Complexity and Composition of Synthesized Web Services

Wenfei Fan
University of Edinburgh &
Bell Labs
wenfei@inf.ed.ac.uk

Floris Geerts
University of Edinburgh
fgeerts@inf.ed.ac.uk

Wouter Gelade
Hasselt University &
Transnational Univ. of Limburg
wouter.gelade@uhasselt.be

Frank Neven
Hasselt University &
Transnational Univ. of Limburg
frank.neven@uhasselt.be

Antonella Poggi
Sapienza Università di Roma
poggi@dis.uniroma1.it

Abstract

The paper investigates fundamental decision problems and composition synthesis for Web services commonly found in practice. We propose a notion of *synthesized Web services* (SWS's) to specify the behaviors of the services. Upon receiving a sequence of input messages, an SWS issues multiple queries to a database and generates actions, in parallel; it produces external messages and database updates by synthesizing the actions parallelly generated. In contrast to previous models for Web services, SWS's advocate parallel processing and (deterministic) synthesis of actions. We classify SWS's based on what queries an SWS can issue, how the synthesis of actions is expressed, and whether unbounded input sequences are allowed in a single interaction session. We show that the behaviors of Web services supported by various prior models, data-driven or not, can be specified by different SWS classes. For each of these classes we study the non-emptiness, validation and equivalence problems, and establish matching upper and lower bounds on these problems. We also provide complexity bounds on composition synthesis for these SWS classes, identifying decidable cases.

Categories and Subject Descriptors: H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *Web-based services*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic — *Computational Logic*
General Terms: Languages, Theory, Design, Verification.

1. Introduction

The prevalent use of Web services highlights the need for studying fundamental decision problems associated with Web services. Given a service and a transaction, one wants to know whether the transaction is the result of a run of the service. This problem is referred to as the validation problem [2, 29] and is useful for, *e.g.,* fraud detection. We also want to determine whether two given services are equivalent (the equivalence problem [2, 29]). This allows us to replace a service with an equivalent yet cheaper one. We are also interested in finding out, at compile time, whether a service makes sense, *i.e.,* it can generate valid transactions (the non-emptiness problem). These decision problems are not only of theoretical interest, but are also important in practice.

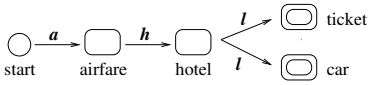
Another central technical issue concerns Web service composition. When a client request cannot be satisfied by any available service, one wants to automatically generate a mediator [7] to coordinate available services and meet the client's requirements. The composition synthesis problem is to determine, given a request and a set of available services, whether there exists such a mediator that, taken together with the available services, delivers the requested service.

The complexity of the decision problems and composition synthesis highly depends on how Web services are defined. A variety of standards (*e.g.,* [7, 25, 27, 30, 31]) and models (*e.g.,* [2, 5, 6, 12, 13, 15, 18, 26, 29]) have been proposed for Web services (see [21] for a survey). Some of the models specify the behaviors and interactions of Web services in terms of finite-state automata (FSA) [6, 15, 18]. Other models are based on data-driven transducers that support (infinite) states parametrized with input relations and local databases, and at each state, generate actions via queries on the data [2, 5, 12, 13, 29]. The validation and equivalence problems, as well as other interesting verification problems, have been studied for the transducers [2, 12, 13, 29]. While composition synthesis was studied for FSA abstractions [6, 18, 24], little is known for data-driven transducers.

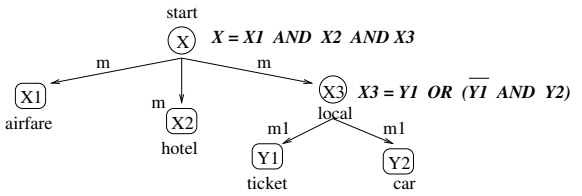
To study the decision problems and composition synthesis for various models in a comparative basis, this paper proposes a notion of *synthesized Web services* (SWS's) as a uniform formalism to characterize FSA and transducer abstractions. An SWS specifies the behavior of a service in terms of actions, *i.e.,* external messages and database updates, generated in response to various sequences of input messages. As opposed to prior models, SWS's advocate deterministic synthesis of actions and parallel processing.

Example 1.1: Consider a service for booking travel packages to Disney World Orlando. Customers commit to purchase a package only if they can get (1) a reasonable airfare,

PODS'08, June 9–12, 2008, Vancouver, BC, Canada.



(a) An FSA expressing the service for travel packages



(b) SWS specification

Figure 1: FSA specification vs. SWS specification

(2) a nice hotel, *and* (3) *either* (a) Disney tickets *or* (b) a rental car with discount. An FSA abstraction of the service is shown in Fig. 1(a), in which *airfare*, *hotel*, *ticket* and *car* denote FSA specifications for booking flights, hotel, tickets and rental car, respectively (details omitted). Similarly a data-driven transducer can be specified, which outputs actions at each state transition. Such specifications are, however, hampered by the following concerns. First, there is no temporal dependency between airfare and hotel checking; they can be conducted *in parallel*, instead of *sequentially*. Second, the customers make a commitment *only if* conditions 1–3 are *conjunctively* satisfied. Thus the booking of flight and hotel should be *deferred* until condition 3 is inspected, since the failure of 3 may force the earlier transactions to rollback. Third, although conditions (a) and (b) are disjunctive, the customers may want to *deterministically* commit to one of the two options, rather than make decisions *nondeterministically* or worse, commit to book both rental car and tickets.

An SWS specification is depicted in Fig. 1(b), along the same lines as alternating finite automata (AFA). Upon receiving an input message m from a user (*e.g.,* travel date and price range), the SWS checks airfare, hotel, followed by car rental and ticket sale based on input message m_1 specifying user requirements, *in parallel*. Each state inspects a certain condition and keeps its truth value in a Boolean variable. The value of the variable may also be computed by a Boolean formula on variables associated with the successor states. For example, X_3 is defined by $Y_1 \vee (\bar{Y}_1 \wedge Y_2)$, in favor of ticket sale when both Y_1 and Y_2 are true, and it is evaluated as soon as the values of Y_1, Y_2 are available. The SWS returns true only if the condition at the start state holds.

A data-driven SWS can be specified similarly. At each state certain actions can be generated, beyond just Boolean values, via *queries* on the input messages and databases (*e.g.,* information about flight tickets and hotel rooms). Actions are passed upward and synthesized at each state. The actions are generated *deterministically*, and are *not* committed *until* the condition at the start state is confirmed to hold. With SWS's one can specify various versions of the service, data-driven or not, in a uniform framework.

These SWS's advocate parallelism by decoupling unnecessary temporal dependencies, and support "backward" determinism to synthesize actions (messages and updates). □

In a nutshell, an SWS specifies the observable behavior of a Web service in response to inputs from users or other services. It is defined with a set of states parametrized with messages and a local database. At each state it simultaneously issues multiple queries to the database and input message, and passes the result of each query to a successor state. It generates actions when no more successor states can be triggered. The actions are passed upward to its parent state, which synthesizes actions from its successor states, again via a query. The actions produced at the start state are the output of the SWS. Intuitively, for each input sequence, the output indicates the behavior of the service in a communication session. The sessions are "flexible" in that they are determined by the lengths of the input sequences. At the end of each session the actions are committed, *i.e.,* the external messages are sent and the updates are executed on the local database. The behavior of the Web service is thus captured by the runs of the SWS over all possible input sequences.

Main results. We characterize various service models with different classes of SWS's. For each of the classes we establish matching lower and upper bounds on the decision problems, and provide complexity on its composition synthesis.

FSA and transducer abstractions. We denote various classes of SWS's by $\text{SWS}(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$, where $\mathcal{L}_{\text{Msg}}$ and $\mathcal{L}_{\text{Act}}$ are the languages in which transition queries and action synthesis are expressed, respectively. We consider $\mathcal{L}_{\text{Msg}}$ and $\mathcal{L}_{\text{Act}}$ ranging over propositional logic (PL), conjunctive queries (CQ), union of conjunctive queries (UCQ) and first-order logic (FO). For each class we also study its subclass $\text{SWS}_{\text{nr}}(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$ consisting of SWS's that are not recursively defined,*i.e.,* for services that take a bounded number of input messages in a communication session. We examine Web services defined via FSA or transducer abstractions, and show that their behaviors can be specified by various SWS classes. We show that FSA of the Roman model [6] can be expressed in SWS(PL, PL), while data-driven transducers of [2, 12, 13, 29] can be expressed in SWS(FO, FO).

Decision problems. We study the non-emptiness, validation and equivalence problems for various $\text{SWS}(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$ and their nonrecursive subclasses $\text{SWS}_{\text{nr}}(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$. We establish lower and upper bounds on these decision problems, *all matching*, ranging from NP-complete to undecidable. The results are proved using a variety of techniques, including finite model constructions and a wide range of reductions.

Composition synthesis. We also provide complexity bounds on service composition for various $\text{SWS}(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$ and $\text{SWS}_{\text{nr}}(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$ classes. The results are established by, among other things, exploring connections between composition synthesis and (equivalent) query rewriting using views [3, 8, 14, 23]. We show that composition synthesis is infeasible in most of the cases. In light of these undecidability results we identify several decidable cases.

To our knowledge, this work is among the first efforts to study the complexity of composition synthesis for data-driven Web services. Moreover, the results on the decision problems provide a comprehensive picture of the complexity of static analyses for Web services with action synthesis. The notion of SWS's allows us to study various abstractions of Web services in a uniform model. In addition, its capability of expressing synthesis of actions is a step toward specifying practical Web services with aggregations, such as finding a travel package with minimum cost.

Related work. There have been a number of standards for specifying Web services, *e.g.,*WSDL [31], WSCL [30], OWL-S [25], SWFL [27], BEPL [7]. A variety of models have also been developed to characterize services supported by those standards, *e.g.,* [2, 5, 6, 12, 13, 15, 18, 26, 29]. The Roman model [6] specifies a Web service as a deterministic finite

state automaton (DFA). This model is not data-driven: neither queries nor database updates are considered. A notion of guarded automata [15], an extension of (nondeterministic) Mealy machine, is proposed to characterize conversation protocols. Extending the two, Colombo [5] models a service via a guarded nondeterministic finite state automaton (NFA), and incorporates *world states* of local databases. The model of [26] extends an NFA by directing transitions with input and output. Data-driven transducer abstractions are introduced in [2] and extended in [12, 13, 29]. Notably is the powerful model of [12, 13]. It specifies transitions and generates actions via FO queries, and supports infinite states as well as other features such as asynchronous message passing and ideal vs. lossy communication channels, which are not considered in this paper. As remarked earlier, SWS's aim to characterize FSA and transducer abstractions in a uniform framework, and emphasize synthesis of actions generated by services. To our knowledge, except the split-join operator of OWL-S [25], action synthesis does not find counterparts in FSA or transducer models. The connection between these models and SWS's will be explored in Section 3.

A variety of verification problems specified in terms of *e.g.,* linear-time temporal first-order logic, have been studied for transducer abstractions of data-driven services [2, 12, 13, 29]. In particular, (log) equivalence and validation problems are shown undecidable for transducers defined in terms of FO queries, but they become decidable with certain restrictions, *e.g.,* fixed database, bounded inputs, or decidable FO fragments [2, 29]. Validation problems have also been studied for conversation protocols [15, 16]. We revisit equivalence and validation analyses for SWS's in this paper.

While there has been a host of work on Web service composition, few complexity bounds are known. An EXPTIME upper bound is established in [6] on composition synthesis of services defined in the Roman model, by reduction to the satisfiability problem for propositional dynamic logic (PDL). A matching lower bound is given for the Roman model in [24]. The upper bound is proved to hold for an extension of the Roman model with lookahead operations [18]. To our knowledge, the only complexity results on composition synthesis for data-driven services are 2EXPTIME upper bounds given in [5] for special cases of services definable in Colombo, also by reduction to decision problems for PDL.

Another issue for service composition concerns *orchestration* [21]. It is to determine whether available services can be coordinated, without building a mediator, such that when taken together, these services deliver a requested service. We focus on composition synthesis only in this paper.

There is a close connection between SWS composition synthesis of Web services and query rewriting using views. This connection will be studied in Section 5.

Organization. Section 2 defines SWS's. Section 3 characterizes FSA and transducer abstractions with various classes of SWS's. For each of these classes, Sections 4 and 5 study the decision problems and the composition synthesis problem, respectively. Section 6 summarizes the main results and identifies open problems.

2. Synthesized Web Services

In this section we define synthesized Web services (SWS's).

An overview. Given a sequence $\mathcal{I} = I_1, \ldots, I_n$ of input messages, an SWS τ iteratively issues queries to $\mathcal{I}$ and a local database D, and produces an output relation $\mathcal{O}$ denoting actions, *i.e.,* tuples to be inserted into or deleted from D, and external messages to be sent to other services or users.

The run of the SWS τ on $\mathcal{I}$ can be illustrated using an execution tree of depth at most $n+1$. The tree is built top-down, starting from the root. Consider the tree of depth $j \leq n$ constructed so far. At each leaf node v of the tree, τ spawns k children of v, *in parallel*, following a *transition rule*; at each child u_i it issues a query to D and I_j, and stores the result in a *message register* $\mathsf{Msg}(u_i)$. Intuitively, τ processes input I_j at the j-th level of the tree. The expansion of the tree proceeds at u_i except when (a) $j = n+1$ (*i.e.,* all inputs have been processed), or (b) $\mathsf{Msg}(u_i) = \emptyset$ (*i.e.,* the input I_j is not meaningful), or (c) no further expansion is required by the transition rule (*i.e.,* τ reaches a "final" state). If any of the conditions are satisfied, actions are generated at u_i, via a query to D, I_j and $\mathsf{Msg}(u_i)$, and are stored in an *action register* $\mathsf{Act}(u_i)$. The parent v of u_i generates its own actions via a query on the action registers $\mathsf{Act}(u_i)$ of all of its children, following a *synthesis rule*. The process continues until the actions at the root are generated, which are returned as the output $\mathcal{O}$ of the run of τ on $\mathcal{I}$ and D.

In a nutshell, $\mathcal{O}$ denotes the actions of τ in response to a sequence $\mathcal{I}$ of input messages in a communication session. The notion of sessions is proposed in [12], denoting runs beginning at login and ending at logout such that no database updates occur within a session. Here we adopt a more flexible notion: a session is determined by an input sequence $\mathcal{I}$, and the actions of τ are committed at the end of the session. This notion of sessions suffices for the study of the decision problems and composition synthesis of SWS's, for which we consider the behaviors of an SWS in response to all possible input sequences. One can also treat a long (possibly infinite) input sequence as a list of consecutive sessions, by adding a delimiter # to indicate the end of a session, such that actions are committed whenever # is encountered.

Notations. An SWS τ is defined on a relational schema $\mathcal{R}$ for local databases D, an *input schema* R_{in} for input messages $\mathcal{I}$, and an *external schema* R_{out} for output relations $\mathcal{O}$. To simplify the discussion, following [12, 13] we assume that the local database D remains unchanged during a run of τ until the output actions are committed. We also assume an infinite domain $\mathbf{D}$ of *data values*, on which the local databases, input messages and output actions are defined.

To specify an input sequence $\mathcal{I}$ we assume that the input schema R_{in} has a *timestamp* attribute ts of natural numbers, such that $I_j = \{t \mid t \in \mathcal{I} \wedge t[\mathsf{ts}] = j\}$ denotes the j-th input message of $\mathcal{I}$. In other words, $\mathcal{I}$ encodes an input sequence $I_1, \ldots, I_n$. We also assume that R_{in} and R_{out} simply consist of a single relation schema to simplify the presentation.

The SWS τ is defined with a set Q of states to control its behavior. Associated with each state $q \in Q$ are an *internal message register* $\mathsf{Msg}(q)$ and an *action register* $\mathsf{Act}(q)$. We assume *w.l.o.g.* that $\mathsf{Msg}(q)$ and $\mathsf{Act}(q)$ store relations of schemas R_{in} and R_{out}, respectively.

The queries in the transition and synthesis rules of τ are expressed in languages $\mathcal{L}_{\text{Msg}}$ and $\mathcal{L}_{\text{Act}}$, respectively.

SWS's. Putting these together, we define SWS's as follows.

Definition 2.1: Over schemas $\mathcal{R}, R_{in}, R_{out}$, a *synthesized Web service* τ in the class SWS($\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}}$) is defined to be (Q, δ, σ, q_0), where Q is a finite set of *states*, q_0 is the *start state*, and δ, σ are sets of *transition rules* and *synthesis rules*,

respectively, such that for each $q \in Q$, there exist a unique transition rule $\delta(q)$ and a unique synthesis rule $\sigma(q)$:

$$\delta(q): \quad q \rightarrow (q_1, \phi_1(\bar{x}_1)), \ldots, (q_k, \phi_k(\bar{x}_k)).$$
$$\sigma(q): \quad \mathsf{Act}(q) \leftarrow \psi(\bar{y}).$$

Here $q_i \in Q$, ϕ_i is a query in $\mathcal{L}_{\text{Msg}}$ from $\mathcal{R}, R_{in}, \mathsf{Msg}(q)$ to $\mathsf{Msg}(q_i)$; $\psi \in \mathcal{L}_{\text{Act}}$ is from $\mathsf{Act}(q_1), \ldots, \mathsf{Act}(q_k)$ to $\mathsf{Act}(q)$ if $k > 0$, *i.e.,* if the RHS of $\delta(q)$ is nonempty, and it is from $\mathcal{R}, R_{in}, \mathsf{Msg}(q)$ to $\mathsf{Act}(q)$ if $k = 0$. Note that for $k > 0$ we allow ψ to only access previously computed action registers, while at final states ($k = 0$), ψ has access to external information stored in the local database, the current input, and its internal message register. As will be seen in Section 3, SWS's of this form are already expressive enough to express services supported by previous models. We assume that q_0 does not appear in the RHS of any rule. We refer to q as the *parent* state of q_i, and q_i as a *successor state* of q. □

Note that $\delta(q)$ is parametrized with instances of $\mathcal{R}, R_{in}$ and $\mathsf{Msg}(q)$, *i.e.,* δ is a finite representation of infinitely many transition rules; similarly for synthesis rules σ. We remark that the synthesis rules allow to model alternation in Web services, for which the need is advocated in Section 1.

Example 2.1: We define an SWS $\tau_1 = (Q, \delta, \sigma, q_0)$ to specify the service described in Example 1.1. Assume that R_{in} consists of attributes specifying user requirements, and an additional attribute tag with values of a, h, t, c, indicating whether an R_{in} tuple is about airfare, hotel, ticket or car rental. Also assume that an R_{out} tuple is of the form $R_{out}(\bar{x}_a, \bar{x}_h, \bar{x}_t, \bar{x}_c)$, with attribute lists $\bar{x}_a, \bar{x}_h, \bar{x}_t, \bar{x}_c$ for airfare, hotel, tickets and cars, respectively. Moreover, assume that $\mathcal{R}$ consists of relations R_a, R_h, R_t, R_c containing information about airfares, hotels, tickets or cars, respectively. The set Q of states includes q_0, q_a, q_h, q_c, q_t, and their transition and synthesis rules δ and σ are given as follows:

$q_0 \rightarrow (q_a, \phi_a), (q_h, \phi_h), (q_t, \phi_t), (q_c, \phi_c).$
$\mathsf{Act}(q_0) \leftarrow \psi_0(\bar{y}),$ where

$\phi_a(\mathsf{tag}, \bar{x}) = R_{in}(\mathsf{tag}, \bar{x}) \wedge \mathsf{tag} = a$ /* similarly for ϕ_h, ϕ_t, ϕ_c*/

$\psi_0(\bar{x}_a, \bar{x}_h, \bar{x}_t, \bar{x}_c) = \mathsf{Act}(q_a)(\bar{x}_a, \_, \_, \_)$ /* _ for don't-care */
$\wedge\ \mathsf{Act}(q_h)(\_, \bar{x}_h, \_, \_) \wedge (\mathsf{Act}(q_t)(\_, \_, \bar{x}_t, \bar{x}_c)$
$\vee\ \neg\exists \bar{x} \mathsf{Act}(q_t)(\bar{x}) \wedge \mathsf{Act}(q_c)(\_, \_, \bar{x}_t, \bar{x}_c))$

$q_a \rightarrow .$ $\quad \mathsf{Act}(q_a) \leftarrow \psi_a(\bar{y}).$ /* similarly for q_h, q_t, q_c */

Here ψ_a (not shown) is an FO query that accesses $\mathsf{Msg}(q_a)$ and R_a, and returns airfares that satisfy the user's requirement. The register $\mathsf{Msg}(q_a)$ is computed by ϕ_a that simply selects input tuples for airfare; similarly for ϕ_h, ϕ_t and ϕ_c. Synthesis queries ψ_h, ψ_c and ψ_t (not shown) instantiate $\mathsf{Act}(q_h)$, $\mathsf{Act}(q_c)$ and $\mathsf{Act}(q_t)$ with hotel room ($\bar{x}_h$), car rental ($\bar{x}_c$) and ticket sale ($\bar{x}_t$) information, respectively. The output $\mathsf{Act}(q_0)$ is deterministically computed by ψ_0, which synthesizes actions from successor states of q_0, in favor of sale tickets, *i.e.,* it copies $\mathsf{Act}(q_t)$ if it is nonempty, and takes $\mathsf{Act}(q_c)$ otherwise. It is nonempty only if airfares, hotels *and* either sale tickets *or* rental cars are found, as required.

One can readily extend τ_1 and define τ_2 to accommodate repeated user inquiries $\mathcal{I}$ for, *e.g.,* airfare, as follows:

$q_a \rightarrow (q_a, \phi_a), (q_f, \phi_a),$ $\mathsf{Act}(q_a) \leftarrow \psi'_a(\bar{y})$, where

$\psi'_a(\bar{x}_a, \bar{x}_h, \bar{x}_t, \bar{x}_c) = \mathsf{Act}(q_a)(\bar{x}_a, \bar{x}_h, \bar{x}_t, \bar{x}_c)$
$\vee\ \neg\exists \bar{x} \mathsf{Act}(q_a)(\bar{x}) \wedge \mathsf{Act}(q_f)(\bar{x}_a, \bar{x}_h, \bar{x}_t, \bar{x}_c)$

$q_f \rightarrow .$ $\quad \mathsf{Act}(q_f) \leftarrow \psi_f(\bar{y}),$

where ϕ_a selects user requirements for airfare from input message I_j in $\mathcal{I}$ as above, and ψ_f finds airfares based on user inquiry I_j by querying $\mathsf{Msg}(q_a)$ and R_a, similar to ψ_a. The synthesis query ψ'_a is in favor of actions $\mathsf{Act}(q_a)$ that are in turn synthesized at the "recursive" state q_a; in other words, it returns airfares that meet the latest user requirement I_j in $\mathcal{I}$, if there is any, discarding earlier inquiry results. □

Runs of SWS's. Following [5, 6], we specify the *run* of an SWS τ on a local database D and an input message sequence $\mathcal{I}$ using a notion of *execution trees*. Each node v in an execution tree is labeled with a state $q \in Q$, a timestamp j, a message register $\mathsf{Msg}(v)$ and an action register $\mathsf{Act}(v)$. Every step of the run rewrites an execution tree to another, following a step relation $\Rightarrow_{(\tau,D,\mathcal{I})}$.

The run starts from the root node r of an execution tree ξ labeled with q_0 and $j = 0$, carrying $\mathsf{Msg}(r) = \emptyset$ whereas $\mathsf{Act}(r) = \perp$ (undefined). For two execution trees ξ and ξ', $\xi \Rightarrow_{(\tau,D,\mathcal{I})} \xi'$ if one of the following conditions holds.

Generating. If there is a leaf node v of ξ labeled q, $j, \mathsf{Msg}(v)$ and $\mathsf{Act}(v) = \perp$, ξ' is obtained from ξ as follows. Assume that $\mathcal{I} = I_1, \ldots, I_n$, and that the rules $\delta(q), \sigma(q)$ are

$$q \rightarrow (q_1, \phi_1(\bar{x}_1)), \ldots, (q_k, \phi_k(\bar{x}_k)). \quad \mathsf{Act}(q) \leftarrow \psi(\bar{y}).$$

(1) If either $j > n$ or $\mathsf{Msg}(v)$ is empty, then ξ' is obtained from ξ by setting $\mathsf{Act}(v) = \emptyset$. A special case is when $q = q_0$, at the root r: even if $\mathsf{Msg}(r)$ is empty, we allow the construction to proceed as follows if $\mathcal{I}$ is nonempty.

(2) Otherwise, if $k > 0$, then ξ' is obtained from ξ by spawning k children $u_1, \ldots, u_k$ for v, in parallel. More specifically, for each $i \in [1, k]$, a distinct node u_i is created as the i-th child of v, labeled q_i and $j+1$ such that $\mathsf{Msg}(u_i)$ carries $\phi_i(D, I_j, \mathsf{Msg}(v))$ but leaving $\mathsf{Act}(u_i) = \perp$. Here $\phi_i(D, I_j, \mathsf{Msg}(v)) = \{\bar{d} \mid (D, I_j, \mathsf{Msg}(v)) \models \phi_i(\bar{d})\}$.

Gathering. Action register $\mathsf{Act}(v)$ is populated as follows.

(3) If $k = 0$, *i.e.,* the RHS of $\delta(q)$ is empty and v is a leaf node, then ξ' is obtained from ξ by letting $\mathsf{Act}(v) = \psi(D, I_j, \mathsf{Msg}(v))$, where $\psi(\bar{y})$ is the query in the synthesis rule $\sigma(q)$, producing actions by accessing $\mathsf{Msg}(v), I_j$ and D.

(4) Otherwise, if for all the children $u_1, \ldots, u_k$ of v, $\mathsf{Act}(u_i) \neq \perp$, then ξ' is obtained from ξ by setting $\mathsf{Act}(v) = \psi(\mathsf{Act}(u_1), \ldots, \mathsf{Act}(u_k))$, *i.e.,* the query ψ synthesizes actions from $\mathsf{Act}(u_i)$. In other words, the synthesis is halted at v until $\mathsf{Act}(u_i)$ is available for all $i \in [1, k]$.

Observe that an execution tree stops spawning new nodes at v if (a) the input message $\mathcal{I}$ is entirely consumed (*i.e.,* $j > n$), (b) it receives an empty internal message $\mathsf{Msg}(v)$ (if $q \neq q_0$), or (c) v is at a "final state" indicated by the transition rule (with an empty RHS). While the tree is generated top-down, action gathering and synthesis are conducted bottom-up. The run takes one sweep: each node is accessed at most twice, for node generation and action gathering, respectively.

Output. Denote by $\Rightarrow^*_{(\tau,D,\mathcal{I})}$ the reflexive and transitive closure of $\Rightarrow_{(\tau,D,\mathcal{I})}$. The *result* of the *run* of the SWS τ on $(D, \mathcal{I})$ is an execution tree ξ such that $r \Rightarrow^* \xi$, where r is the root labeled q_0 and timestamp 1, and there exists no node v in ξ with $\mathsf{Act}(v) = \perp$. The *output* of the τ-run, denoted by $\tau(D, \mathcal{I})$, is the content of the action register $\mathsf{Act}(r)$.

Example 2.2: Recall the SWS τ_1 defined in Example 2.1. Given an instance $D = (I_a, I_h, I_c, I_t)$ of the local database

schema $\mathcal{R}$ and a sequence of input messages $\mathcal{I} = I_1, \ldots, I_n$, the result $\tau_1(D, \mathcal{I})$ of the run of τ_1 is a relation that is nonempty iff the conditions 1–3 given in Example 1.1 are all satisfied. The execution tree of the run is constructed as follows, starting from the root node r. Four children v_a, v_h, v_t, v_c of r are created in parallel. The node v_a is a leaf of the tree labeled with $\mathsf{ts} = 1$, state q_a and two registers. Its message register $\mathsf{Msg}(v_a)$ is set by $\phi_a(I_1)$ with input tuples for airfare, followed by the instantiation of the action register $\mathsf{Act}(v_a)$ by $\psi_a(I_a, \mathsf{Msg}(v_a))$ with tuples for booking flights that meet the user request, if any; similarly for v_h, v_t and v_c. As soon as values are assigned to $\mathsf{Act}(v_a)$, $\mathsf{Act}(v_h)$, $\mathsf{Act}(v_t)$ and $\mathsf{Act}(v_c)$, the synthesis query $\psi_0(\mathsf{Act}(v_a), \mathsf{Act}(v_h), \mathsf{Act}(v_c), \mathsf{Act}(v_t))$ populates $\mathsf{Act}(r)$, as described in Example 2.1. The content of register $\mathsf{Act}(r)$ is the output. Here $I_2, \ldots, I_n$ are *not* consumed by τ_1; *i.e.,* it suffices for τ_1 to produce output when $\mathcal{I}$ consists of a single input message.

Note that the synthesis query ψ_0 makes a deterministic decision between $\mathsf{Act}(v_c)$ and $\mathsf{Act}(v_t)$, and it specifies a conjunctive condition on reservations of flight, hotel and local arrangement (either car rental or tickets). To simplify the discussion, while user requests for flight, hotel, tickets and cars are usually given in different input relations, we encode them in a single relation.

The SWS τ_2 behaves similarly, except that it can accept an input sequence $\mathcal{I} = I_1, \ldots, I_n$ of an unbounded length n. Here below v_a is a chain of node pairs (v_j, f_j), which indicate the processing of user inquiry I_j for airfare, for $j \in [2, n]$. The synthesis query ψ_a assures that the nonempty $\mathsf{Act}(f_j)$ with the largest j is used to populate $\mathsf{Act}(v_a)$. □

SWS classes. To characterize prior models developed for specifying Web services, we study various SWS($\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}}$) classes, where $\mathcal{L}_{\text{Msg}}$ and $\mathcal{L}_{\text{Act}}$ are the languages for expressing queries embedded in transition and synthesis rules, respectively. As suggested by condition (3) in the definition of execution trees, we assume $\mathcal{L}_{\text{Act}}$ to be at least as expressive as $\mathcal{L}_{\text{Msg}}$, but also consider special cases when $\mathcal{L}_{\text{Act}}$ is not as powerful as $\mathcal{L}_{\text{Msg}}$. In particular, we study the following classes: SWS(FO, FO), SWS(CQ, UCQ) and SWS(PL, PL), for first-order logic (FO), conjunctive queries (CQ), union of conjunctive queries (UCQ), and propositional logic (PL), with '=' and inequality '≠' in CQ and UCQ. For example, τ_1, τ_2 given in Example 2.1 are in SWS(FO, FO). We do not consider SWS(CQ, CQ) since not many interesting services can be specified in the absence of union in synthesis rules.

As will be seen shortly, Web services of the Roman model [6] can be modeled as a special case of SWS(PL, PL). Unlike CQ, UCQ or FO, a PL-formula is defined over propositional variables. An SWS τ in SWS(PL, PL) aims to specify a service that is *not* data-driven, *i.e.,* on empty local databases. An input message I_j of R_{in} for τ is a truth assignment, represented as a set of propositional variables such that a variable X is *true* iff $X \in I_j$. Note that a sequence $\mathcal{I}$ of R_{in} instances can encode a string over an alphabet found in FSA abstractions of Web services. Message and action registers, as well as output $\mathcal{O}$ of τ consist of a single truth value *true* or *false*, along the same lines as FSA abstractions.

For each SWS($\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}}$), we also study its subclass $\text{SWS}_{\mathsf{nr}}(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$ consisting of all *nonrecursive* SWS's, defined in terms of a notion of dependency graphs as follows. The *dependency graph* G_τ of an SWS τ is a graph in which each $q \in Q$ is a distinct node, and there is an edge from q to q_i iff q_i is in the RHS of the transition rule of q. An SWS τ is said to be *recursive* if the graph G_τ is cyclic. For instance, τ_2 of Example 2.1 is recursive, but τ_1 is nonrecursive. Nonrecursive SWS's aim to model services that respond to a user input within a bounded number of computation steps. As will be seen in Section 5, a nonrecursive SWS may be repeatedly invoked by a composite service (mediator) for an unbounded number of times to process various inputs.

3. Specifying Web Services as SWS's

In this section we show that the behaviors of Web services defined in terms of FSA or transducer abstractions can be specified by various SWS classes. We consider two representative models: the Roman model [6] and the peer model of [13]. The Roman model is an FSA abstraction for specifying non-data-driven services, and peers of [13] are a transducer abstraction for specifying data-driven Web services.

For each of these models, we identify an SWS class $\mathcal{S}$ and give two functions f_τ and f_I, such that (a) for each service ω defined in the model, $f_\tau(\omega)$ is an SWS τ in $\mathcal{S}$, and (b) for any input sequence $\mathcal{I}$ of ω, $f_I(\mathcal{I})$ is an input sequence $\mathcal{I}$ for τ such that for any local database D, $\tau(D, \mathcal{I})$ yields the same output as ω on $\mathcal{I}$ and D.

The Roman model [6]. A service in the model is expressed as a DFA (or an NFA for a composite service), in which the alphabet is interpreted as a set of actions. It takes a string over the alphabet and returns true if the string leads to a final state, indicating that the service legally terminates.

One can define a function f_τ that, given such a DFA ω, derives τ in SWS(PL, PL). The SWS τ includes all the states of ω and an additional q_f. The transition rule for each state q from ω collects all transition rules for q in ω: $q \to (q_1, \phi_1), \ldots, (q_k, \phi_k)$, $[(q_f, \phi_f)]$, where $q \to q_i$ is a transition in ω upon receiving a letter a_i, and ϕ_i simply checks whether the input message is a_i; moreover, if q is a final state, it has q_f as a successor state, for which ϕ_f checks whether the input is a special delimiter #. The synthesis rule for q_f is $\mathsf{Act}(q_f) \leftarrow \mathsf{Msg}(q_f)$, while the synthesis query for q is the disjunction of $\mathsf{Act}(q_i)$ of all q_i. The function f_I is defined such that given a string $\mathcal{I}$ over the alphabet, it derives $\mathcal{I}$ that augments each letter with its index in $\mathcal{I}$ and adds the delimiter # to the end of $\mathcal{I}$. One can easily see that for any $\mathcal{I}$, $\omega(\mathcal{I}) = \tau(D, \mathcal{I})$, where D is an empty local database. Note that τ only needs disjunction in its synthesis rules.

One can also define τ in SWS(CQ,UCQ) that returns the empty set if $\mathcal{I}$ is not accepted by ω, and the same $\mathcal{I}$ otherwise. That is, τ defers the commitment of the actions until it confirms that all actions in $\mathcal{I}$ can be legally conducted.

The peer model [13]. A peer of [13] is characterized by (1) a local database D that is fixed during execution; (2) a set $\mathcal{S}$ of *state* relations keeping track of updates to D; (3) a set $\mathcal{I}$ of *user input* relations; (4) a set $\mathcal{A}$ of *action* relations; (5) a set of in-queues $\mathcal{Q}_i$ and out-queues $\mathcal{Q}_o$, in which elements are *input* and *output message* relations from and to other services, respectively; and (6) a set of rules defined as FO queries on $D, \mathcal{S}, \mathcal{Q}_i$, current and previous $\mathcal{I}$. At each step the peer uses the rules to produce actions, updates and output messages to be included in $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{Q}_o$, respectively.

A function f_τ can be defined that derives from a peer ω an SWS τ in SWS(FO, FO). The SWS τ encodes $\mathcal{I}$ and

	Non-emptiness	Validation	Equivalence
SWS_{nr}(FO, FO)	undecidable	undecidable	undecidable
SWS(CQ, UCQ)	EXPTIME-complete	undecidable	undecidable
SWS_{nr}(CQ, UCQ)	PSPACE-complete	NEXPTIME-complete	coNEXPTIME-complete
SWS(PL, PL)	PSPACE-complete	PSPACE-complete	PSPACE-complete
SWS_{nr}(PL, PL)	NP-complete	NP-complete	coNP-complete

Table 1: Complexity of decision problems

$\mathcal{Q}_i$ in terms of a single input schema R_{in}, and $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{Q}_o$ with a single R_{out}. It is recursively defined, with three states q_0, q_s, q_f. The transition rule for q_f has an empty RHS, while for q_0, q_s, they are $q_0 \rightarrow (q_s, \phi), (q_f, \phi)$ and $q_s \rightarrow (q_s, \phi), (q_f, \phi)$, where ϕ is an FO query that combines all queries in the rules of ω. The synthesis rule for q_f copies the query result to the output; the synthesis rules for q_0 and q_s take unions of actions generated by their successor states.

The peer ω produces an output at each step of execution. To cope with this interactive behavior, we define function f_I such that for each input sequence $\bar{I} = I_1, \ldots, I_n$ for ω, it returns $\mathcal{I} = I_1, \#, I_1, I_2, \#, \ldots, \#, I_1, \ldots, I_n, \#$. The SWS τ produces an output whenever a delimiter $\#$ is encountered. Then for any $\bar{I}$ and any database D, $\tau(D, \mathcal{I})$ yields the same output as $\omega(\bar{I}, D)$ at each step $j \in [1, n]$.

Other models. As observed in [13], services supported by the Colombo model [5] or expressed as guarded automata of [15] can also be expressed as peers of [13]. As a result, one can also use SWS(FO, FO) to study the behaviors of the Colombo services and guarded automata [15].

4. Complexity of Synthesized Web Services

In this section we investigate static analyses of SWS's. For a class $\text{SWS}(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$, we study the following decision problems. As remarked in Section 1, these problems are of both theoretical and practical interest.

• The *non-emptiness problem* is to determine, given an SWS τ in the class defined over a database schema $\mathcal{R}$ and an input schema R_{in}, whether there exist an instance D of $\mathcal{R}$ and a sequence $\mathcal{I}$ of R_{in} instances such that $\tau(D, \mathcal{I})$ is nonempty. That is, whether or not τ can generate actions at all.

• The *validation problem* is to determine, given τ in the class over $\mathcal{R}$ and R_{in}, and an instance $\mathcal{O}$ of its external schema R_{out}, whether or not there exist an instance D of $\mathcal{R}$ and a sequence $\mathcal{I}$ of R_{in} instances such that $\tau(D, \mathcal{I}) = \mathcal{O}$. That is, whether τ is capable of producing the desired actions.

• The *equivalence problem* is to determine, given two SWS's τ_1 and τ_2 in this class, both defined over the same $\mathcal{R}$, R_{in} and R_{out}, whether or not $\tau_1(D, \mathcal{I}) = \tau_2(D, \mathcal{I})$ for all instances D of $\mathcal{R}$ and sequences $\mathcal{I}$ of R_{in} instances. With a cost model, the equivalence analysis helps users decide what service to choose from a set of available services.

We establish matching complexity bounds on these problems for the SWS classes given in Section 2, recursive or not. The results are summarized in Table 1. They tell us the following. First, for nonrecursive SWS_{nr}(FO, FO) all these problems are already beyond reach. Second, while compile-time equivalence and validation analyses are still infeasible for SWS(CQ, UCQ), they are decidable for SWS_{nr}(CQ, UCQ) in coNEXPTIME and NEXPTIME, respectively. That is, the absence of recursion makes our lives easier, although still not easy enough. Third, the equivalence problem for SWS(PL, PL) has the same complexity as its NFA and AFA counterparts, and its emptiness analysis is no harder than that for AFA.

Theorem 4.1: The non-emptiness, validation and equivalence problems are

1. undecidable for SWS(FO, FO) and SWS_{nr}(FO, FO);
2. EXPTIME-complete, undecidable and undecidable for SWS(CQ, UCQ); but for SWS_{nr}(CQ, UCQ) they become PSPACE-complete, NEXPTIME-complete and coNEXPTIME-complete, respectively;
3. PSPACE-complete for SWS(PL,PL); for SWS_{nr}(PL,PL) these problems become NP-complete, NP-complete and coNP-complete, respectively. □

PROOF. (1) For SWS_{nr}(FO, FO), the undecidability of these problems is proved by reduction from the satisfiability problem for FO queries, which is undecidable (cf. [1]).

(2) For SWS(CQ, UCQ), the validation and equivalence problems are shown undecidable by reduction from the non-emptiness problem for deterministic finite 2-head machines, which is undecidable [28]. The lower bound on non-emptiness analysis is by reduction from the problem for deciding whether a single ground fact, single rule datalog program (sirup) accepts a goal, an EXPTIME-complete problem [19]. Its upper bound is by reduction to the emptiness problem for a form of tree automata [8] that characterize the execution trees of SWS's; while the latter can be checked in PTIME, the reduction takes EXPTIME.

For SWS_{nr}(CQ, UCQ), we show the lower bound on non-emptiness analysis by reduction from Q3SAT, a PSPACE-complete problem (cf. [17]), and the upper bound by giving a PSPACE checking algorithm. The lower bound on the equivalence problem is by reduction from the problem for checking whether a nondeterministic Turing machine (NTM) halts in exponential steps on a given input. Similarly we prove the lower bound on validation analysis. We develop a coNEXPTIME checking algorithm for SWS_{nr}(CQ, UCQ) equivalence, by extending the containment algorithm of [22] for nonrecursive datalog with inequality. We show that validation analysis is in NEXPTIME by establishing a small model property: given an SWS τ and output $\mathcal{O}$, if there exist a database D and an input sequence $\mathcal{I}$ such that $\tau(D, \mathcal{I}) = \mathcal{O}$, then there exist such D and $\mathcal{I}$ bounded by an exponential in the size of $\mathcal{O}$ and τ. Since evaluating τ on such a $(D, \mathcal{I})$ and checking whether $\tau(D, \mathcal{I}) = \mathcal{O}$ are both in EXPTIME, a NEXPTIME algorithm is immediate.

(3) For SWS(PL, PL), the non-emptiness problem coincides with the validation problem, and is PTIME-equivalent to the complement of the equivalence problem. The lower bound on the non-emptiness problem follows from the PSPACE lower bound on the emptiness problem for AFA [32], which can be expressed in SWS(PL, PL), in PTIME. For the upper bound, one can check non-emptiness in PSPACE along the same lines as AFA non-emptiness checking.

For SWS_{nr}(PL, PL), the lower bound is proved by reduction from SAT, which is NP-complete (cf. [17]), and the upper

bound by noting that the non-emptiness checking algorithm for SWS(PL, PL) is in NP on nonrecursive SWS's. □

Special cases. For services defined as NFA (resp. DFA) in, *e.g.,* the Roman model, the complexity on the equivalence and non-emptiness problems is known to be PSPACE-complete (resp. NLOGSPACE-complete) and NLOGSPACE-complete, respectively [32]. Note that the validation problem studied here is quite different from the membership problem for NFA (resp. DFA). We remark that for SWS(PL, PL) and for services defined as NFA (resp. DFA), the validation problem coincides with the non-emptiness problem. In contrast, it is entirely different from non-emptiness for data-driven services in SWS(FO, FO) and SWS(CQ, UCQ).

5. Composition Synthesis

We next focus on composition synthesis of SWS's. We first formalize the notion of SWS mediators, and then provide complexity bounds on composition synthesis for various classes of SWS's identified in Section 2. Our main conclusion is that the composition synthesis problem is nontrivial: it is undecidable or highly intractable for data-driven or recursive SWS's. In light of the undecidability results we identify decidable cases of data-driven or recursive SWS's.

5.1 SWS Mediators and Composition

A mediator coordinates available services by routing the output of one service to the input of another, in order to deliver a requested service by invoking available services as component services [7, 5]. It receives and redirects messages, but does not directly access local databases. An SWS mediator also synthesizes actions produced by component services.

We parametrize SWS mediators with languages $\mathcal{L}_{\text{Act}}$ for expressing action synthesis. For a fixed $\mathcal{L}_{\text{Act}}$, we define a class of mediators, denoted by $\text{MDT}(\mathcal{L}_{\text{Act}})$, in the same way as SWS's except that component services are embedded in transition rules and are treated as "oracle queries".

Definition 5.1: Over a set S of available SWS's defined on schemas $\mathcal{R}, R_{in}, R_{out}$, an SWS *mediator* in $\text{MDT}(\mathcal{L}_{\text{Act}})$ is defined to be $\pi = (Q, \delta, \sigma, q_0)$, where Q and q_0 are as given in Definition 2.1, and δ is a set of *transition rules* such that for each $q \in Q$, there exists a unique rule of the form

$$q \quad \rightarrow \quad (q_1, \mathsf{eval}(\tau_1)), \ldots, (q_k, \mathsf{eval}(\tau_k)).$$

Here $q_i \in Q$, $\tau_i \in S$, where τ_i is referred to as a *component service* of π, and $\mathsf{eval}(\tau_i)$ is an evaluation operator whose semantics will be explained below. The synthesis rule $\sigma(q)$ is defined via a query $\psi \in \mathcal{L}_{\text{Act}}$ as in SWS's, except that if $k = 0$, ψ is from $\mathsf{Msg}(q)$ to $\mathsf{Act}(q)$ without accessing database and input. □

To simplify the presentation we assume that SWS's in S are defined over the same input, external and database schemas $\mathcal{R}, R_{in}$ and R_{out}. This does not lose generality for composition synthesis since one can take $\mathcal{R}$ as the collection of local database schemas of all component SWS's in S, and unify R_{in} (resp. R_{out}) by *e.g.,* taking outer union of input (resp. external) relations of SWS's in S.

Runs of mediators. Like SWS's, a mediator π takes as input an instance D of the database schema $\mathcal{R}$ and a sequence $\mathcal{I}$ of instances of the input schema R_{in}; the run of π on $(D, \mathcal{I})$ returns an instance of the external schema R_{out}. A step relation $\Rightarrow_{(\pi,D,\mathcal{I})}$ is defined for runs of π as for SWS's.

There are, however, subtle differences in the run of π on D and $\mathcal{I}$. More specifically, for execution trees ξ and ξ', cases (2) and (3) of the transition $\xi \Rightarrow_{(\pi,D,\mathcal{I})} \xi'$ described in Section 2 are now modified as follows. Consider a leaf node v in ξ, labeled with q, j, $\mathsf{Msg}(v)$ and $\mathsf{Act}(v) = \bot$. Assume that $\mathcal{I} = I_1, \ldots, I_n$, and that $\delta(q)$ and $\sigma(q)$ are

$$q \ \rightarrow \ (q_1, \mathsf{eval}(\tau_1)), \ldots, (q_k, \mathsf{eval}(\tau_k)), \quad \mathsf{Act}(q) \ \leftarrow \ \psi(\bar{y}).$$

(2) If $k > 0$, then as before, ξ' is obtained from ξ by spawning k children $u_1, \ldots, u_k$ for v, in parallel, such that for each $i \in [1, k]$, a distinct node u_i is created as the i-th child of v, labeled with q_i and $\mathsf{Act}(u_i) = \bot$. In contrast to its SWS counterpart, here $\mathsf{Msg}(u_i)$ is instantiated with the result of $\mathsf{eval}(\tau_i)$. That is, $\mathsf{eval}(\tau_i)$ is equal to $\tau_i(D, \mathcal{I}^j)$ of the component SWS τ_i on D and $\mathcal{I}^j$, where $\mathcal{I}^j = I_j, \ldots, I_n$. That is, τ_i operates on $I_j, \ldots, I_n$, runs to completion, and its output is used as the internal message of u_i. Moreover, in the run of τ_i, the message register of the start state of τ_i is instantiated with $\mathsf{Msg}(v)$. Furthermore, u_i is labeled with timestamp $l_i + 1$, where l_i is the *maximum* timestamp in the result execution tree of the run of τ_i on $(D, \mathcal{I}^j)$. In other words, I_{l_i+1} indicates the first input message that has *not* been "consumed" by the run of τ_i.

(3) If $k = 0$, then the synthesis query ψ can access $\mathsf{Msg}(v)$ but neither the database D nor the input message I_j,*i.e.,* ξ' is obtained from ξ by letting $\mathsf{Act}(v) = \psi(\mathsf{Msg}(v))$.

The rest of the run is defined as for SWS's, as described in Section 2. The commitment of actions produced by the component services is deferred to the end of the run of π. The output of the run of π on $(D, \mathcal{I})$ is denoted by $\pi(D, \mathcal{I})$.

We can compare SWS's and mediators as follows. Consider a mediator π over a set of component SWS's defined on schemas $\mathcal{R}, R_{in}, R_{out}$, and an SWS τ defined on the same $\mathcal{R}$, R_{in}, R_{out}. We say that π and τ are *equivalent*, denoted by $\pi \equiv \tau$, if $\pi(D, \mathcal{I}) = \tau(D, \mathcal{I})$ for any instance D of $\mathcal{R}$ and any input sequence $\mathcal{I}$ of R_{in} instances.

We study $\text{MDT}(\mathcal{L}_{\text{Act}})$ for $\mathcal{L}_{\text{Act}}$ ranging over PL, UCQ and FO. We denote by $\text{MDT}_{\mathsf{nr}}(\mathcal{L}_{\text{Act}})$ the class of nonrecursive mediators in $\text{MDT}(\mathcal{L}_{\text{Act}})$ defined in the same way as their nonrecursive SWS counterparts. Note that component SWS's embedded in a nonrecursive mediator may be recursive.

Example 5.1: Recall SWS τ_1 and $R_{out}(\bar{x}_a, \bar{x}_h, \bar{x}_t, \bar{x}_c)$ from Example 2.1. Suppose that we are given a set S of component services consisting of τ_a for flight reservations, τ_{hc} for reserving *both* hotel rooms and rental cars, and τ_{ht} for reserving *both* hotel and Disney tickets. One can define a mediator $\pi_1 = (Q_1, \delta_1, \sigma_1, q_1)$, where $Q_1 = \{q_1, q_a, q_{hc}, q_{ht}\}$, and transition and synthesis rules δ_1 and σ_1 are as follows.

$q_1 \rightarrow (q_a, \mathsf{eval}(\tau_a)), (q_{hc}, \mathsf{eval}(\tau_{hc})), (q_{ht}, \mathsf{eval}(\tau_{ht}))$,
$\mathsf{Act}(q_1) \leftarrow \psi_1(\bar{y})$, where
$\psi_1(\bar{x}_a, \bar{x}_h, \bar{x}_t, \bar{x}_c) = \mathsf{Act}(q_a)(\bar{x}_a, \_, \_, \_) \wedge$
$(\mathsf{Act}(q_{ht})(\_, \bar{x}_h, \bar{x}_t, \bar{x}_c) \vee$
$\neg\exists \bar{y}\ \mathsf{Act}(q_{ht})(\bar{y}) \ \wedge \ \mathsf{Act}(q_{hc})(\_, \bar{x}_h, \bar{x}_t, \bar{x}_c))$

$q_a \rightarrow .$ $\quad \mathsf{Act}(q_a) \leftarrow \psi_a(\bar{y})$ /* similarly for q_{hc} and q_{ht} */
where $\psi_a(\bar{x}_a, \_, \_, \_) = \mathsf{Msg}(q_a)(\bar{x}_a, \_, \_, \_)$

Here τ_a is invoked by q_1 to find airfares, and its output is used to instantiate $\mathsf{Act}(q_a)$; similarly for τ_{hc} and τ_{ht}. The outputs of these component services are synthesized by ψ_1, which is in favor of Disney tickets. One can verify that τ_1 and π_1 are equivalent provided that (a) τ_a finds all flights that ϕ_a of Example 2.1 can find, (b) both τ_{hc} and τ_{ht} find all hotel rooms that ϕ_a finds, and (c) τ_{ht} (resp. τ_{hc}) finds all

tickets (resp. cars) that ϕ_t (resp. ϕ_c) finds, and vice versa. This mediator is in the class MDT_{nr}(FO). □

Composition synthesis. In the reminder of the section we study the following family of *composition synthesis problems*, referred to $\text{CP}(\mathcal{G}, \mathcal{M}, \mathcal{C})$, where $\mathcal{G}, \mathcal{M}$ and $\mathcal{C}$ are called the *goal* (user-requested) service, *mediator* and *component* (available) service classes, respectively. Here $\mathcal{G}$ and $\mathcal{C}$ are either $\text{SWS}(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$ or $\text{SWS}_{\text{nr}}(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$, and $\mathcal{M}$ is either $\text{MDT}(\mathcal{L}_{\text{Act}})$ or $\text{MDT}_{\text{nr}}(\mathcal{L}_{\text{Act}})$. We consider $(\mathcal{L}_{\text{Msg}}, \mathcal{L}_{\text{Act}})$ ranging over (PL, PL), (CQ, UCQ) and (FO, FO).

Given a goal SWS $\tau \in \mathcal{G}$ and a finite set $S \subset \mathcal{C}$ of component services, all defined over the same schemas $\mathcal{R}$, R_{in} and R_{out}, $\text{CP}(\mathcal{G}, \mathcal{M}, \mathcal{C})$ is to determine whether or not there exists a mediator $\pi \in \mathcal{M}$ over S such that $\pi \equiv \tau$.

5.2 Complexity of Composition Synthesis

There is a connection between SWS composition and query rewriting using views. For a fixed relational query language $\mathcal{L}$, the problem of equivalent query rewriting using views (cf. [20]) is to determine, given a query $\psi \in \mathcal{L}$ and a set of view definitions $\mathcal{V}$ in $\mathcal{L}$, whether or not there exists an *equivalent rewriting* of ψ, *i.e.,* a query $\psi' \in \mathcal{L}$ such that ψ and ψ' are equivalent, and ψ' accesses only relations in $\mathcal{V}$. A query ψ' is referred to as a *maximally-contained* rewriting of ψ if ψ' is contained in ψ, ψ' accesses only relations in $\mathcal{V}$, and there exists no $\psi_1 \in \mathcal{L}$ such that ψ' is contained in ψ_1, ψ_1 is contained in ψ, and ψ_1 accesses only relations in $\mathcal{V}$. For $\mathcal{L}$ ranging over datalog, UCQ, CQ and regular path queries, the problems of finding equivalent and maximally-contained rewritings have been studied (*e.g.,* [8, 3, 14, 23]). The connection between these problems and SWS composition is evident if we treat the goal service $\tau \in \mathcal{G}$ and mediator $\pi \in \mathcal{M}$ as queries, and component services $S \subset \mathcal{C}$ as views.

We study SWS composition synthesis by capitalizing on this connection. It is, however, nontrivial to apply prior results on query rewriting to SWS composition. First, SWS's may not be expressible in any well-studied query languages. Second, for SWS's that can be expressed in a query language, the rewriting problem for that language may not be settled. For example, although SWS's in SWS_{nr}(CQ, UCQ) can be converted to UCQ queries with inequality, to our knowledge, the complexity on equivalent query rewriting for UCQ with inequality is not yet known (some special cases are considered in [4]). Our results on SWS composition may shed light on the study of equivalent query rewriting.

For SWS composition synthesis we show the following. First, it is infeasible already for nonrecursive SWS's and mediators defined in terms of FO. Second, it remains undecidable for goal services in SWS(CQ, UCQ) even when either mediators or components are nonrecursive, but it becomes decidable when all of them are nonrecursive. Third, for SWS(PL, PL) composition synthesis involves arbitrary combinations of concatenation, intersection and complementation of SWS's, and is more intriguing than the rewriting problems studied for regular path queries in previous work (*e.g.,* [8]). While we do not yet know whether it is decidable in general, we show that it is decidable when either goal SWS's or both mediators and components are nonrecursive.

Theorem 5.1: Composition synthesis $\text{CP}(\mathcal{G}, \mathcal{M}, \mathcal{C})$ is

1. undecidable when $\mathcal{G}, \mathcal{C}$ are SWS(FO, FO) and $\mathcal{M}$ is MDT(FO), even when $\mathcal{G}, \mathcal{M}, \mathcal{C}$ are all nonrecursive;
2. undecidable when $\mathcal{G}, \mathcal{C}$ are SWS(CQ, UCQ) and $\mathcal{M}$ is MDT(UCQ), even when either $\mathcal{M}$ or $\mathcal{C}$ is nonrecursive;
3. in 2EXPSPACE when $\mathcal{G}, \mathcal{C}$ are SWS_{nr}(CQ, UCQ) and $\mathcal{M}$ is MDT_{nr}(UCQ), *i.e.,* when services and mediators are all nonrecursive;
4. decidable when $\mathcal{G}$ is SWS_{nr}(PL,PL), $\mathcal{C}$ is SWS(PL, PL) and $\mathcal{M}$ is MDT(PL); and
5. decidable when $\mathcal{G}$ is SWS(PL,PL), $\mathcal{C}$ is SWS_{nr}(PL, PL) and $\mathcal{M}$ is MDT_{nr}(PL). □

PROOF. (1) The undecidability of $\text{CP}(\text{SWS}_{\text{nr}}(\text{FO,FO}), \text{MDT}_{\text{nr}}(\text{FO}), \text{SWS}_{\text{nr}}(\text{FO,FO}))$ is verified by reduction from the satisfiability problem for FO queries.

(2) When either mediators or component services are nonrecursive, the undecidability can already be established by reduction from the equivalence problem for SWS(CQ,UCQ), which is proved undecidable by Theorem 4.1.

(3) We first show that $\text{CP}(\text{SWS}_{\text{nr}}(\text{CQ,UCQ}), \text{MDT}_{\text{nr}}(\text{UCQ}), \text{SWS}_{\text{nr}}(\text{CQ,UCQ}))$ can be reduced to the problem for equivalent query rewriting using views for UCQ with $\neq$, in EXPTIME. We then show that the rewriting problem is decidable in EXPSPACE for UCQ queries with $\neq$, by establishing a small model property, *i.e.,* if there exists an equivalent rewriting then there exists one bounded by EXPSPACE.

(4) We use the notion of k-prefix recognizable languages, *i.e.,* languages for which membership is determined by the first k symbols of the input sequence, for some $k \in \mathbb{N}$. We show that for each τ in SWS_{nr}(PL,PL) there is a k such that τ defines a k-prefix recognizable language. Moreover, given k, there is a bound on the size of mediators defining k-prefix recognizable languages. Thus we can guess a mediator of bounded size and check equivalence with the goal SWS.

(5) Similarly, we show that an MDT_{nr}(PL) mediator over component SWS's in SWS_{nr}(PL,PL) is only capable of defining k-prefix recognizable languages, for some $k \in \mathbb{N}$. Further, we show that if the language defined by the goal SWS τ is k-prefix recognizable, then we can put a bound on the value of k. If π and τ are equivalent, τ must be k-prefix recognizable. This again yields a bound on the size of possible mediators equivalent to τ, and thus decidability. □

For SWS(CQ, UCQ), one might be tempted to consider finding a nonrecursive mediator π over nonrecursive component SWS's for a *recursive* goal SWS τ such that $\pi \equiv \tau$. Indeed, there has been work on equivalence between recursive and nonrecursive datalog programs [10]. There is, however, a subtle difference between equivalence of SWS's and mediators, and equivalence on datalog queries. To see this, recall the runs of SWS's and mediators. Each time when an execution tree is expanded, *i.e.,* whenever new leaf nodes are spawned, an input message in the sequence $\mathcal{I}$ is consumed, by SWS's and (component services in) mediators alike. In other words, the computation steps of an SWS or a mediator is bounded by the length of $\mathcal{I}$. Therefore, while the computation of a nonrecursive π can be conducted in a fixed number of steps, the computation of a recursive τ may need an unbounded number of steps. Hence one can find a long enough sequence $\mathcal{I}$ of input messages such that *different* outputs are produced by π and τ on $\mathcal{I}$. In other words, π and τ are not equivalent on all input sequences. Indeed, in this setting the composition synthesis problem is reduced to composition synthesis of *nonrecursive* goal SWS's and nonrecursive mediators and components, *i.e.,* case (3) above. This is what the proof of (5) does for SWS(PL, PL): only k-prefix recognizable goal services make sense in that

context. In contrast, this is not an issue for the study of equivalence between datalog programs.

For the same reason, as indicated in the proof of (4), for a goal SWS τ in SWS_{nr}(CQ, UCQ), it suffices to consider only *nonrecursive* mediators and component SWS's, *i.e.,* case (3). This is in contrast to containment of nonrecursive datalog programs in recursive datalog programs (*e.g.,* [11]).

Special cases. Theorem 5.1 tells us that in general, composition synthesis for data-driven SWS's is beyond reach in the presence of recursion. Below we identify a decidable case for recursive data-driven services in SWS(CQ,UCQ), by leveraging a result on UC2RPQ queries established in [9].

A 2-way regular path query (2RPQ) Q expresses a regular expression on a semistructured database D. The database is an edge-labeled graph, encoded by a collection of binary relations for edges, along with their inverse. On D the query Q computes a set (d_0, d_q) of nodes such that $e_1(d_0, d_1), \ldots, e_q(d_{q-1}, d_q)$ is a path in D and $e_1, \ldots, e_q$ is a word in the regular language of Q. Unions of conjunctions of 2RPQ's can be naturally defined, denoted by UC2RPQ's. It is shown [9] that containment of UC2RPQ's is decidable in 2EXPTIME. One can express a UC2RPQ in SWS(CQ,UCQ). Let us denote by SWS(UC2RPQ) the class of all SWS's in SWS(CQ,UCQ) that express UC2RPQ's, and by MDT(UC2RPQ) the class of all mediators in MDT(UCQ) that are equivalent to some SWS in SWS(UC2RPQ). We also denote by $\text{SWS}_{\text{nr}}(\text{CQ}^r)$ the subclass of SWS_{nr}(CQ, UCQ) such that each SWS in the class expresses a CQ query (with two states). We show below that given a goal service τ in SWS(UC2RPQ) and a set S of components in $\text{SWS}_{\text{nr}}(\text{CQ}^r)$, it is decidable in 2EXPTIME to determine whether or not there exists a mediator π in MDT(UC2RPQ) such that $\pi \equiv \tau$.

Corollary 5.2: The synthesis problem CP(SWS(UC2RPQ), MDT(UC2RPQ), $\text{SWS}_{\text{nr}}(\text{CQ}^r)$) is decidable in 2EXPTIME. □

PROOF. One can verify that the composition synthesis problem is PTIME equivalent to the problem for equivalent query rewriting for UC2RPQ queries using CQ views. Further, using the maximally-contained rewriting algorithm developed for datalog [14] one can show that if an equivalent rewriting of a UC2RPQ query exists, then an equivalent UC2RPQ rewriting can be found in PTIME. From these and [9] the 2EXPTIME upper bound follows. □

While the composition synthesis problem is open when goal and component services are in SWS(PL, PL) and mediator is in MDT(PL), we identify several decidable cases. First, the analysis is simplified when only disjunction is allowed in the synthesis queries of mediators. We denote this subclass of MDT(PL) by MDT(∨). In this setting the problem becomes decidable in 3EXPSPACE when mediators, goal and component services are all recursive. Second, we can reduce the complexity if goal services are NFA's (2EXPSPACE-complete) or DFAs (in EXPSPACE), as found in, *e.g.,* the Roman model. Third, we restrict MDT(PL) such that each component services is invoked at most a fixed number of times in all transition rules combined, and the sizes of the synthesis functions are bounded. We refer to MDT(PL) with such a bound as MDT^b(PL). Then the composition synthesis problem is decidable in EXPSPACE, and it is PSPACE-complete if only nonrecursive component services are allowed.

Theorem 5.3:

1. CP(SWS(PL, PL), MDT(∨), SWS(PL, PL)) is decidable in 3EXPSPACE;
2. CP(NFA, MDT(∨), SWS(PL, PL)) is 2EXPSPACE-complete, while CP(DFA, MDT(∨), SWS(PL, PL)) is in EXPSPACE;
3. CP(SWS(PL, PL), MDT^b(PL), SWS(PL, PL)) is in EXPSPACE; CP(SWS(PL, PL), MDT^b(PL), SWS_{nr}(PL, PL)) is PSPACE-complete.

□

PROOF. (1) Every SWS(PL,PL) is translated in exponential time to an equivalent NFA. The proof then employs the 2EXPSPACE NFA rewriting algorithm of [8], taking into account the subtle interplay between a mediator and the SWS's it calls. That is, the corresponding NFA's should stop processing the input the first time a final state is encountered. The overall algorithm is therefore in 3EXPSPACE.

(2) When goal services can be expressed as an NFA, and mediators are in MDT(∨) we show that composition synthesis is PTIME-equivalent to the rewriting problem of [8] for NFA. From these and the complexity bounds of [8] follows the result for an NFA goal. We show that the bound can be lowered to EXPSPACE for a DFA goal.

(3) For MDT^b(PL), composition synthesis has a small model property: if a mediator in MDT^b(PL) exists, then there must be one of which the size is bounded by a polynomial in the size of the goal SWS and component SWS's. Further, testing equivalence of an SWS and such a mediator can be done in EXPSPACE in general, and in PSPACE when the component SWS's are nonrecursive. From this the upper bounds of (3) follow. The PSPACE lower bound is by a reduction from nonemptiness analysis of SWS(PL,PL) (Theorem 4.1 (3)). □

It should be remarked that SWS composition is quite different from service composition studied for the Roman model [6]. In contrast to the Roman model that allows interleaving of executions of component services, SWS composition requires that component services run to completion *in order to synthesize their actions.* The difference in the semantics leads to different complexity bounds on the synthesis: it is EXPTIME-complete for the Roman model [6, 24], whereas it is 2EXPSPACE-hard for SWS(PL,PL) composition.

6. Conclusion

We have proposed a notion of synthesized Web services to uniformly characterize FSA and transducer abstractions of Web services. Moreover, SWS's decouple unnecessary temporal dependencies imposed by FSA abstractions, and facilitate deterministic synthesis of service actions. We have established matching lower and upper bounds on the nonemptiness, validation and equivalence problems associated with various classes of SWS's. We have also provided complexity bounds on their composition synthesis.

The main results are summarized in Tables 1 and 2. Consistent with the findings of [2, 5, 6, 12, 13, 16, 24, 29], the decision and composition synthesis problems are either undecidable or highly intractable for recursively defined services. In light of these negative results we have identified decidable cases and special cases with lower complexity bounds. Moreover, our proof techniques, *e.g.,* finding service composition via query rewriting using views, may help develop practical heuristic algorithms in certain application domains.

There is naturally much more to be done. First, the composition synthesis problem is open when goal and component services are in SWS(PL, PL) and mediators are in MDT(PL). This is, however, nontrivial. In fact the proofs

$\mathcal{G}$: goal service	CP($\mathcal{G}, \mathcal{M}, \mathcal{C}$) $\mathcal{M}$: mediator	$\mathcal{C}$:component	**Complexity**
SWS(FO,FO)	MDT(FO)	SWS(FO,FO)	undecidable
SWS_{nr}(FO,FO)	MDT_{nr}(FO)	SWS_{nr}(FO,FO)	undecidable
SWS(CQ,UCQ)	MDT(UCQ)	SWS(CQ,UCQ)	undecidable
SWS(CQ,UCQ)	MDT(UCQ)	SWS_{nr}(CQ,UCQ)	undecidable
SWS(CQ,UCQ)	MDT_{nr}(UCQ)	SWS(CQ,UCQ)	undecidable
SWS_{nr}(CQ,UCQ)	MDT_{nr}(UCQ)	SWS_{nr}(CQ,UCQ)	2EXPSPACE
SWS_{nr}(PL,PL)	MDT(PL)	SWS(PL,PL)	decidable
SWS(PL,PL)	MDT_{nr}(PL)	SWS_{nr}(PL,PL)	decidable
Special cases			
SWS(UC2RPQ)	MDT(UC2RPQ)	$\text{SWS}_{\text{nr}}(\text{CQ}^r)$	2EXPTIME
SWS(PL,PL)	MDT($\vee$)	SWS(PL,PL)	3EXPSPACE
NFA	MDT($\vee$)	SWS(PL, PL)	2EXPSPACE
DFA	MDT($\vee$)	SWS(PL, PL)	EXPSPACE
SWS(PL, PL)	MDT^b(PL)	SWS(PL, PL)	EXPSPACE
SWS(PL, PL)	MDT^b(PL)	SWS_{nr}(PL, PL)	PSPACE

Table 2: Complexity of composition synthesis

for its special cases (Theorem 5.1 (4, 5)) are already non-elementary. Second, in several decidable composition cases, we have only provided upper bounds, for which the exact complexity remains unknown. Third, the undecidability and highly intractable results suggest that we identify practical restrictions that allow decidable and better still, tractable static analyses and composition synthesis. Fourth, a practical topic for future work is to extend SWS's by incorporating aggregation and a cost model into action synthesis to find, *e.g.,* a travel package with minimum total cost when airfare, hotel and other components are all taken together. While aggregation on composed services is certainly needed in practice, we are not aware of any formal study of this issue. Finally, we also plan to investigate for SWS's the verification problems and restrictions studied in [12, 13], as well as the impact of lossy channel [13] and global constraints [5, 15] on static analyses and composition synthesis of SWS's.

Acknowledgments. Wouter Gelade is Research Assistant of the Fund for Scientific Research - Flanders (Belgium). Wenfei Fan is supported in part by EPSRC GR/S63205/01, GR/T27433/01 and EP/E029213/1. Floris Geerts is supported in part by EPSRC EP/E029213/1.

7. References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] S. Abiteboul, V. Vianu, B. S. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000.

[3] F. N. Afrati, M. Gergatsoulis, and T. G. Kavalieros. Answering queries using materialized views with disjunctions. In *ICDT*, 1999.

[4] F. N. Afrati, C. Li, and P. Mitra. Rewriting queries using views in the presence of arithmetic comparisons. *Theor. Comput. Sci.*, 368(1-2):88–123, 2006.

[5] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB*, 2005.

[6] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic service composition based on behavioral descriptions. *Int. J. Cooperative Inf. Syst.*, 14(4):333–376, 2005.

[7] Business Process Execution Language for Web Services version 1.1 (BEPL4WS), 2004. *http://www.ibm.com/developerworks/library/specification/ws-bpel/*.

[8] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. *JCSS*, 64(3):443–465, 2002.

[9] D. Calvanese, G. D. Giacomo, and M. Y. Vardi. Decidable containment of recursive queries. *TCS*, 336(1):33–56, 2005.

[10] S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. *JCSS*, 54(1):61–78, 1997.

[11] S. S. Cosmadakis and P. C. Kanellakis. Parallel evaluation of recursive rule queries. In *PODS*, 1986.

[12] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven Web applications. *JCSS*, 73(3):442–474, 2007.

[13] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven Web services. In *PODS*, 2006.

[14] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *PODS*, 1997.

[15] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL Web services. In *WWW*, 2004.

[16] X. Fu, T. Bultan, and J. Su. Conversation protocols: a formalism for specification and verification of reactive electronic services. *TCS*, 328(1-2):19–37, 2004.

[17] M. Garey and D. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Company, 1979.

[18] C. E. Gerede, R. Hull, O. H. Ibarra, and J. Su. Automated composition of e-services: lookaheads. In *ICSOC*, 2004.

[19] G. Gottlob and C. Papadimitriou. On the complexity of single rule datalog queries. *Inf. Comput.*, 183(1):104–122, 2003.

[20] A. Y. Halevy. Theory of answering queries using views. *SIGMOD Record*, 29(4), 2001.

[21] R. Hull and J. Su. Tools for composite web services: a short overview. *SIGMOD Record*, 34(1):5–12, 2005.

[22] A. Klug. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160, 1988.

[23] A. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, 1995.

[24] A. Muscholl and I. Walukiewicz. A lower bound on Web services composition. In *FoSSaCS*, 2007.

[25] OWL-S: Semantic Markup for Web Services, 2004. *http://www.w3.org/Submission/OWL-S/*.

[26] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS web services. In *ISWC*, 2005.

[27] Semantic Web Services Framework (SWSF) Version 1.1, 2005. *http://www.daml.org/services/swsf/1.1/*.

[28] M. Spielmann. *Abstract State Machines: Verification Problems and Complexity*. PhD thesis, RWTH , 2000.

[29] M. Spielmann. Verification of relational transducers for electronic commerce. *JCSS*, 66(1):40–65, 2003.

[30] Web Services Conversation Language (WSCL) 1.0, 2002. *http://www.w3.org/TR/wscl10/*.

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

[32] S. Yu. Regular languages. In *Handbook of Formal Languages*, volume 1. Springer, 1996.