can be "projected" to build (abstract) web services, that when combined will realize the desired conversation.

Recently, [3] developed a very interesting approach to automated composition of web services, called here the *Roman* model. The model focuses on activity-based finite state automata. One input to this approach is a set of descriptions of existing web services (called here "subcontractors"), each given as an automaton that describes possible sequences of activities. (This resembles web services residing in a UDDI repository). The second input is a desired behavior or "target" behavior, also specified as an automaton. The output is a subset of the atomic web services, and a "mediator" that will coordinate the activities of those services, through a form of "delegation".

Based on the Roman model introduced in [3], this paper makes the following contributions:

1. A general notion of delegation is developed. Specifically, a delegator in the model of [3] assigns an activity to atomic services based only on the past activities. In our model, a delegator can determine the assignment based on an entire sequence of activities.

2. We show that deciding the existence of a mediator in the general model can be done in EXPSPACE, the result generalizes the result in [3].

3. We introduce the notion of "lookahead" for a mediator to delegate activities, which is the number of future activities the mediator has to know in order to delegate an activity. The delegator notion in [3] corresponds to having no lookahead or lookahead being 0. In particular, we give an alternative constructive proof using automata-theoretic techniques for determining whether a delegator without lookahead exists, originally shown in [3]. A benefit of the automata-theoretic approach is a finer characterization of the complexity bound in terms of the size and the number of individual e-services.

4. We study the impact of lookahead on automated design of mediators. We show that there exists a strict hierarchy based on lookahead.

5. We show that in the case of deterministic e-services, a k-lookahead delegator can be constructed in time polynomial in the size of the target and subcontractor e-services, and exponential in k and the number of subcontractor e-services.

6. Finally, we present a tool for automated mediator construction named *W*ozart.

The paper is organized as follows. Section 2 introduces the model of e-services and the central notion of a delegator. Section 3 focuses on determining the existence of delegator. Lookahead is defined in Section 4, along with results and algorithms concerning lookahead. Section 5 talks about the tool *W*ozart and Section 6 concludes the paper.

2. A MODEL FOR E-SERVICES AND COMPOSITIONS

In this section we outline the key concepts for the technical discussions, which include e-services, and "composition" of e-services.

Roughly speaking, an e-service is a (publishable) specification of a software program. Although data strucures and types are fundamental in such specifications, the more important (and difficult) aspect concerns the "behavioral signatures" of e-services, since the latter captures the "actions" and "reactions" an e-service can engage in during the execuion. Behavioral signatures provides the foundation for composing e-services.

A behavior model for individual and composite e-services has fundamental implications on how they can be discovered, combined, and analyzed. It is not surprising that several paradigms for behavioral modeling are being used in current standards and research explorations on web services composition.

In a broad term, the behavior of a service describes the changes of its "states". Depending on specific research topics, the "state" can be (a) the actual internal execution state, (b) only a part of the state of relevance to the parties connected with the e-service, or (c) the state of the "external world". Furthermore, different models rely on different kinds of "actions" to change state; these might be (i) messages, (ii) activities, and (iii) events.

The WSDL standard focuses heavily on passing messages between e-services. This leads naturally to behavioral models that use messages as the "actions", and use (abstract) internal states of individual e-services as the states that messages cause transitions between [6, 10]. This perspective is closely related to work on process algebras [17] and in the verification community [7], all of which study distributed automata with message passing of one form or another. It can also support investigations based on partial information of the internal states, as typical in the verification community.

The workflow community has traditionally focused on activity-based models. These represent a process by combining activities with essentially some forms of control flow. The typical formalisms in workflow community are flowcharts, Petri nets, and finite state machines or state charts.

The semantic web services community also favors an activity-based perspective. Much of that work assumes that atomic services perform activities, which have the effect of changing the state of an "external world". A situation calculus [21] is typically used to provide formal underpinnings. This framework permits the use of logic-based axiomatizations and reasoning about how composite web services affect their "external" world, and thereby permit the use of goal-based planning algorithms for automated construction of compositions.

Event-based formalisms have been used primarily in the context of workflows [22]. An event can be viewed as an abstract version of an activity. Event-based models allow declarative, logic based semantics and provide an alternative to analysis of workflow specifications [9].

In this paper, we consider the Roman model which represents e-services as activity-based finite state automata. The model was introduced in [3] in their study on automated composition.

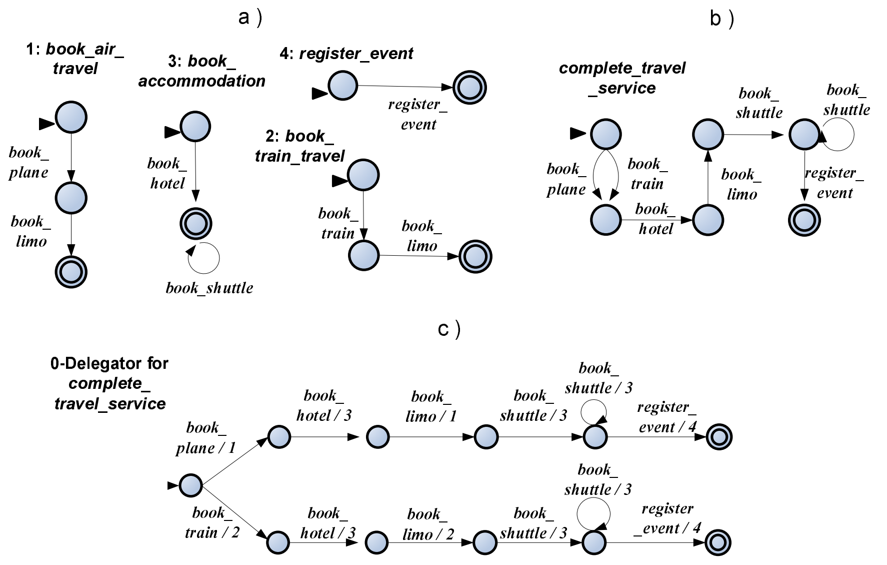


Figure 1: A Composition Problem: Complete Travel Service

Before formulating our model of composition and e-services, let's have a look at an example illustrating the composition problem.

EXAMPLE 2.1. [1] Suppose that the services in Figure 1(a) are available in some UDDI++ directory, which includes finite state automata-based service descriptions, among other things. Each service has different functionalities. For example, the user can use:
i) *book_air_travel* service to book a plane for a trip and then book a limo to the airport, ii) *book_accommodation* to arrange for accommodation and iii) *register_event* to register for the event. Note that booking a limo service is provided by both *book_air_travel* and *book_train_travel* services but the overall behaviors of these two services are different. Suppose further that a 3rd-party vendor would like to create and operate a composite service which is specified again as a finite state automaton shown in Figure 1(b). Here, we are interested in the question of whether it is possible to reuse the existing services in order to achieve the desired e-service. If that is possible, then effectively there must be a mediator who orchestrates the services collectively in order to model the desired e-service. For example, Figure 1(c) shows such a delegator that assigns each activity to an existing e-service. Note that as the example shows, a delegator may not simply be a labeling of the desired e-service. ∎

2.1 Preliminaries

We assume some familiarity with formal languages and finite state automata. Let Σ be a finite alphabet of activities (or symbols). A *word of length* $k \in \mathbb{N}$ over Σ is a sequence of k symbols in Σ. Let Σ^* be the set of all words over Σ of length k for some $k \in \mathbb{N}$. A *language* is a subset of Σ^*.

We now define the central notion of an e-service.

DEFINITION 2.2. An *e-service* is a (possibly nondeterministic) finite state automaton (FA) $A = (S, \Sigma, \delta, s^0, F)$ where S is the finite set of *states*, Σ is the input (activity) alphabet, δ 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.

[1]This example is inspired from one developed by Daniela Berardi.

Intuitively, an e-service A is viewed as an acceptor. The notion of a word *accepted* by A is defined in the standard manner. The *(activity) language* of an e-service A, denoted as $L(A)$, is the set of words accepted by A.

In this model, FAs represent both "atomic" as well as composite e-services. In other words, a composition of a set of e-services is expected to be another e-service [3]. Although FAs, therefore e-services, are closed under composition (under various product constructions), the goal of this paper aims at determining whether a desired e-service can be composed from a set of existing e-services.

We now formulate the notions of composition model and delegator studied in this paper.

DEFINITION 2.3. A *composition system* $\mathcal{C}$ is a tuple (A_T, I) where A_T is the *target* (or *desired*) e-service and $I = \{A_1, A_2..., A_e\}$ is a set of *subcontractor* e-services (that are available to compose the target e-service).

We use the following preliminary notion to define the concept of delegator.

DEFINITION 2.4. Let $e > 0$. For a word w, a *delegation assignment* over e is a mapping $\beta : [1..|w|] \rightarrow \mathcal{P}^{>0}([1..e])$ (i.e., from the integers between 1 and $|w|$ inclusive to non-empty subsets of the integers from 1 to e, inclusive). Let β be a delegation assignment for $w = w_1 \ldots w_n \in L(A_T)$. For $j \in [1..e]$, the *j-image* of w under β is the subsequence $image_j^\beta(w)$ of w obtained by including in $image_j^\beta(w)$ the letter occurrences w_i such that $i \in \beta(j)$. Finally, delegation assignment β over e for $w \in L(A_T)$ is *valid* in composition system $\mathcal{C} = (A_T, \{A_1, \ldots, A_e\})$ if $image_j^\beta(w) \in L(A_j)$ for each $j \in [1..e]$.

Intuitively, then, a delegation assignment for a word w is a mapping that specifies which subcontractors should process each activity in the sequence. It is valid if at the end of the delegations, each participating e-service (FA) ends up in an accepting state.

We now have:

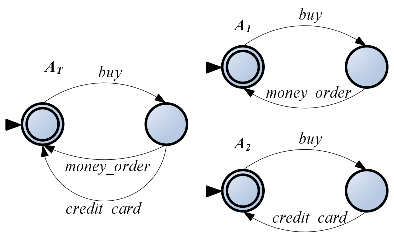


Figure 2: A Composable E-Service

DEFINITION 2.5. Let $\mathcal{C} = (A_T, \{A_1, \ldots, A_e\})$ be a composition system over alphabet Σ. Let $\alpha : \Sigma^* \times \mathbf{N} \rightarrow \mathcal{P}^{>0}([1..e])$ be a partial function. Then α is a *delegator* for $\mathcal{C}$ if for each $w \in L(A_T)$, the function $\alpha(w, \cdot)$ is total on $[1..|w|]$ and is a valid delegation assignment for w in $\mathcal{C}$.

So, there is a delegator for composition system $\mathcal{C}$ iff for each word $w \in L(A_T)$ there is a delegation assignment for w.

The following example illustrates a composition system and a delegator. We keep the example very simple for the sake of simplicity of the discussion.

EXAMPLE 2.6. For a commercial web site, let's say, we would like to develop a shopping service such that users first buy an item and then pay for it. We would like to be flexible on the payment method. That's why, users can pay either by money order or by credit card. This desired e-service can be represented by the FA A_T in Figure 2.

In the mean time, there are two existing e-services. As the payment method, one of them uses only money order and the other uses only credit card. These e-services are represented by the FAs A_1 and A_2 in Figure 2.

Let b, m and c denote the activities 'buy', 'money order', and 'credit card' respectively. The desired target e-service A_T accepts the language $(b(m|c))^*$ while the languages for A_1 and A_2 are $(bm)^*$ and $(bc)^*$, respectively. It can be verified that the desired e-service A_T can be achieved through the composition of A_1 and A_2 (i.e., *composable*). For instance, for a sequence of activities $bmbc$ (i.e., the user first pays by money order then pays by credit card), the assignments would be $b/1, m/1, b/2, c/2$ (A_1 processes the first shopping, and A_2 does the second). Obviously, for any word in $L(A_T)$, the activity m should be delegated to A_1, while the activity c should be delegated to A_2. On the other hand, the delegation of the activity b depends on whether the next incoming activity is m or c. Therefore, in an online processing model where there is an incoming sequence of activities, the decision on which e-service a particular activity should be delegated to may depend on the future activities in the sequence. ∎

Our model is a generalization of that in [3]. Specifically, in their model, the decision on delegating a particular activity depends only on activities that have already processed, while in our model we allow delegation decisions to be made based on a *lookahead*, i.e., expected activities in the future. In Figure 2, A_T is composable from A_1 and A_2 in our model, although it is not composable in their model. Therefore, the composition model studied in this paper is more powerful.

The remainder of the paper focuses on determining the existence of such delegators, and on special classes of delegators, called "k-lookahead".

3. DELEGATOR EXISTENCE

A key question for automated composition of e-services in this framework is to determine whether a delegator exists and if so, how to construct one. In this section, we use the standard automata theoretic technique to give positive answers to both questions. The key idea is to build a finite state automaton for a given composition system $\mathcal{C} = (A_T, I)$ using a variant of standard product construction for FAs[14]. Then, the product machine is used to check for the existence of a delegator for the system, and in case of positive answer, the product machine itself is a representation of the delegator.

For the simplicity of the discussion, we have a number of assumptions on FAs. First, we assume the desired e-service A_T is a Deterministic FA (DFA). If A_T is an Nondeterministic FA (NFA), then we can convert it to an equivalent Deterministic FA. We know that this conversion may take exponential time in the size of A_T [14]. However, even if we do this conversion, our complexity results in this section don't change, which is explained later. Second, all machines in the system use the same input (activity) alphabet Σ. Third, there is no *nonproductive* state in the target e-service, i.e., there exists a path from each state to an accepting state. This is a valid assumption because an FA with nonproductive states can be converted to an equivalent FA with no nonproductive states in polynomial time and also in practical sense, an e-service shouldn't have nonproductive states. In addition, each e-service in the system has no incoming transition to its starting state. Every FA can easily be modified to satisfy this property by introducing a new starting state while copying all the outgoing transition of the original starting state.

Note that every FA can be seen as a labeled directed graph where each state is represented by a node, and for each transition $r \in \delta(q, a)$, there is an edge from the node q to the node r labeled with a.

Before we formally define the product construction which is used to check the existence of a delegator, let's look at an example:

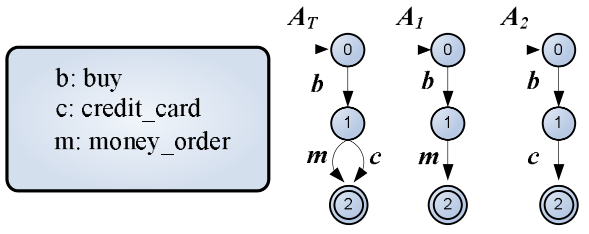


Figure 3: A Simple Composition System

EXAMPLE 3.1. In Figure 3, there is a simplified version of the system described in the previous section. We would like to get the service A_T, using the existing e-services A_1 and A_2. The corresponding product machine for this system is shown in Figure 4. Basically, the product machine keeps track of configurations of the system. For example, the configuration [110] implies that the current states of A_T, A_1 and A_2 are 1,1 and 0 respectively. Each edge represents a transition of the system from one configuration to another with the processing of the current activity. For example, the edge $(m/1)$ from [110] to [220] represents that the processing of m is done by 1 and because of this delegation, A_1 changes its state to from 1 to 2, while A_2 stays in the state 0. In configuration [000], the activity b can be delegated to A_1 and/or A_2 because at this configuration, both A_1 and A_2 can process a b. It is also possible that in some configuration, processing of an activity cannot be delegated to any e-services. For example, under the configuration [110], A_T requires processing of a c. However, under this configuration neither A_1 nor A_2 has a c transition. Therefore, for such cases, the delegation cannot be done and the machine goes to an error configuration. ∎

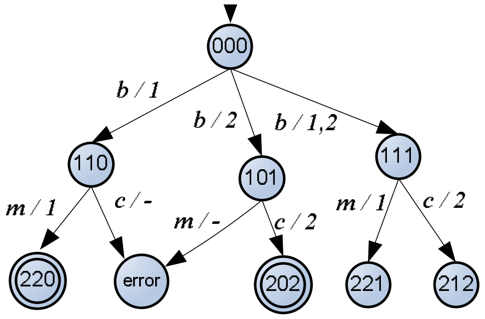


Figure 4: Product Machine for Figure 3

Below we give a formal definition for the product machine of a composition system.

DEFINITION 3.2. Given a composition system $\mathcal{C} = (A_T, I)$ of FAs where $A_T = (S_T, \Sigma, \delta_T, s_T^0, F_T)$ is the target e-service, and $I = \{A_1, A_2, ..., A_e\}$ is the set of subcontractors where $A_i = (S_i, \Sigma, \delta_i, s_i^0, F_i)$, the *product machine* $\text{PROD}_\mathcal{C}$ is a *Mealy* automaton $(S_\mathcal{C}, \Sigma_\mathcal{C}^{in}, \Sigma_\mathcal{C}^{out}, \delta_\mathcal{C}, s_\mathcal{C}^0, F_\mathcal{C})$ where

- Input and output alphabets: $\Sigma_\mathcal{C}^{in} = \Sigma$, and $\Sigma_\mathcal{C}^{out} = 2^{\{1,2...,e\}}$,
- States: $S_\mathcal{C} \subseteq ((S_T \times S_1 \times ... \times S_e) \cup \{error\})$,
- Starting state: $s_\mathcal{C}^0 = [s_T^0; s_1^0, ..., s_e^0]$,
- Accepting states: $F_C = \{[q_T; q_1, ..., q_e] \mid q_T \in F_T \wedge \forall i \in [1,e]\ (q_i = F_i \vee q_i = s_i^0)\}$.

  We use the term *configuration* for a state of the product machine. For each word accepted by the desired e-service, there is a corresponding *run*, i.e. a sequence of configurations, in $\text{PROD}_\mathcal{C}$. If in the final configuration, the target e-service and *participating*[2] subcontractors are all in their accepting states, then such a run is called an *accepting run*. Note that we distinguish an e-service who participates in the computation from the one who doesn't by simply checking whether the state of the e-service in the final configuration is the initial state or not[3].

- Before we define the transition mapping $\delta_\mathcal{C}$, let's define the set V of *"volunteer"* subcontractors. For a given configuration and an activity, V defines the set of e-services capable of processing the requested activity. In other words, under the given configuration, the activity can be delegated to any subcontractor in V. More formally, $V_{([q_T;q_1,...,q_e],a)} = \{j \mid \delta_j(q_j, a) \neq \varnothing,\ j \in [1,e]\}$.

[2]An e-service is *participating* in the computation if it processes some activity.

[3]As mentioned before, we assume for this construction that there is no incoming edge to the initial state of an e-service; therefore, once it processes an activity, it can never come back to the initial state again.

The transition mapping $\delta_\mathcal{C} : S_\mathcal{C} \times \Sigma_\mathcal{C}^{in} \rightarrow 2^{(S_\mathcal{C} \times \Sigma_\mathcal{C}^{out})}$ is defined as follows: For each configuration $[q_T; q_1, ..., q_e]$ in $\text{PROD}_\mathcal{C}$, for each activity a where $\delta_T(q_T, a)$ is defined, and for each $v \subseteq V_{([q_T;q_1,...,q_e],a)}$:

i) if $V_{([q_T;q_1,...,q_e],a)} = \varnothing$, then $\delta_\mathcal{C}([q_T; q_1, ..., q_e], a) = \{(error, \varnothing)\}$.
Intuitively, this says that if there is no volunteer, then this is an error, because under the given configuration, even though the target e-service requires the processing of a, nobody can do it.

ii) else, $\delta_\mathcal{C}([q_T; q_1, ..., q_e], a) = \{ ([r_T; r_1, ..., r_e], v) \mid r_T = \delta_T(q_T, a) \wedge (r_i = \delta_i(q_i, a)$ if $i \in v$, and $r_i = q_i$, otherwise$)\}$. When an activity is processed, the system moves from one configuration to another and the next configuration depends on the current configuration and delegation (output). In addition, the delegations are determined by the current configuration and activity (input).

The size of the product machine is $O(2^e \times |\Sigma| \times s^{2e} \times s_T)$ where e is the number of subcontractors and, s is the maximum number of states among the subcontractors and s_T is the number of states in the target e-service.

Now, we can make the connection between a product machine and a delegator. Let $L(\text{PROD}_\mathcal{C})$ denote the language accepted by the FA derived from $\text{PROD}_\mathcal{C}$ by removing the outputs from the transitions. Then, we have the following lemma which is easily verified:

LEMMA 3.3. For a composition system $\mathcal{C} = (A_T, I)$, there exists a delegator iff $L(A_T) = L(\text{PROD}_\mathcal{C})$.

According to lemma 3.3 checking language equivalence of $\text{PROD}_\mathcal{C}$ and A_T is the same question with the existence of a delegator. The equivalence of two FAs can be checked in polynomial space in the size of the FAs [14]. Therefore, this implies the following result.

THEOREM 3.4. Existence of a delegator of a composition system can be determined in exponential space.

4. DELEGATOR WITH LOOKAHEAD

In the previous section, we consider delegators in general. In this section, we formalize the concept of lookahead and introduce delegators having lookaheads, and then prove a series of characterization and complexity rusults. We assume in this section that all e-services are deterministic.

4.1 Lookahead

Let α be a delegator for $\mathcal{C} = (A_T, \{A_1, \ldots, A_e\})$. Let $u, v, v' \in \Sigma^*$ such that $uv, uv' \in L(A_T)$. Under the definition of delegator, which is quite general, there need not be any relationship between the two delegation assignments $\alpha(uv, \cdot)$ and $\alpha(uv', \cdot)$. In practice, we are interested in a delegator that can delegate incoming requests *online*, i.e., it deterministically chooses the correct subset of subcontractors that should process a request based only on letters previously read. Such delegators will be formally defined as "0-lookahead delegator" below. Although having such a delegator is appealing, in some cases it may not be possible to decide on the delegation without knowing the future requests. Let's have a look at an example illustrating this problem

EXAMPLE 4.1. Consider the composition system $\mathcal{C} = (A_T, \{A_1, A_2\})$ shown in Figure 5(a). When the system is in the starting configuration, let's say, a request for a arrives. Both A_1 and A_2

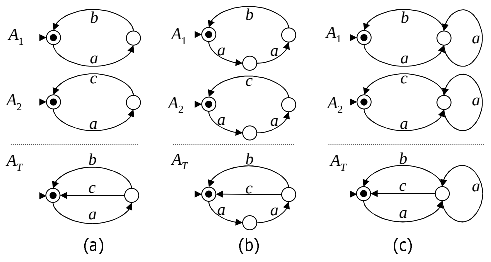


Figure 5: Lookahead in Compositions

can process it. However, who should process a is determined by the next request. If it is b, then A_1 should process a. If it is c, then A_2 should process a. Therefore, the decision of the delegation requires to check the next request. A similar concept in "Compilers" is called *lookahead*. Therefore, we say a delegator for the system in Figure 5(a) needs *at least* 1 lookahead. ∎

We now formulate what we mean by a k-lookahead delegator.

DEFINITION 4.2. Let $\mathcal{C} = (A_T, \{A_1, \ldots, A_e\})$ be a composition system, α a delegator for $\mathcal{C}$, and k a number in $\mathbb{N}$. Then α is a *k-lookahead delegator* for $\mathcal{C}$ (also described as an LA$_k$ delegator) if the following property holds: Suppose that $u, v, w, w' \in \Sigma^*$ and $|v| = k$. Then $\alpha(uvw, \cdot)$ and $\alpha(uvw', \cdot)$ coincide on each $i \in [1..|u|]$ (i.e., for each i either both are undefined, or both are defined and equal).

It is easily seen that a delegator α is k-lookahead if its behavior on the i^{th} activity of word $w = w_1 \ldots w_n$ is dependent only on the prefix $w_1 \ldots w_{i+k}$ of w (or simply w if $i + k > n$).

Intuitively, a (non-restricted) delegator uses unbounded lookahead; we sometimes use the phrase '**-lookahead delegator*' or '*LA$_*$ delegator*' in place of 'delegator', to emphasize the analogy to k-lookahead delegator. At the other extreme, a 0-lookahead delegator doesn't consider any of the activities that it hasn't read.

Let $\mathcal{C}$ be a composition system. It is straightforward to verify that there is a 0-lookahead delegator for $\mathcal{C}$ iff there is, in the vocabulary of [3], a "composition" for $\mathcal{C}$.

There is an automaton-based analog of k-lookahead delegator, which we introduce in the following two definitions.

DEFINITION 4.3. Let k be a number in $\mathbb{N}$. A *k-lookahead pre-delegator* for a composition system $\mathcal{C} = (A_T, \{A_1, A_2, ..., A_e\})$ is a (modified) Mealy automaton $D = (S_D, \Sigma, \Gamma, \delta_D, s_D^0, F_D)$ where S_D is a set of states; Σ is the input alphabet; Γ, the output alphabet, is equal to $\mathcal{P}^{>0}([1..e])$; $\delta_D : S_D \times \Sigma \times \Sigma^{\leqslant k} \to S_D \times \Gamma$ is the transition function (which may be a partial function); s_D^0 is the starting state and $F_D \subseteq S_D$ is the set of accepting states.

Intuitively, in a 2-lookahead DFA-based pre-delegator D, a transition, let's say, $\delta_D(s_1, a, bc) = (s_2, \{1, 4\})$ denotes the fact that the execution of the activity of type a is delegated to the subcontractors A_1 and A_4, based on the knowledge that the next 2 activities are b and c.

More generally, a computation on input word w by a k-lookahead pre-delegator $D = (S_D, \Sigma, \Gamma, \delta_D, s_D^0, F_D)$ proceeds as follows, starting from the start state s_D^0. Suppose now that prefix u of w has been processed, the computation is in state $s \in S_D$, and the letter after u is a. Let v be the maximal subword of w that starts immediatly after ua and has length $\leqslant k$. Suppose that $\delta_D(s, a, v) = (s', Z)$ is defined. In this case, the computation proceeds by reading a, moving to state s', and producing set Z as output. (Computation fails on w if $\delta_D(s, a, v)$ is undefined.)

If the computation of D on w succesfully processes all of w, then the delegation assignment β determined by D on w is defined so that $\beta(i)$ is the output of the i^{th} step of the computation of D on w.

DEFINITION 4.4. Let $D = (S_D, \Sigma, \Gamma, \delta_D, s_D^0, F_D)$ be a k-lookahead DFA-based pre-delegator for a composition system $\mathcal{C} = (A_T, \{A_1, A_2, ..., A_e\})$. Then D is a *k-lookahead DFA-based delegator* for $\mathcal{C}$ if the following conditions hold:

(a) If the output alphabet of D is ignored, then the language accepted by D is $L(A_T)$.

(b) For each word $w \in L(A_T)$, the delegation assignment determined by D on w is valid for $\mathcal{C}$.

It is straightforward to show that if there exists a k-lookahead DFA-based delegator for composition system $\mathcal{C}$, then there exists a k-lookahead delegator for $\mathcal{C}$. The converse also holds, as shown in Theorem 4.6 and Corollary 4.13 below.

In a practical sense, having lookahead corresponds to buffering the input symbols. For instance, a composite e-service with a 1-lookahead delegator can buffer the current activity and before processing the activity, it can ask the customer what she intends to do as the next activity. This way, the delegator can better guide the delegation of the current activity.

Continuing with Figure 5, there exist LA$_*$ delegators for all three composition systems (each system is composable). This can be verified using the method described in the previous section. In addition, (a) has an LA$_1$ delegator, while (b) has an LA$_2$ delegator but no LA$_1$ delegator. In (c), the activity a can occur any number of times, thus in this case there is no LA$_k$ delegator for any $k \in \mathbb{N}$. On the other hand, if *all* succeeding activities are known, each activity can be delegated to A_1 or/and A_2 properly; therefore, (c) has an LA$_*$ delegator. As can be seen from this example, the hierarchy based on the amount of lookahead is strict. As a result, the following is established.

THEOREM 4.5. For each $k > 0$, there exists a composition system $\mathcal{C}$ such that $\mathcal{C}$ has LA$_k$ delegators but no LA$_{(k-1)}$ delegators. There is also a composition system $\mathcal{C}$ that has an LA$_*$ delegator but no LA$_k$ delegator for any k in $\mathbb{N}$.

As noted above, the model of delegators used in [3] is exactly LA$_0$, i.e., with *no* lookahead. Although it was shown in [3] that existence of an LA$_0$ delegator can be determined in exponential time, we present a direct analysis on LA$_0$ delegators using FAs having a finer characterization on the complexity bound. Later, we use this technique for k-lookahead delegators.

4.2 0-Lookahead Delegator Construction

In this section, we show how to check the existence of a 0-lookahead (LA$_0$) delegator, and if one exists, how to construct a 0-lookahead DFA-based delegator.

The following theorem makes a connection between an LA$_0$ delegator and the product machine.

THEOREM 4.6. Let $\mathcal{C} = (A_T, \{A_1, \ldots, A_e\})$ be a composition system with deterministic e-services. Then $\mathcal{C}$ has a 0-lookahead delegator iff there is a 0-lookahead DFA-based delegator D for $\mathcal{C}$ that is a subgraph of $\text{PROD}_{\mathcal{C}}$ (denoted $D \subseteq \text{PROD}_{\mathcal{C}}$).

PROOF. (Sketch) Let Σ be the alphabet of $\mathcal{C}$. Let α be a 0-lookahead delegator for $\mathcal{C}$. We begin by creating a minimal tree $\mathcal{T}$ with a branch corresponding to each word of $L(A_T)$. (In particular if $uv \in L(A_T)$ then the branch corresponding to u in $\mathcal{T}$ is a subbranch of the branch corresponding to uv.) For $w = w_1 \ldots w_n \in L(A_T)$ label the i^{th} edge along the branch of w by the pair $(w_i, \alpha(w, i))$, i.e., by the i^{th} letter of w and by the delegation assigned to position i by α acting on w. Since α is a 0-lookahead delegator this labeling is well-defined. (Tree $\mathcal{T}$ is essentially the "internal schema of a composition", in the terminology of [3].)

We introduce a labeling function λ that associates to each node $\mathcal{T}$ an element of $S_{\mathcal{C}}$, the set of states of the product machine $\text{PROD}_{\mathcal{C}}$, in the following inductive manner. Label the root with the start state of $\text{PROD}_{\mathcal{C}}$. Suppose that $\lambda(x)$ is defined for some node x of $\mathcal{T}$, and that y is a child of x where the edge from x to y is labeled by (a, Z). Then set $\lambda(y)$ to be the state of $S_{\mathcal{C}}$ reached from $\lambda(x)$ by moving the A_T-coordinate of $\lambda(x)$ according to the transition in A_T upon reading a, and for each $j \in Z$ moving the A_j-coordinate of $\lambda(x)$ according the transition in A_j upon reading a.

We note that by construction λ has the following properties with respect to α and $\mathcal{T}$. First, λ is *consistent* on $\mathcal{T}$ with $\text{PROD}_{\mathcal{C}}$ in the following sense: If y is a child of x, and the edge from x to y is labeled by (a, Z), then there is a transition in $\text{PROD}_{\mathcal{C}}$ from state $\lambda(x)$ that is labeled by (a, Z) which leads to state $\lambda(y)$. Second, λ is *consistent* on $\mathcal{T}$ with α in the following sense: for each word $w = w_1 \ldots w_n \in L(A_T)$, if x is the node reached by traversing $\mathcal{T}$ while reading $w_1 \ldots w_i$, then $\lambda(x)$ is the state of $\text{PROD}_{\mathcal{C}}$ that is reached when processing $w_1 \ldots w_i$ according to α. These two consistency properties will be preserved during each step of the construction below.
We now perform a splicing argument on $\mathcal{T}$ which results in a 0-lookahead DFA-based delegator for $\mathcal{C}$. Suppose that $u, v \in \Sigma^*$, and that x is the node of $\mathcal{T}$ corresponding to the end of word u and y is the node of $\mathcal{T}$ corresponding to the end of word uv. Suppose further that $\lambda(y) = \lambda(x)$, and that there is no pair of nodes x', y' that are proper ancestors of y with $\lambda(y') = \lambda(x')$. (Note that the depth of y in $\mathcal{T}$ is at most $|S_{\mathcal{C}}|$.) It is easily verified, from the fact that α is a 0-lookahead delegator, that for all w, $uw \in L(A_T)$ iff $uvw \in L(A_T)$. Let z be the parent of y and let the edge (z, y) in $\mathcal{T}$ be labeled by (b, Z). Create a graph $\mathcal{G}$ from $\mathcal{T}$ by (a) deleting the subtree rooted at y, and (b) inserting a "back-edge" from z to x, which is labeled by (b, Z). Define labeling function λ' on the nodes of $\mathcal{G}$ to be the restriction of the labeling λ on $\mathcal{T}$ to the nodes of $\mathcal{G}$. Create a new 0-lookahead delegator α' from α, by using $\mathcal{G}$ as a guide. (So in particular, for each $w \in \Sigma^*$ and each $j > |uv|$, $\alpha'(uvw, j) = \alpha(uw, j - |v|)$. An inductive argument can be used to verify that α' is again a 0-lookahead delegator. Furthermore, λ' is consistent on $\mathcal{G}$ for both $\text{PROD}_{\mathcal{C}}$ and α'.

This construction can be continued iteratively on graph $\mathcal{G}$, at each step deleting an infinite "subtree" of $\mathcal{G}$. We continue to call the new edges introduced "back-edges". The end result is a graph $\mathcal{H}$, 0-lookahead delegator β, and a labeling function γ which is consistent on $\mathcal{H}$ with $\text{PROD}_{\mathcal{C}}$ and β, where there is no back-edge-free path in $\mathcal{H}$ that has two distinct nodes with the same value under λ. This graph has size bounded by $|\Sigma|^{s^e}$, where s is the maximum size of the e-services in $\mathcal{C}$.

Now that $\mathcal{H}$ is finite, we continue with the splicing operation. Specifically, given nodes x, y with $\lambda(x) = \lambda(y)$, we delete y and replace all in-edges of y by in-edges to x (again retaining the same labels). Also, delete all nodes no longer reachable from the "root" of $\mathcal{H}$. After each such step, inductive arguments are used to show that the new graph again yields a 0-lookahead delegator γ for $\mathcal{C}$, and that the labeling function is consistent with $\text{PROD}_{\mathcal{C}}$ and γ. The conclusion of this iteration is a graph $\mathcal{J}$ where each node has a distinct label. From here it is easy to convert $\mathcal{J}$ into a 0-lookahead DFA-based delegator for $\mathcal{C}$ which is a deterministic sub-automaton of $\text{PROD}_{\mathcal{C}}$. ∎

Theorem 4.6 says that we can check the existence of an LA_0 delegator by looking for a deterministic subgraph D of $\text{PROD}_{\mathcal{C}}$ such that $L(D) = L(A_T)$. Therefore, we can search through all subgraphs and check language equivalence; however, since in the worst case there are exponential number of subgraphs, this brute force approach can take exponential time in the size of $\text{PROD}_{\mathcal{C}}$ which is also exponential. Hence, the total time complexity of this approach would be double exponential. A better approach to reduce the search space before checking for language equivalence. In order to reduce the size of the search space we need to filter out some of the subgraphs. For that purpose, we have the following observation:

LEMMA 4.7. Let a subgraph D of $\text{PROD}_{\mathcal{C}}$ be an LA_0 delegator for a composition system $\mathcal{C}$. Then, a configuration $Q = [q_T; q_1, ..., q_e]$ in $\text{PROD}_{\mathcal{C}}$ cannot be part of the subgraph D if either

- $Q \notin F_{\mathcal{C}}$ and $q_T \in F_T$ (the configuration is not accepting but the corresponding state of the target e-service is accepting),
- or there exists an activity a such that $\delta_T(q_T, a)$ is defined even though $\delta(Q, a)= \{(error, \varnothing)\}$.

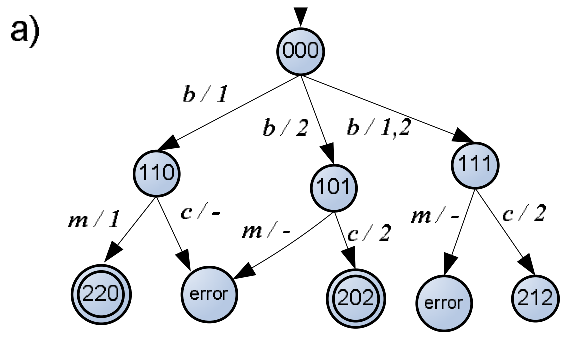
PROOF. (Sketch) Before go into the details, note that for every configuration Q in D, there exists a path from the starting configuration to Q with some word w (otherwise, it is not connected). In addition, this path corresponds to a run of the machines in I on w. We have two cases:

i) Assume $Q \notin F_{\mathcal{C}}$ and $q_T \in F_T$. Then, since q_T is an accepting state in the target, this implies $w \in L(A_T)$. On the other hand, D is LA_0, and therefore there cannot be more than one path for each word on D (remember that delegations of an LA_0 delegator doesn't depend on any succeeding symbol). Therefore, $w \notin L(D)$ is true which means $L(A_T) \neq L(D)$ and this is a contradiction.

ii) Assume there exists an activity a such that $\delta_T(q_T, a)$ is defined while $\delta_{\mathcal{C}}\ (Q, a)= \{(error, \varnothing)\}$. Similar to the previous case, there is a path in D with some word w, ending at Q. Since q_T is a productive[4] state and $\delta_T(q_T, a)$ is defined, there exists a word $wav \in L(A_T)$. According the definition of LA_0 delegator, first $|w|$ activities of w and wav are delegated in the same way. Therefore, wav cannot be in $L(D)$ since $\delta(Q, a)$ is error. Therefore, this implies $L(A_T) \neq L(D)$ which is a contradiction.
∎

Let's call a configuration described in Lemma 4.7 a *bad* configuration. According to Lemma 4.7, such a configuration cannot be

[4]Recall that a state is productive if an accepting state is reachable from that state. In our system, the target e-service contains only productive states. Note that any FA with nonproductive states can be converted an FA with no nonproductive states in polynomial time.



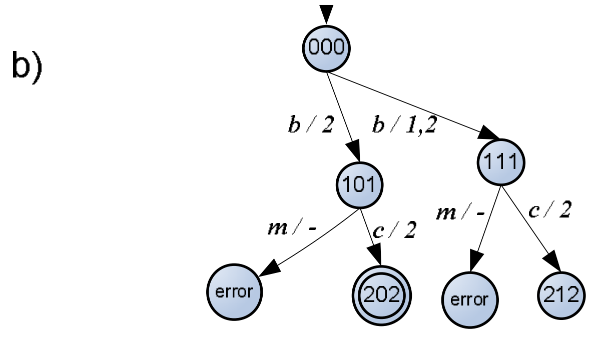


Figure 6: First 2 Steps of Bad Configuration Removal from $\text{PROD}_\mathcal{C}$ **in *Figure 4*: a)** [221] **is removed, b)**[110] **is removed.**

part of any LA_0 delegator $D \subseteq \text{PROD}_\mathcal{C}$; therefore, it can be removed from $\text{PROD}_\mathcal{C}$ which reduces our search space. Algorithm 1 describes the remove operation and explains how to update $\text{PROD}_\mathcal{C}$.

EXAMPLE 4.8. Let's reexamine the product machine shown in Figure 4. The configurations [221] and [212] are bad because they are not accepting, while the state 2 is accepting in A_T. Also [110] and [101] are bad configurations. For [110], neither A_1 nor A_2 can process c, although A_T has a transition from the state 1 with c. Figure 6 (a) and (b) show the product machine after [221] and [110] are removed respectively. Note that when [221] is removed, [111] becomes a bad configuration. On the other hand, the removal of [110] doesn't cause [000] to become bad. ∎

Algorithm 1 RemoveBadConfiguration(Q, $\text{PROD}_\mathcal{C}$)

```
 1: for each incoming edge $e$ to $Q$ do
 2:     Let $e$ be labeled as $(a, v)$.  /* $v$ is the set of delegations for an activity $a$ */
 3:     Let $Q^s$ be the source configuration of $e$.  /* $e$ is an edge from $Q$ to $Q_s$ */
 4:     Let $\delta(Q^s, a)$ be the transition defined in $Q^s$ with the symbol $a$.  /* set of next configurations and delegations */
 5:     if $\delta(Q^s, a) - \{(Q, v)\} \neq \varnothing$ then
 6:         $\delta(Q^s, a) = \delta(Q^s, a) - \{(Q, v)\}$  /* there exist other possible delegations */
 7:     else
 8:         $\delta(Q^s, a) = \{(error, \varnothing)\}$  /* no other delegations */
 9:     end if
10: end for
```

Let λ denote Algorithm 1. Then, the following can be proven easily.

LEMMA 4.9. Let Q be a bad configuration in $\text{PROD}_\mathcal{C}$. If there is an LA_0 delegator D, then $D \subseteq \lambda(Q, \text{PROD}_\mathcal{C}$).

Lemma 4.9 suggests that we can apply Algorithm 1 enough number of times to eliminate all the bad configurations, and then we can check for the existence of an LA_0 delegator. For notational convenience we call the final structure having no bad configurations also $\text{PROD}_\mathcal{C}$. Algorithm 2 follows this approach. It removes all the bad configurations and updates $\text{PROD}_\mathcal{C}$ accordingly. Note that the removal of a bad configuration may cause some other states to become bad and the algorithm takes care of that while updating the structure at each step. It is important to see that if the starting configuration of $\text{PROD}_\mathcal{C}$ is removed then there doesn't exist an LA_0 delegator since the starting configuration must be part of any delegator. Therefore, if $\text{PROD}_\mathcal{C}$ is not empty, it contains an LA_0 delegator as a subgraph. For example, in Example 4.8, the product machine becomes empty. This shows that there is no LA_0 delegator for that system which is true because it can be verified that the system requires at least 1 lookahead.

Although we reduce our search space using bad configuration removal, we still need to search for an LA_0 delegator among all the left subgraphs. The following lemma shows that we can directly construct an LA_0 delegator from the product machine.

LEMMA 4.10. For a configuration $[q_T; q_1, ..., q_e] \in \text{PROD}_\mathcal{C}$ and an e-service A, let's define a function α such that $\alpha([q_T, q_1, ..., q_e], i) = q_i$ where $i \in \{T, 1, 2..., e\}$ (i.e., for the given configuration, q_i is the corresponding state of the ith machine). Also, for a configuration $Q \in \text{PROD}_\mathcal{C}$, let $L(Q)$ denote the language of the same machine except Q is the starting configuration. Then, for any two configurations Q_1 and Q_2 in $\text{PROD}_\mathcal{C}$, $\alpha(Q_1, T) = \alpha(Q_2, T)$ implies $L(Q_1) = L(Q_2)$.

PROOF. (Sketch) Assume there exists a word w such that $w \in L(Q_1)$ but $w \notin L(Q_2)$. Since $w \in L(Q_1)$, the run on w starting at Q_1 ends in an accepting configuration Q_1' while the run on w starting at Q_2 ends either in a rejecting or in the *error* configuration. That's why, we have two cases.

i) The run on w starting at Q_2 ends in a rejecting configuration Q_2'. Q_1' is accepting; therefore, $\alpha(Q_1', T)$ must be an accepting state in A_T. Since A_T is a deterministic finite automata[5] $\alpha(Q_1', T) = \alpha(Q_2', T)$. But then, Q_2' is a bad configuration because it must be an accepting configuration since $\alpha(Q_2', T)$ is an accepting state in A_T. This is a contradiction, because there doesn't exist any bad configuration in the final product machine.

ii) The run on w starting at Q_2 ends in the error configuration. Then, there exists a prefix word ua of w such that run on u ends in a configuration Q_2' and from that configuration, the system goes to the error configuration while processing a. Assume run on u starting at Q_1 ends in a configuration Q_1'. Since A_T is a DFA, $\alpha(Q_1', T) = \alpha(Q_2', T)$. Since run on w starting at Q_1 doesn't end in an error state, $\delta(Q_1', a)$ is defined. That means $\delta_T(\alpha(Q_1', T), a)$ is also defined. Then, Q_2' must be a bad configuration because even though $\delta(\alpha(Q_2', T), a)$ is defined (processing of a is in the desired e-service is required), there is no volunteer subcontractor to process a. Since there is no bad configuration in the final product machine, this is a contradiction. ∎

[5]As we specified before, we assume A_T is a DFA. If it is an NFA then we can first convert it to a DFA and the construct the product machine. Our complexity results still hold despite the conversion.

By using Lemma 4.10, we can construct a DFA realizing an LA$_0$ delegator D using the product machine in the following manner: Start from the initial configuration. In each configuration Q, for each activity a, we have a number of, say m, possible delegations, i.e., $\delta(Q,a) = \{(Q^1,v^1),(Q^2,v^2),...,(Q^m,v^m)\}$ where each v^i is a subset of volunteer subcontractors $V_{(Q,a)}$ and each Q^i is a next configuration. However, for all these next configurations, the corresponding state of the target e-service is the same, i.e., $\alpha(Q^1,T) = \alpha(Q^2,T) = ... = \alpha(Q^m,T)$, because the target e-service is a DFA. Therefore, by Lemma 4.10, it is true that $L(Q^1) = L(Q^2) = ... = L(Q^m)$. That's why, picking one delegation and removing all the other delegations don't affect $L(\text{PROD}_{\mathcal{C}})$. Algorithm 2 below removes all bad configurations and then does one graph traversal on the product machine and at each configuration, picks one delegation for each possible activity. By this way, it constructs a deterministic FA representing a delegator for a given system.

Algorithm 2 Constructs an LA$_0$ delegator for a composition system $\mathcal{C}$ =(A_T, I)

```
1: Construct PROD_C
2: repeat
3:    Find a "bad" configuration in PROD_C
4:    Remove it and update PROD_C
5: until No "bad" configuration is left
6: if PROD_C is empty then
7:    return error /∗ there is no LA_0 delegator ∗/
8: else
9:    Construct a deterministic finite automata representing an
      LA_0 delegator
10: end if
```

Now let's look at the time complexity of Algorithm 2. First line of the algorithm takes exponential time as explained before. Line 2-5 is repeated at most the number of states which is exponential and removal a bad configuration can be carried out in exponential time. Line 9 requires a depth first search on PROD$_{\mathcal{C}}$ which takes exponential time. More specifically, if the target e-service is a DFA, the total time complexity of the algorithm is $O(|\Sigma| \times 2^{2e} \times s^{2e} \times s_T^2)$ where $|\Sigma|$, e, s and s_T denote the size of alphabet, the number of e-services, the maximum number of states among subcontractors and the number of states in the target e-service. Therefore, the complexity is polynomial in $|\Sigma|$, s and s_T, and exponential in e. (If the target e-service is an NFA, then instead of s_T^2 we have 2^{2s_T} in the complexity formula.) To summarize:

THEOREM 4.11. For a composition system with deterministic e-services, existence of an LA$_0$ delegator can be determined in time polynomial in the alphabet and sizes of the target and subcontractor e-services, and exponential in the number of subcontractors.

4.3 k-Lookahead Delegator Construction

Here, we show how we can reduce the problem of deciding whether a composition system of DFA's has an LA$_k$-delegator (for a given k) to the problem of deciding whether a composition system of DFA's has an LA$_0$-delegator. Note that for the sake of discussion, we assume all the machines are DFA's. The same approach can be applied if the machines are NFAs.

Let $\mathcal{C} = (A_T, \{A_1,...,A_e\})$ be a composition system of DFA's and k be a positive integer. So that there are always always k lookahead symbols, let $\#$ be a new symbol and f (resp., f_i) be a new state. Extend the transition function of A_T (resp., A_i) by defining the transitions from any accepting state, including f (resp., f_i), on symbol $\#$ to f (resp, f_i). Then make f (resp., f_i) the only accepting state. Thus the new DFA accepts the language $L(A_T)\#^+$ (resp., $L(A^i)\#^+$). For the notational convenience, also call the new machines $A_T, A_1, ..., A_e$.

Let δ be the transition function of the target DFA A_T. Let x be a string of length k over $\Sigma \cup \{\#\}$. We construct from A_T a DFA A_T^x with the transition function δ^x as follows:

1. The starting state of A_T^x is $[q_0, x]$, where q_0 is the starting state of A_T.
2. For every state q of A_T, every string y of length $k-1$, and any symbols a, b, let
   $\delta^x([q, ay], b) = [\delta(q,a), yb]$.
3. The only accepting state of A_T^x is $[f, \#^k]$ where f is the unique accepting state of A_T.

Similarly we can construct for each A_i ($1 \leqslant i \leqslant e$), the DFA A_i^x.

Figure 7 shows the application of the described construction to the system in Figure 5(b).

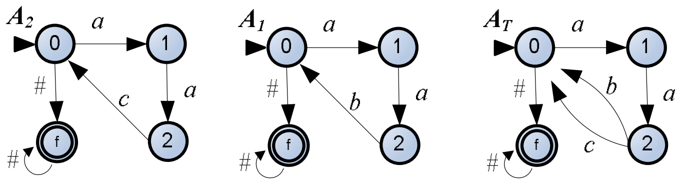


Figure 7: Addition of # symbols to the machines in Figure 5(b)

Then we have:

THEOREM 4.12. Let $\mathcal{C} = (A_T, \{A_1,...,A_e\})$ be a composition system of DFA's and k be a positive integer. Then $\mathcal{C}$ has an LA$_k$-delegator if and only if for every string x of length k, the system $\mathcal{C}^x = (A_T^x, \{A_1^x,...,A_e^x\})$ has an LA$_0$-delegator.

PROOF. Clearly, $\mathcal{C}$ has an LA$_k$ delegator if and only if for every string x of length k, the product machine PROD$_{\mathcal{C}^x}$ has a deterministic subgraph that accepts $L(A_T^x)$, and (by Theorem 4.6) if and only if $\mathcal{C}^x$ has an LA$_0$ delegator. ∎

The following is easily shown.

COROLLARY 4.13. Composition system $\mathcal{C}$ has a k-lookahead delegator iff it has a k-lookahead DFA-based delegator.

The complexity of the algorithm just described can be determined as follows: We have to check for every x of length k whether $\mathcal{C}^x$ has an LA$_0$-delegator. Thus the time complexity is $|\Sigma|^k$ times the time complexity of checking if $\mathcal{C}^x$ has an LA$_0$-delegator (where $|\Sigma|$ is the size of the input alphabet of the DFA's). The size of each DFA in $\mathcal{C}^x$ is $|\Sigma|^k$ times the size of the original DFA. ¿From this and Theorem 4.11 we obtain:

COROLLARY 4.14. For a composition system with deterministic e-services and k in $\mathbb{N}$, existence of an LA$_k$ delegator can be determined in time polynomial in the alphabet and sizes of the target and subcontractor e-services, and exponential in k and the number of subcontractors.

We now give an example to illustrate the reduction.

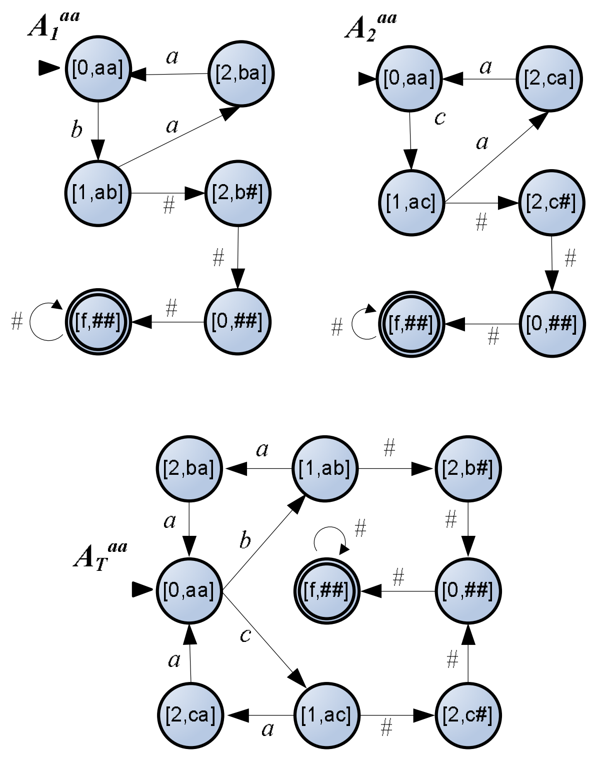


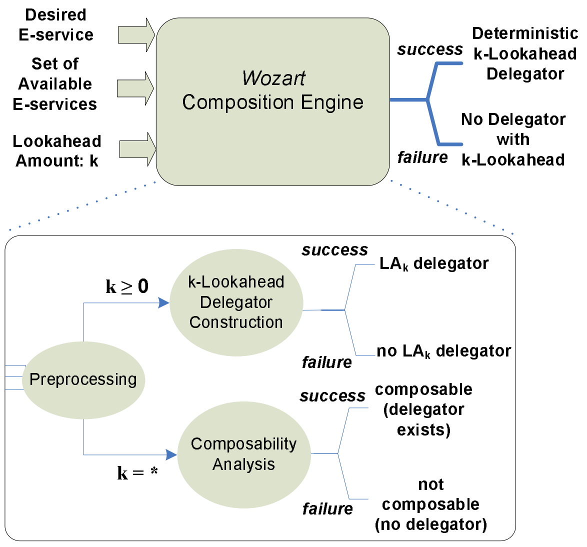
Figure 8: computation of $\mathcal{C}^{aa}$ for Figure 5 (b)

EXAMPLE 4.15. Figure 5 (b) shows a system having an LA_2 delegator. As described above, first a new accepting state f and the transitions for the ending symbol $\#$ are added. Figure 7 shows the resulting machines. Then, for every word of length 2, a new composition system $\mathcal{C}^x$ is computed and checked for the existence of an LA_0 delegator. Figure 8 shows the system for $x = aa$. Note that the real construction produces many more states than that are shown in Figure 8. For instance, in A_T^{aa}, the state [1,aa] is reachable from [0,aa] and that's why it should be part of A_T^{aa}. However, [1,aa] can't reach any accepting states (not productive); therefore, it has no effect on the delegation construction as described before. As a result, because of the space limitations such nonproductive states are not shown. ∎

Note that by using the reduction described above, any technique for LA_0 checking, e.g., the one proposed by [3], can be used. To compare with [3], as we mentioned before, our technique has a finer characterization of the complexity bound.

5. *WOZART*: A TOOL FOR AUTOMATED COMPOSITION OF E-SERVICES

In this section, we briefly describe an automated composition tool named *Wozart* implemented using the approaches presented in the previous sections. Figure 9 shows the overall architecture of Wozart. Wozart has two functionalities. First functionality is the construction of a deterministic delegator. Given a desired e-service, existing e-services and a lookahead amount k (a number or *), the tool applies Algorithm 2 and tries to generate a k-lookahead delegator. If it succeeds, the output is a Mealy FA having the activities as input and the delegations as output so that it can be used to orchestrate e-services collectively to achieve the desired e-service. Otherwise, it means that k lookahead is not enough to determine the delegation of the desired e-service to the given e-services.



Desired E-service
Set of Available E-services
Lookahead Amount: k
Wozart Composition Engine
success
Deterministic k-Lookahead Delegator
failure
No Delegator with k-Lookahead
Preprocessing
k ≥ 0
k-Lookahead Delegator Construction
success
LA_k delegator
failure
no LA_k delegator
k = *
Composability Analysis
success
composable (delegator exists)
failure
not composable (no delegator)

Figure 9: Wozart Architecture

Secondly, Wozart can give an answer to the question of delegator existence. As described in the previous section, there are cases where there is no bound on lookahead (i.e., determination of delegations requires to check all succeeding activities), even though a delegator exists. Therefore, for the user, it may not be possible to specify a bound on the lookahead. In that case, the tool performs a composability analysis and it gives a positive or negative answer to the question of composability.

For the *complete travel service*, Figure 10 shows the LA_0 delegator computed by Wozart. As it can be seen, the delegator assigns each activity to the existing e-services and achieves the behavior of the desired complete travel e-service.

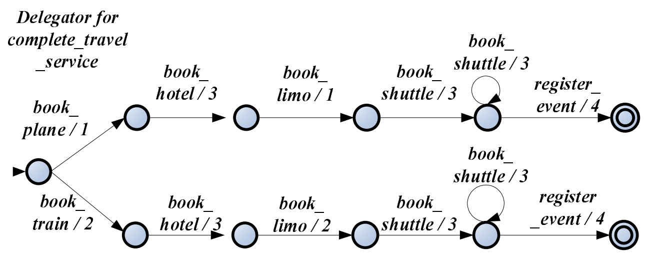


Figure 10: LA_0 delegator for the Travel Service Problem

Note that before both delegator construction and composability analysis, Wozart first performs a preprocessing on the FAs. Remember that for the simplicity of the discussion, we previously assumed that the target e-service is a DFA and doesn't contain any nonproductive states. Also, we assumed there is no incoming edge to the starting states in the machines. Wozart preprocesses and modifies the FAs so that they satisfy those assumptions.

Wozart is implemented using the WSAT library[12]. The input/output format, source code and the other details can be found in the following address: *http://www.cs.ucsb.edu/˜gerede/Wozart/index.html*

6. CONCLUSION AND FUTURE WORK

In this paper, we defined a general class of delegators called ”lookahead” delegators and investigated the complexity of constructing such delegators, if they exist. We showed that for the case of deterministic e-services, a k-lookahead delegator can be constructed in time polynomial in the size of the target and subcontractor e-services, and exponential in the size of k and the number of subcontractor e-services. We also briefly described *W*ozart, an automated mediator construction tool that we implemented using the techniques presented in this paper.

We should mention that a recent paper [8] also looked at the the decidability of composability and existence of a bounded delegator for various classes of machines including finite automata augmented with unbounded storage (e.g., counters and pushdown stacks). If some Presburger constraints (e.g., some linear relationships on the number of symbols delegated to each service) are imposed on a mediator, then the existence of such a k lookahead mediator for a fixed k is also decidable. However, the complexities of the decision procedures in [8] are rather high. In particular for the case of systems of nondeterministic finite automata, the procedure for deciding the existence of a k lookahead delegator takes *nondeterministic* exponential time in k and the sum of the sizes of the automata. We have improved this result with a procedure that runs in *deterministic* exponential time in this paper.

The result achieved in [8] concerning decidability of existence of constrained mediator is quite interesting because in practical applications there may exist some constraints a mediator has to follow. For example, it is possible that a specific type of activity shouldn't be delegated to the same service 5 times more than it is delegated to the other services for fairness reasons. For another example, assume delegation of an activity to each service costs differently and the total cost of activities shouldn't exceed some specified amount. Both cases can be described by some Presburger formulas, and the existence of such a constrained delegator is decidable. Therefore, finding a delegator which delegates activities in an optimal way with respect to a Presburger formula (e.g., cost minimization) is a significant problem. In addition to this, the question of whether there exists a bounded lookahead delegator (a bound on k) which is left open in [8] is also left as future work.

Acknowledgement. The authors thank Daniela Berardi for many useful discussions on material related to this paper.

7. REFERENCES

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In *Proc. 16th Int. Colloq. on Automata, Languages and Programming*, 1989.

[2] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2003.

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

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

[5] J. Buchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

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

[7] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 2000.

[8] Z. Dang, O. Ibarra, and J. Su. Composability of infinite-state activity automata. In *Proceedings of the 15th International Symposium on Algorithms and Computation*, Hong Kong, December 2004. To appear.

[9] H. Davulcu, M. Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic based modeling and analysis of workflows. In *Proc. ACM Symp. on Principles of Database Systems*, pages 25–33, 1998.

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

[11] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *Proc. Int. World Wide Web Conf. (WWW)*, May 2004.

[12] X. Fu, T. Bultan, and J. Su. Wsat: A tool for formal analysis of web services. In *16th Internatioanal Conference on Computer Aided Verification*, July 2004.

[13] F. Gandon and N. Sadeh. A semantic eWallet to reconcile privacy and context awareness. In *Proc. Second Int. Semantic Web Conf. (ISWC)*, Florida, Oct. 2003.

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

[15] O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *Proc. IEEE Symposium on Logic In Computer Science*, 2001.

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

[17] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

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

[19] OWL-S 1.0 Release. `http://www.daml.org/services/owl-s/1.0/`, May 2003.

[20] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. IEEE Symp. on Foundations of Computer Science*, 1990.

[21] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, MA, 2001.

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

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

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

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