On The Expressive Power of Service Models for Automated Service Composition

Yin Wang, Hamid Motahari-Nezhad, Sharad Singhal

HP Laboratories
HPL-2010-33

Abstract:

Automatic service composition is an important problem in service oriented computing. Existing service models, whether embodied in specification standards or implicit in composition methods, offer tradeoffs between the level of detail at which component services are described and resulting opportunities for automated composition. For example, WSDL describes service interfaces whereas OWL-S describes preconditions and effects, and the two standards support different approaches to automated composition. The precise relationship between service models and the constraints they place on automated composition, however, remains unclear. It is therefore difficult to understand the full implications of commitment to a particular way of describing services. This paper presents a formal framework that relates service models to composition capabilities. Our framework covers the three most commonly used service models: input/output, precondition/effects, and automaton models. It uses logic formulae and formal languages to characterize precisely the expressiveness and composition capabilities inherent in each service model. Our framework facilitates choosing an appropriate service model for a specific composition task. Applied to common workflow patterns, it identifies minimally expressive service models that enable composition of these patterns. As a case study, we analyze 33 workflows used for IT transformation services within HP Enterprise Services. Our analysis identifies the appropriate service model for the automated construction of these workflows.

External Posting Date: March 6, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: March 6, 2010 [Fulltext]

On The Expressive Power of Service Models for Automated Service Composition

Yin Wang
HP Labs, Palo Alto, CA
yin.wang@hp.com

Hamid Motahari-Nezhad
HP Labs, Palo Alto, CA
hamid.motahari@hp.com

Sharad Singhal
HP Labs, Palo Alto, CA
sharad.singhal@hp.com

Abstract

Automatic service composition is an important problem in service oriented computing. Existing service models, whether embodied in specification standards or implicit in composition methods, offer tradeoffs between the level of detail at which component services are described and resulting opportunities for automated composition. For example, WSDL describes service interfaces whereas OWL-S describes preconditions and effects, and the two standards support different approaches to automated composition. The precise relationship between service models and the constraints they place on automated composition, however, remains unclear. It is therefore difficult to understand the full implications of commitment to a particular way of describing services.

This paper presents a formal framework that relates service models to composition capabilities. Our framework covers the three most commonly used service models: input/output, precondition/effects, and automaton models. It uses logic formulae and formal languages to characterize precisely the expressiveness and composition capabilities inherent in each service model. Our framework facilitates choosing an appropriate service model for a specific composition task. Applied to common workflow patterns, it identifies minimally expressive service models that enable composition of these patterns. As a case study, we analyze 33 workflows used for IT transformation services within HP Enterprise Services. Our analysis identifies the appropriate service model for the automated construction of these workflows.

1 Introduction

Technological advances in Service Oriented Architecture (SOA) have led to a significant increase in the number of services available on the Web. This provides an unprecedented opportunity for exploiting the composability principle [12] of services in SOA, and composition of services on the Web. Service composition techniques and languages have received significant attention in industry (demonstrated by languages such as WS-BPEL [2]) and academia (see surveys [24, 32, 10]).

The service composition problem takes as input a set of component services with a composition goal, and generates a composite service, usually represented by a *workflow*, that achieves the goal. Automated composition approaches are based on service models that characterize component services. These models have a profound impact on the capabilities of the composition algorithm. In general, a more expressive service model enables the composition of more complex composite services. However, expressive service models usually require more manual service description effort and result in computationally expensive composition algorithms. Currently, there are three common categories of service models for service composition (see Table 1):

- **Input Output (I/O)** A service is modeled as a pair of input and output sets. An input or an output is identified by its data schema.
- **Precondition Effect (P/E)** A service is modeled as a pair of precondition and effect sets. These sets include logic literals that represent the state *external* to the service.

- **Automaton (stateful)** A service's *internal* state and dynamics are modeled using finite automata.

Many variations of each model have been developed with different composition capabilities. For example, the WSDL standard describes input and output data types of a Web service. However the composition method based on these I/O models may assume that the output is consumed by a service instead of copied for repeated use [34], and the set of composite services one can build is different in this case. For P/E models, deterministic models and non-deterministic models with conditional effects result in different control flow structures in composites. The OWL-S standard employs P/E service models and allows composition based on AI planning methods; alternative P/E models have also been proposed [29, 23]. Existing automaton-based service models synchronize automata representing local services in different ways, e.g., using asynchronous messages [9], action delegation [6], or parallel product synchronization [30]. Each alternative carries different implications for service composition.

Manual service composition is the norm today, but is formidably difficult due to the sheer complexity of individual services, which provide human-readable documentation but not model-based descriptions. Manual composition is likely to become untenable as services rapidly proliferate and rapidly evolve. Increasingly automated composition is the most attractive alternative going forward, and automation requires standardized service models. Before service vendors commit to specific service description approaches, however, they must first understand how various approaches enable and constrain automated composition.

This paper explains the implications and limitations of existing service models in terms of their expressiveness and the opportunities for service composition that they afford. Comparing different composition methods directly by their composition semantics is difficult because they assume different formal frameworks for service models. The output of the composition approach (composite service) cannot be used as a basis for comparison either because of differences in workflow representations. However, regardless of the representation, the *execution traces* of a composite service are always sequences of component service invocations. All possible execution traces form an execution *language*, which provides a common ground to compare service composition approaches.

Our contribution is to introduce a formal framework based on the execution languages of the composite services that each class of service model supports. Our framework characterizes the three common service models precisely using logic formulae and language properties. It has at least two interesting applications. First, for each service model, our framework defines the capabilities and complexity of the corresponding composition algorithms and also defines the space of control flow structures in the composite services. Second, given a composite service workflow or target execution language, our framework identifies the simplest service model that is sufficient for the composition.

We demonstrate the value of our framework via two applications. We consider well-known workflow patterns [35] and find the minimally expressive service model to support the automated composition of each. We then apply our framework to a collection of real-world IT transformation workflows used by HP and identify which service models suffice to automatically compose each workflow from component services.

The rest of the paper is organized as follows. Section 2 provides background on service composition, service models, and temporal logic. Section 3 introduces our formal framework for service models and Section 4 uses it to compare service models along several dimensions. Section 5 extends the framework to variations of service models and Section 6 presents applications of the framework. Section 7 reviews related work, and Section 8 concludes with a discussion.

2 Background

Numerous variants of each service model exist in the literature. Here we present definitions of I/O, P/E, and automaton service models that formalize these categories. This section also covers the background of *Linear Temporal Logic*, which we use to characterize I/O and P/E service models.

Model	Features	Description	Standard
Input/Output	interface only, no semantics	input and output (schema, type etc.)	WSDL
Precondition/Effect	semantics only, service is stateless	preconditions/effects (situation calculus)	OWL-S (draft)
Automaton	stateful service model	states, actions and transition function	

Table 1: Summary of service models for composition

2.1 Service Composition

Given a set of component services and a goal, the service composition problem is to assemble components into a composite whose execution achieves the goal. For I/O and P/E models, we assume that every component is an *atomic service*. The atomicity assumption means that one service's execution is independent of that of others, and cannot be interrupted or affected by others. Under this assumption, an execution of a composite service can be serialized, e.g., by the start time of each atomic service that it invokes. Therefore, a composite service induces a collection of execution strings describing all possible execution sequences of its atomic services.

Stateful service models define states and transitions *within* component services, and the composition method defines synchronization among these stateful models. We therefore treat the *internal state transitions* within automaton models of component services as atomic services. Under this view, a stateful model essentially defines *execution dependencies* among its atomic services. The final composition built upon these stateful models must comply with these dependencies. This view connects stateful models with I/O and P/E models since execution sequences under different models are all based on atomic services. We define the language of a composite service in terms of these execution sequences.

Definition 1. *(Execution Language) Given a set A of atomic services, and a string s over A, i.e., $s \in A^*$, s is a valid execution sequence if the serial execution of s complies with the permitted uses of A. The set of all valid execution sequences of A is a language, denoted as $\mathscr{L}_A^{ex}$.*

This definition concerns the execution semantics of composites and does not depend on any service model or composition method. It is the responsibility of the service model to define execution semantics in such a way that composition methods yield composites whose execution sequences are consistent with $\mathscr{L}_A^{ex}$. Different service models define different execution languages, and this is the focus of our analysis. For example, a loan offer service may follow a credit report service, but not the other way around. If we were to compose a loan application service, the service model must communicate this ordering constraint to the composition method.

Definition 1 assumes no initial condition for the execution. This simplifies our comparison of different service models. When initial conditions are present, our analysis results still apply, but we need to encode these conditions in different service models in ways consistent with their definitions.

2.2 Input/Output Service Model

The input/output model for an atomic service defines a set of input data needed to execute the service, and the set of output data produced after the execution. The input and output may include identifiers such as name and data type.

Definition 2. *(I/O Model) Given set A of atomic services and set D of data types, an input/output model for an atomic service $a \in A$ is a pair (I_a, O_a), where $I_a, O_a \subseteq D$, and $I_a \cap O_a = \emptyset$.*

If s is a string of length n, i.e., $|s| = n$, the position an element of s is indexed by $0, 1, ..., n-1$. The i-th element of s is denoted $s[i]$. Execution semantics under the I/O model require that the inputs of each atomic service be provided by the outputs of some previously invoked atomic service:

Definition 3. *(Language of I/O Model) A string $s \in A^*$ is a valid execution sequence under the I/O model if $\forall i < |s|, I_{s[i]} \subseteq \bigcup_{0 \le j < i} O_{s[j]}$. The set of all valid execution sequences under the I/O model is a language, denoted as $\mathscr{L}_A^{IO}$.*

This definition is independent of any composition algorithm under the I/O model. It defines an execution language over A by the semantics of the I/O model. Any I/O model based composition algorithm must output execution sequences within the language.

Example 1. *Consider three services $A = \{a, b, c\}$, representing user input, credit report, and loan services, respectively. We use I/O models $a = (\emptyset, \{name, SSN\})$, $b = (\{SSN\}, \{name, score\})$, and $c = (\{name, score\}, \{loan\})$, respectively. Then strings a, ab, and abc all belong to $\mathscr{L}_A^{IO}$, but b or ac does not.*

Under our I/O model, once a service is enabled to execute, it is enabled forever, and can be executed repeatedly. In other words, the set of atomic services that may execute is non-decreasing as composite service execution proceeds. If this monotonicity property does not hold—e.g., if services *consume* their inputs as by-products of execution [34]—we require a different model, such as the P/E model.

2.3 Precondition/Effect Service Model

The P/E service model defines the preconditions and effects of a service using logic formulae whose literals describe system state. An atomic service's preconditions must be satisfied before the service can execute; immediately after its execution, system state is consistent with the service's effects. We formalize the P/E service model along the lines of the *classical representation* of AI planning system [28]. Our formalization is consistent with planning frameworks based on *situation calculus*, which is widely used in service composition [27].

The classical representation expresses system state as a conjunction of literals; unmentioned literals are implicitly false (the "closed world" assumption). First order literals such as *At(goods, warehouse)* are allowed but quantifiers $\forall$ or $\exists$ are not permitted. First order literals must be *ground* and *function-free*, i.e., *At(x, y)* or *At(f(Bob), home)* are not allowed. As a result, any P/E schema for a set of services can be propositionalized, i.e., turned into a finite collection of purely propositional formulae with no variables [28]. In other words, preconditions and effects are expressed in a way that is equivalent to propositional logic, but is more succinct and convenient in practice.

The precondition of a P/E service is a conjunction of (positive) literals. The effects of a service are literals that may be both positive and negative. Positive literals are added to the state after execution, while negative literals are removed from the state. We first discuss deterministic service models here, and extend to models with conditional effects in Section 5.

Definition 4. *(P/E Model) Given a set of literals L, the P/E model of an atomic service $a \in A$ is a triple (P_a, E_a^+, E_a^-), where $P_a \subseteq L$ is the precondition, $E_a^+ \subseteq L$ is the positive effect, and $E_a^- \subseteq L$ is the negative effect, $P_a \cap E_a^+ = \emptyset$ and $E_a^+ \cap E_a^- = \emptyset$.*

If all literals in P_a are true in the current state T, i.e., $P_a \subseteq T$, service a may execute. After execution, literals in E_a^+ are added to T and literals in E_a^- are removed from T, i.e., the resulting state $T' = T \cup E_a^+ - E_a^-$. We separate positive and negative effects into two sets for notational convenience and to facilitate comparisons between I/O and P/E models. Formally, we define the execution semantics of the P/E model as follows.

Definition 5. *(Language of P/E Model) A string $s \in A^*$ is a valid execution sequence under the P/E model if state T_i before executing $s[i]$ satisfies $P_a \subseteq T_i$. State T_i is defined recursively as follows $T_0 = \emptyset$, $T_i = T_{i-1} \cup E^+_{s[i-1]} - E^-_{s[i-1]}, 0 < i < |s|$. The set of all valid execution sequences under the P/E model is a language denoted $\mathscr{L}_A^{PE}$.*

Example 2. *An online storage system has three services $A = \{a, b, c\}$, representing copy, backup, and hosting services, respectively. We define P/E models $a = (\emptyset, \{copy\}, \emptyset)$, $b = (\{copy\}, \{backup\}, \{copy\})$, and $c = (\{copy, backup\}, \{available\}, \emptyset)$. That is, a copies the data, b marks a copy as a backup, and when both a copy and a backup are ready, c hosts the data online. Strings a, ab, and abac all belong to $\mathscr{L}_A^{PE}$, but abc does not, because b negates precondition copy needed by c.*

P/E models discussed in prior literature sometimes include input and output. For example, the OWL-S standard allows "IOPE" specifications that include input, output, preconditions, and effects. However, preconditions and effects are sufficient to express the same constraints on the execution of composite services that inputs and outputs imply: a precondition can express that an atomic service requires as input data generated as an effect of some other atomic service's execution.

Modeling services with "lifecycle" properties, e.g., the Google Checkout service [1], is sometimes more succinct, natural, and convenient if we can explicitly represent stages in such lifecycles. Automaton models are frequently used for such services [21].

2.4 Automaton Service Models

As stated in Section 2.1, automaton models explicitly represent the internal states and transitions of component services. Transitions are treated as atomic services; they are local actions that change the state of a component service and are often triggered by external events.

Definition 6. *(Automaton Service Model) Given a set A of atomic services, the automaton service model defines a set G of finite automata. An automaton $g \in G$ is a triple (Q_g, A_g, δ_g), where Q_g is the (finite) set of states, $A_g \subseteq A$ is the set of actions, and $\delta_g : Q_g \times A_g \to Q_g$ is the partial transition function.*

For generality, we omit the initial and final states in the above definition by assuming that every state in G_S is both initial and final. Service composition methods using automaton models define the composition semantics that glue automata together, usually based on common transitions. Pistore et al. [30] advocate the *parallel product* of local automata, which synchronizes all local automata that share a common transition; Berardi et al. [6] propose a central orchestrator that can *delegate* an action to one local automaton; Bultan et al. [9] suggest that messages communicated among peers should trigger local transitions asynchronously. Under these approaches, composition goals are usually specified by regular languages, and composition algorithms are consistent with the composition semantics. We give the definition of parallel product here as representative approach.

Definition 7. *(Parallel Product) Given automata $g, h \in G$, their parallel product automaton is $g||h = (Q_g \times Q_h, A_g \cup A_h, \delta_{g||h})$ where $\delta_{g||h}$ is defined as*

$$(q_g, q_h) \times a \to \begin{cases} (\delta_g(q_g, a), q_h) & \delta_g(q_g, a) \text{ defined} \\ (q_g, \delta_h(q_h, a)) & \delta_h(q_h, a) \text{ defined} \\ (\delta_g(q_g, a), \delta_h(q_h, a)) & \text{both defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The above definition extends to more than two automata in a natural way. The execution language of the automaton service model with parallel product is the language defined by the parallel product automaton, which remains regular. When component automata do not share transitions in common, the execution language of the parallel product is the arbitrary interleaving of all regular languages defined by component automata, which is also regular. Another extreme case is when G consists of only one automaton. Then the execution language is equivalent to the regular language defined by the local automaton. These observations also apply to most other composition semantics proposed in the literature. Therefore, instead of describing each composition semantics precisely, we consider a regular language as the execution language for automaton service models.

For brevity and simplicity, in this paper we have restricted attention to finite automata, but our linguistic framework extends straightforwardly to composition methods based on other stateful models of services, e.g., Petri nets [5], workflows [4], and process algebras. Some of these formalisms are more general than finite automata, in the sense that their corresponding formal languages are a superset of regular languages. However our results do not require that the formal language corresponding to a service model be regular.

2.5 Linear Temporal Logic

A linear temporal logic (LTL) formula over some alphabet Σ consists of elements of Σ (propositional literals), boolean connectives $\neg$ (negation), $\vee$ (disjunction), and $\wedge$ (conjunction) and the temporal operators X (next), G (globally), and U (until). All connectives and operators are unary except for $\vee$, $\wedge$ and U.

The semantics of LTL is defined over a string s of characters in Σ. We still use $s[i]$ to denote the i-th character in s. If s is a string of length n and $0 \le i < j < n$, then $s[i, j]$ denotes the string $s[i]s[i+1]...s[j]$. Further, $s[i, *]$ denotes the suffix $s[i, n-1]$. We denote $s \models \varphi$ if the temporal formula φ holds in the string s, which is defined inductively as follows.

— for every symbol $a \in \Sigma$, $s \models a$ if $s[0] = a$

— $s \models \mathsf{X}\varphi$ if $|s| > 1$ and $s[1, *] \models \varphi$

— $s \models \mathsf{G}\varphi$ if $\exists i$ with $0 \le i < |s|$ s.t. $s[i, *] \models \varphi$

— $s \models \varphi \mathsf{U} \psi$ if $\exists i$ with $0 \le i < |s|$ s.t. $s[i, *] \models \psi$ and for any j with $0 \le j < i, s[j, *] \models \varphi$

Intuitively $\varphi \mathsf{U} \psi$ means φ is satisfied at every step until the step where ψ is satisfied. From that point on, there is no further restriction on either φ or ψ. In addition, we introduce the *weak until* binary operator W that simplifies our formulae. Its semantics is similar to that of the until operator but the stop condition is not required to occur, i.e.:

$$\varphi \mathsf{W} \psi = (\varphi \mathsf{U} \psi) \vee \mathsf{G}\varphi.$$

For example, string abc satisfies formulae $\mathsf{X}b$, $\mathsf{G}(a \vee b \vee c)$, and $a\mathsf{U}b$. With the semantics of LTL, we can define a language corresponding to an LTL formula as follows.

Definition 8. *(LTL Language) Given an alphabet Σ and an LTL φ, language $\mathscr{L}_\varphi$ is the set of strings that satisfy φ, i.e., $\mathscr{L}_\varphi = \{s | s \models \varphi\}$. A language $\mathscr{L}$ is definable in LTL iff there exists an LTL formula φ s.t. $\mathscr{L} = \mathscr{L}_\varphi$.*

There is a clear distinction between the temporal logic defined above and the propositional logic used in P/E models to describe states, or the description logic used in semantic web for ontology. Temporal logic captures temporal relationships in a sequence, while the others describe static system states. We use temporal logic to characterize execution languages of different service models.

Languages definable in LTL are in fact a strict subset of the regular language, as the following theorem states. This result is relevant to our comparison of service models.

Theorem 1. *[11] A language is definable by an LTL formula iff it is definable by a star-free regular expression.*

A *star-free regular expression* is built up from symbols in the alphabet Σ, and operators $\cdot, \cup, \cap, \neg$ denoting "concatenation", "union", "intersection", and "complementation (with respect to Σ^*)", respectively. Kleene (or star) closure * is not allowed. Star-free regular expressions are a strict subset of regular expressions. However, the difference between the star-free regular language and the regular language is rather subtle. A star-free regular language is not necessarily finite since Σ^* is included as it is the complement of the empty language.

Let the alphabet be the set of atomic services, $\Sigma = A$. A language over this alphabet represents a set of execution sequences. Since execution languages $\mathscr{L}_A^{IO}$ and $\mathscr{L}_A^{PE}$ are defined over the same alphabet, we represent these languages using LTL formulae as presented in the following.

3 Logic Representation of Service Models

The execution language of the automaton based service model is equivalent to regular language in the extreme case. Therefore, we assume regular language for automaton based models, and focus on the logic representation of I/O and P/E service models in this section.

3.1 Logic Formulae for I/O Models

According to the I/O service model in Definition 2 and its execution semantics in Definition 3, an atomic service $a \in A$ cannot take place until all of its input are available. Input p of a, i.e., $p \in I_a$, is available if another service b outputs p, i.e., $p \in O_b$. We translate this semantics into the following LTL formula using the weak until operator.

$$\bigwedge_{p \in I_a} \Big(\neg a \,\mathsf{W} \bigvee_{p \in O_b} b \Big) \tag{1}$$

In a word, Formula (1) states that service a cannot take place "until" service b happens first, which generates input p for a. An execution string over A satisfies (1) iff a set of services that generates all input of a have occurred before a (or a never occurs). For each input p of a, there could be a set of services that all output p. Therefore we have the disjunctive clause after the W operator. To summarize, $\bigvee_{p \in O_a} a$ "achieves" input p for a, and the Conjunctive Normal Form (CNF) $\bigwedge_{p \in I_a} \bigvee_{p \in O_b} b$ "enables" a.

For any valid execution sequence under the I/O model, Formula (1) must be true for every atomic service in A. We have the following formula for the I/O service model.

$$\varphi = \bigwedge_{a \in A} (1) \tag{2}$$

For example, service a in Example 1 has no formula since I_a is empty, b gives formula $\neg b \mathsf{W} a$, and c gives $(\neg c \mathsf{W}(a \vee b)) \wedge (\neg c \mathsf{W} b)$, which is simplified as $\neg c \mathsf{W} b$. Together, service model in Example 1 is defined by $\neg b \mathsf{W} a \wedge \neg c \mathsf{W} b$, which means a must occur before b and b must occur before c.

Similarly to Definition 3, these logic formulae are defined over execution sequences based on the semantics, and independent of composition algorithms. We establish the relationship between these formulae and execution languages under I/O models by the following two theorems.

Theorem 2. *Given set A of atomic services, I/O model (I_a, O_a) for each service a in A, and formula φ in (2), we have $\mathscr{L}_A^{IO} = \mathscr{L}_\varphi$.*

Proof. If a string s belongs to the language $\mathscr{L}_A^{IO}$ in Definition 3, then for every character $s[i]$ in the string, the input of $s[i]$ must have been produced by atomic services preceding $s[i]$, i.e., $s[i]$ cannot take place until atomic services that output $s[i]$'s input has occurred first. Therefore, string s must satisfy φ in (2), i.e., $s \in \mathscr{L}_\varphi$.

If a string s satisfies φ, Formula (1) says that an atomic service either does not exist in s, or a set of atomic services have occurred before it in the string that produces its input. Therefore, s is a valid execution sequence, i.e., $s \in \mathscr{L}_A^{IO}$. □

The above theorem states that if we have an I/O model for a set of services, there is an LTL formula equivalent to the execution language allowed by the I/O model. On the other hand, given an execution language $\mathscr{L}_A^{ex}$ over A we want to comply with, there are certain conditions $\mathscr{L}_A^{ex}$ must satisfy in order to use the I/O service model.

Theorem 3. *Given language $\mathscr{L}_A^{ex}$ over A, if there exists subsets $A_a^1, A_a^2 \ldots A_a^{X_a}$ for each atomic service $a \in A$, where $A_a^i \subset A, a \notin A_a^i$, and X_a is the index, such that $\mathscr{L}_A^{ex} = \mathscr{L}_\psi$, where*

$$\psi = \bigwedge_{a \in A} \bigwedge_{0 < i \le X_a} \Big(\neg a \,\mathsf{W} \bigvee_{b \in A_a^i} b\Big) \tag{3}$$

then there exists an I/O model for A, such that $\mathscr{L}_A^{IO} = \mathscr{L}_A^{ex}$.

Proof. We construct an I/O model for A as follows. For each atomic service $a \in A$, its input set I_a contains X_a elements $I_a = \{p_a^1, p_a^2, \ldots p_a^{X_a}\}$, corresponding to $A_a^1, A_a^2, \ldots A_a^{X_a}$, respectively. For each element in subset $A_a^i \subset A$, we add p_a^i to its output set, i.e., $\forall b \in A_a^i$, we have $p_a^i \in O_b$. With this construction, ψ becomes

$$\psi = \bigwedge_{a \in A} \bigwedge_{p_a^i \in I_a} \Big(\neg a \,\mathsf{W} \bigvee_{p_a^i \in O_b} b\Big)$$

which is equivalent to φ in (2). Therefore, we have $\mathscr{L}_A^{ex} = \mathscr{L}_\psi = \mathscr{L}_\varphi = \mathscr{L}_A^{IO}$. The last equality is the result of Theorem 2. □

Formula (3) defines the condition $\mathscr{L}_A^{ex}$ must satisfy in order to use the I/O model for composition. If the formula is satisfied, one can define an I/O model that describes atomic services in A, whose execution language is exactly $\mathscr{L}_A^{ex}$. Otherwise, no I/O model generates exactly $\mathscr{L}_A^{ex}$. In this case, we can use a more restrictive I/O model whose execution language is a subset of $\mathscr{L}_A^{ex}$, which implies that there are certain valid execution sequences permitted by A but not composable by the model. On the other hand, a less restrictive I/O model whose execution language is a superset of $\mathscr{L}_A^{ex}$ may output invalid execution sequences. Formula (2) and Formula (3) share the similar form. However, the former is associated with an I/O model, which defines the execution language of the model, while the latter is a stand-alone formula, which characterizes certain properties of a language over A.

3.2 Logic Formulae for P/E Models

Similarly to I/O models, we represent execution sequences allowed by P/E models using LTL formulae. Note that the literals used in these LTL formulae represent atomic services in A, and should not be confused with the literals in preconditions and effects of P/E models that describe system states.

Based on Definitions 2 and 4, literals in the P/E model are analogous to input and output data types in the I/O model. Under this analogy, preconditions in the P/E model are effectively input in the I/O model, and positive effects are output of a service. Therefore, we have the following formula similar to (1).

$$\bigwedge_{l \in P_a} \Big(\neg a \,\mathsf{W} \bigvee_{l \in E_b^+} b\Big) \tag{4}$$

The key difference between these two service models is the negative effect in the P/E model. An negative effect essentially negates a literal that is already available. While under the I/O model, an output cannot be revoked. To capture negative effects, we have

$$\bigwedge_{l \in P_a} \Big(\bigvee_{l \in E_c^-} c \rightarrow (\neg a \,\mathsf{W} \bigvee_{l \in E_b^+} b)\Big) \tag{5}$$

where "$\rightarrow$" is the logic implication operator, $a \rightarrow b = \neg a \vee b$.

The above formula states that if service c takes place and c negates l that is a precondition of a, a cannot take place from now on until some service b "regenerates" l as its positive effect. Formula (4) is the condition needed to execute a. It must be satisfied at the beginning of the execution sequence. While Formula (5) describes the consequence of negative effects, i.e., when a negative effect occurs, the clause in Formula (4) must be satisfied again (at that step), since the precondition needed by a does not exist any more. This formula must be satisfied at every step of the execution sequence. Therefore, we have the following LTL formula that represents the P/E service model.

$$\varphi = \bigwedge_{a_i \in A} (4) \wedge \mathsf{G}(5) \qquad (6)$$

For example, service c in Example 2 gives the formula $\neg c \mathsf{W} a \wedge \neg c \mathsf{W} b$. In addition, the negative effect *data* caused by b gives formula $\mathsf{G}(b \rightarrow (\neg c \mathsf{W} a))$.

These formulae are consistent with the *partial-order planning* method [28]. An action (service) b with positive effect l, i.e., $l \in E_b^+$ "achieves" l for a, called a *causal link*, denoted as $b \xrightarrow{l} a$, corresponding to Formula (4). If l is a negative effect of c, i.e., $l \in E_c^-$, c *conflicts* with this causal link, and must not occur between b and a, corresponding to Formula (5). The following theorem establishes the relationship between the above formulae and the execution language of the P/E model.

Theorem 4. *Given set A of atomic services, P/E model* (P_a, E_a^+, E_a^-) *for each service a in A, and formula* φ *in (6), we have* $\mathscr{L}_A^{PE} = \mathscr{L}_\varphi$.

The proof to this theorem is analogous to that of Theorem 2 and therefore omitted here. Similarly to Theorem 3, if a given execution language $\mathscr{L}_A^{ex}$ can be represented by an LTL formula in the form of (6), we can derive a P/E service model whose execution language is exactly $\mathscr{L}_A^{ex}$. The discussion follows the same strategy of Theorem 3, but a lot lengthier due to the consequences of negative effects. We omit it from the paper.

4 Comparisons of Service Models

We do not argue that any one service model is strictly preferable to the others. Rather, the three models offer different tradeoffs among important considerations including the convenience of describing component services, the computational complexity of composition algorithms, and the range of possible composite workflows that can be supported.

4.1 Complexity of Description

As Table 1 summarizes, describing services using I/O models is relatively easy because the widely accepted WSDL standard already includes the input and output data types of each atomic service. The P/E model requires preconditions and effects to be defined for each atomic service, which OWL-S supports. Describing services using automata requires detailed understanding of this model, and an agreement on composition semantics.

While the amount of description may increase from the I/O model to the automaton model, annotating services using automaton models is sometimes more convenient. For example, if we require that a loan offer service must follow a credit report service, it may be more intuitive to specify this constraint using two consecutive transitions in an automaton. By contrast, we would need to carefully match input and output schema of each service in an I/O model [26] or specify literals that describe states before and after each service's execution in a P/E model, perhaps with the aid of an ontology [29].

4.2 Execution Languages

Based on our analysis, the I/O service model is equivalent to the P/E service model without negative effects. Therefore, the P/E model is more expressive than the I/O model. Since execution languages allowed by P/E models can be characterized using LTL formulae, based on Theorem 1, the P/E model is less expressive than the automaton model. The following corollary and Figure 1 summarize these relationships.

Corollary 5. *The set of execution languages that I/O service models generate is a strict subset of those generated by P/E service models; while the latter is a strict subset of*

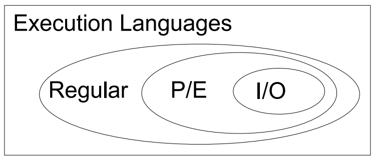

Figure 1: Relationships among execution languages by different service models.

regular languages, i.e., languages allowed by automaton service models.

4.3 Composition Capability

We have discussed the execution language of each service model. Given an execution language, depending on its equivalent logic representation, we can choose an appropriate service model for the composition. Finding equivalent logic representation of a language is difficult, if one is not familiar with formal methods. Here we give intuitive explanations of the composition capability of each model, and a few simple model selection criterions based on control flow structures.

Under the I/O service model, as discussed in Section 3.1, a service is "enabled" by a CNF logic formula, i.e., a set of conditions must all be satisfied before executing the service, and each condition can be satisfied by executing one atomic service from many choices. There is no negation of these literals in the CNF. From the composition point of view, each disjunctive clause in the CNF can be implemented by XOR fork/join structure, while the conjunction can be implemented by AND fork/join. These two structures are sufficient to build composite services under the I/O model.

The P/E model defines a similar CNF formula that enables a service, but the same service can be "disabled" again by services with conflicting negative effects. A control flow structure directly related to this fact is the *milestone* pattern [35]. This pattern states that a service can only be executed during a certain phase. For example, a customer may cancel his/her order as long as it is not shipped. This pattern cannot be composed under I/O service models, but is allowed under P/E service models. Similarly, repeated execution of an atomic service cannot be composed under I/O service models, but is supported by P/E (see Example 2), because the output data set remains unchanged after repeated executions.

The difference between the regular language and the star-free regular language is rather subtle, and in practice it usually does not affect the selection of service model. However, P/E service models correspond to only a subset of the star-free regular language because of the special form expressed by Formula (6). Not every LTL formula can be translated into this form, and therefore even some star-free regular language cannot utilize the P/E service model. Here we give an example that is star-free regular but not allowed by P/E service models.

Example 3. *There are three atomic services $A = \{a, b, c\}$. Let the execution language be $\mathscr{L}_A^{ex} = \{a, b, ab, ba, ac, bc\}$, but $abc \notin \mathscr{L}_A^{ex}$. There is no P/E service model that captures exactly this language.*

This example is based on the above discussion that a service under the P/E model (or I/O model) is enabled by an CNF formula without negation. The XOR logic operator cannot be represented in this form. Therefore, we cannot have the pattern where either a or b must occur before c, but not both. However, the automaton model supports it since $\mathscr{L}_A^{ex}$ is a regular language.

4.4 Complexity of Composition Algorithms

With a specific composition goal g, all possible execution sequences that achieve the goal, denoted as $\mathscr{L}_g$, are a subset of the execution language allowed by the service model. The size of $\mathscr{L}_g$ is often exponential in the length of g. In practice, a service composition algorithm typically outputs one composite service instead of enumerating all strings in $\mathscr{L}_g$, whose execution sequences are a subset of $\mathscr{L}_g$. For example, a planning algorithm typically outputs the most efficient plan to achieve the goal, instead of enumerating all solution plans. Therefore, we compare the computational complexity of composition algorithms that find one solution for the composition goal.

Under the I/O service model, the set of output data grows monotonically as more atomic services are executed. Therefore, we have a transitive closure style composition algorithm. It executes an enabled service during each iteration, and adds its output set to the available data set. The algorithm terminates when the output data set subsumes the goal or no more service can be executed, i.e., the goal is not achievable. The time complexity of the algorithm is $O(m^2n^2)$ where m is the number of distinct input and output data types, and n is the number of atomic services.

Deciding the existence of a plan for a planning model is known to be PSPACE-complete [28]. Therefore we do not have any polynomial composition algorithm for the P/E service model.

The computational complexity of automaton model based composition algorithms varies depending on the composition semantics. However, these algorithms almost all involve the construction of the global state, which is the Cartesian product of state sets of all component automata, e.g., as defined by the parallel product in Definition 7. Therefore their complexity is at least exponential in the number of component automata used for the composition.

5 Extensions

Our linguistic framework that covers the three common service models is not restricted by the specific forms we define them. Here we consider two popular variants of these models.

5.1 Conditional P/E Models

The P/E service model in Definition 4 assumes fixed sets of preconditions and effects. Most planning models proposed in the literature, as well as the OWL-S, support conditional, or nondeterministic, models. A conditional model includes a set of different outcomes associated with a service. Each outcome has its own sets of positive and negative effects. For example, a credit card payment service may result in a successful charge or a failure.

A conditional model may or may not specify different preconditions that trigger different conditional effects. We consider the general conditional P/E model where one can define different preconditions for different effects. Under this consideration, an atomic service with conditional effects can be represented by a set of unconditional atomic services, i.e., according to Definition 4. After this transformation, the execution semantics is exactly the same as Definition 5 implies. However, when considering the execution language of this model, we must map these derived unconditional services back to the symbol that represents the original conditional service. For example, if an atomic service a has two conditional outcomes, we define unconditional services a' and a'' that represent the two outcomes, respectively. Services a' and a'' follow the execution semantics under the unconditional P/E model. But for each valid execution sequence that includes a' or a'', we replace every occurrence of a' or a'' by a. This mapping we apply to the execution sequence is called the *projection* of a language:

Definition 9. *(Language Projection) Let g be a function* $g : \Sigma \to \Sigma'$, *we extend g to a string* $s \in \Sigma^*$ *naturally as* $g(s) = g(s[0])g(s[1])\ldots$. *The projection of a language* $\mathscr{L}$ *over* Σ *is* $\mathscr{L}^g = \{g(s) | s \in \mathscr{L}\}$, $\mathscr{L}^g \subset \Sigma'^*$.

Therefore, the execution language of a conditional P/E service model is equivalent to the execution language of the transformed unconditional P/E model under some projection function. We have shown that the execution language of an P/E model is equivalent to a certain star-free regular expression. A star-free regular expression remains star-free regular after the projection, since the projection does not introduce Kleene * operator or other non-regular operator into the expression. As a result, we have the following corollary.

Corollary 6. *The execution language of a P/E service model with conditional effects is definable by a star-free regular expression. Therefore, languages that can be captured by P/E models with conditional effects remain a strict subset of regular languages.*

The conditional P/E service model is more expressive than the unconditional P/E service model due to the existence of conditional effects. Conditional effects are closely related to the *deferred choice* pattern [35], which is a deferred XOR where the choice is made by the environment at runtime. A service with conditional outcomes lead to different successor services depending on which outcome takes place. Therefore, in a composite service, a conditional atomic service is typically followed by multiple branches, and the runtime outcome enables exactly one of them to be executed next. This pattern is not allowed by the unconditional P/E model or the I/O model. In those models, an execution sequence always results in the same state, with the same set of services enabled.

In addition, loops (arbitrary cycle pattern [35]) are permitted by conditional effects but not with I/O or unconditional P/E models. For example, an online shopping service may put the "add to cart" action in a loop until the customer completes his/her shopping list. Under I/O or unconditional P/E models, there is no need to use loops since the state after each cycle is exactly the same. The execution language in Example 3 can be captured by our conditional P/E model as well. We omit the detailed discussion here due to the space limit.

5.2 Data-Centric Service Models

Recently data-centric, or artifact-centric, business processes have received increasing attention [8, 14]. The data-centric design centers around data objects and their life cycles. A data-centric business process manipulates these objects to reach a goal, which is often expressed as a set of objects reaching certain states.

The data-centric design can be applied to our service composition framework. Under this view, an atomic service operates on one or a set of data objects. This service model defines how an atomic service modifies states of data objects. Analogous to the comparison between process-oriented programming and object-oriented programming, data-centric service model offers better modularity, reusability and possibly easier description. But in terms of the expressive power, associating data objects with a service model does not increase its expressiveness.

For example, we can define a data-centric I/O service model where each input/output data type is attached to a data object (with no further annotation). This helps to avoid input/output name conflict and the model becomes more readable. But the expressiveness remains the same.

The P/E service model has been extended to incorporate the data-centric design in a similar manner [14]. In this case, state literals are associated with data objects, e.g., a $status$ literal becomes structured as $object.status$. The structured literal establishes relationships between data objects and services that operate on them. Similarly to the data-centric I/O model, the data-centric P/E model merely renames literals, and the expressiveness remains the same.

Perhaps the most benefits of the object-oriented design come with the data-centric automaton service model. This idea occurred in [7] under a different composition framework. It is not introduced as a data-centric approach. We present the data-centric automaton model as follows. Recall from Section 2.4 that an automaton service model consists of set A of atomic services and set G of automata. Under the data-centric view, each automaton in G naturally defines the life cycle of some data object, i.e., we associate with each automaton a data object. The state of an automaton reflects the state of the object associated with it. A transition (atomic service) of the automaton accepts the object in a certain state and moves it to a different state. This explanation comes closest to the human perception of how services interact with data objects. However, since we are not changing the definition of the automaton service model, the expressive power is again the same.

6 Applications of The Framework

As an application of our framework, we can identify the appropriate service model that enables the automated construction of a given workflow based on its control flow patterns. This section discusses this application using well-known workflow patterns and a case study of HP IT transformation services.

6.1 Workflow Patterns

Control flow patterns are an important criterion to evaluate the expressiveness of work languages [35]. Similarly we use these patterns to evaluate the composition capability of different service models. Section 4.3 already discussed some of the basic patterns. Here we thoroughly evaluate all relevant patterns mentioned in [35], and give the minimally expressive service model that supports the composition of each pattern. The result is summarized in Table 2.

The sequential pattern is supported by every service model. The parallel pattern AND fork/join and choice pattern XOR fork/join are supported by I/O model (as well as other service models) as discussed in Section 4.3. Here we assume that the composition algorithm always uses an AND fork/join whenever multiple services can be executed in parallel, and uses an XOR fork/join whenever multiple alternative services are available. A less optimized composition algorithm may output a serial work-

Group	Pattern	Service Model
Basic Control Flow	Sequence	I/O
	AND fork/join	I/O
	XOR	I/O
Advanced Branching Sync	OR fork/join	I/O
	Multi-merge	I/O
	Discriminator	I/O
Structural Patterns	Cycles	Conditional P/E
	Implicit Termination	I/O
State-Based Patterns	Deferred Choice	Conditional P/E
	Interleaved Parallel	Automaton
	Milestone	P/E

Table 2: Workflow patterns and the minimally expressive service models that support them

flow as the solution, which does not imply that the service model supports only sequential patterns. With the automaton model, we assume that the execution of one component automaton is serial while the execution of different component automata can be parallelized using the AND pattern. The XOR pattern can be used when there are choices among different automata, e.g., the delegation approach [6].

The OR fork pattern represents multiple choices, i.e., a subset of available branches is executed. This pattern can be completed by OR join, *multi-merge*, or *discriminator* pattern, which represent different ways to merge of multiple branches. These advanced branching patterns can be represented by combinations of basic AND and XOR patterns. Therefore, the I/O service model is sufficient.

The cycle pattern is enabled by conditional P/E model but not the I/O or the unconditional P/E model as discussed in Section 5.1. The implicit termination pattern is essentially a terminal node. One can implement this with any service model.

State based patterns are closely related to our classification of service models. The deferred choice pattern is supported by the conditional P/E model as Section 5.1 discusses. The interleaved parallel pattern is when we need to execute a set of atomic services sequentially, but with arbitrary order. The I/O or P/E service model does not support this because they cannot differentiate between the AND pattern and this pattern. If multiple atomic services are enabled, it is natural to use the AND fork/join to maximize the parallelism rather the using the interleaved parallel pattern. The automaton model allows this pattern because the local automaton could explicitly include all possible execution ordering. The milestone pattern is a typical example that the P/E model supports but not the I/O model, as discussed in 4.3. Other workflow patterns include the group of multiple instance patterns and the group of cancelation patterns. These patterns are irrelevant to our discussion and therefore not included in the table.

Table 2 provides a quick guideline on which service model to use based on the control flow patterns we need to compose. For example, if we want to compose a BPEL workflow [2], we can analyze the control flow structures of BPEL. Basic control flow structures include *sequence, flow, switch*, and *while loop*, corresponding to the sequence, AND, XOR, and cycle patterns, respectively. The conditional P/E model supports the composition of all these structures. However, the semantics of *link* in BPEL is rather unusual and complicated. An activity cannot start until all source activities of its incoming links are completed and the `joinCondition` evaluates to true. Since the `joinCondition` permits arbitrary boolean formulae, depending on its format, we may need an automaton service model for the composition.

6.2 HP IT Transformation Services Case Study

We used our framework to analyze a set of 33 workflow templates used for IT transformation services within HP Enterprise Services. The templates represent activities undertaken by project teams as part of outsourcing and consulting engagements, and include between 12 to 30 atomic activities each. For example a server virtualization project contains activities such as “define KPIs”, “design management and host infrastructure”, “procure infrastructure”, and “test VMs”. Because it is important to make service engagements repeatable and efficient across customers, specialized teams handle different activities, and need to schedule their time and resources across multiple projects and customers. For each new engagement, the templates are customized and the required workflow is composed manually from the templates. Because customization usually introduces changes in the templates, the individual workflows become error-prone, and difficult to construct and maintain.

Motivated by the goal of automating the composition of workflows for new engagements from these templates, we wanted to identify the service models that could be used to represent the activities described in the templates. An examination of these templates identified a total of 577 activities. As many of them are shared between the templates, we found 270 activities that could be represented as atomic services. We further found that the majority of workflows only required sequential and AND patterns. More specifically, we found 7 workflows involving deferred choice patterns. For example, a storage backup service contains the condition "need additional hardware?", if yes, "procure", else "proceed". From Table 2 we observe that these workflows require a conditional P/E service model, while the remaining workflows can be composed using I/O service models only.

We built a prototype based on I/O service models using the algorithm mentioned in Section 4.4. Our algorithm generated all workflows that do not involve the deferred choice patterns. By specifying the proper inputs and outputs in the I/O model as the composition goal, the prototype was able to generate all desired workflows. We believe that by properly annotating the templates, we can provide tools that can construct the custom workflows necessary in a more consistent and error-free manner. As an example, we found that in many cases the template workflows provide alternate paths for "customer approval?" while others simply assume that the result is deterministic and always results in true. Similarly, while the "close project" activity follows "handover to support", sometimes there is an additional step "customer signoff" in between. We believe that the missing steps are due to omissions in the templates rather than the semantics of the workflows. Automatically composed workflows can handle such conditional effects in a comprehensive and consistent way. The value of our framework comes from being able to handle the different service models in a unified manner, as opposed to requiring different ways of analyzing workflows requiring different service models.

7 Related Work

Existing work in the area of service composition can be studied into two broad categories: those presenting frameworks for manual composition (surveyed in [10, 24]) and languages such as WS-BPEL [2], and those that provide automated techniques (described in surveys [32, 24]). The theoretical framework presented in this paper is related to the later category. Most existing service composition methods in this category are based on one of the three service models discussed in this paper. Other than those discussed in this paper, composition methods based on I/O service models include [33, 18, 16, 20], P/E models include [3, 22, 31], and automaton models include [15, 25, 17]. This shows that the presented framework is generic and covers a wide range of existing composition work.

Some approaches mix the I/O and P/E service models [19]. In this case, the input and output are used to compose sequential and parallel structures while precondition and effect are used to generate condition branches, i.e., deferred choice patterns. Another composition approach is based on the dependency graph model [13]. These dependency graphs can be viewed as simplified automaton models, and therefore the approach belongs to the automaton model category. The increasingly popular RESTful framework promotes resources as its central principle, which is close to the data-centric service model. The P/E service model has been applied to the RESTful framework recently [36].

8 Conclusions

This paper presents a formal language framework for analyzing service composition. It covers the three most common service models and characterizes their expressiveness, composition capability, and the computational complexity of composition. We have also presented a practical application of our framework in the context of service composition for HP IT Transformation services.

We have shown that our framework clarifies the implications of service models for service composition and identifies the least expressive service model that supports given workflow language capabilities. This in turn may facilitate automated service composition by aiding the selection of the most appropriate service model for desired composition capabilities.

References

[1] Google checkout service. `http://code.google.com/apis/checkout/developer/index.html`.

[2] WS-BPEL 2.0, OASIS standard. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.

[3] V. Agarwal, K. Dasgupta, N. M. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava. A service creation environment based on end to end composition of Web services. In *WWW*, pages 128–137, 2005.

[4] P. Albert, L. Henocque, and M. Kleiner. A constrained object model for configuration based workflow composition. In *BPM Workshops*, pages 102–115, 2005.

[5] P. Álvarez, J. A. Bañares, and J. Ezpeleta. Approaching Web service coordination and composition by means of Petri nets: The case of the nets-within-nets paradigm. In *ICSOC*, pages 185–197, 2005.

[6] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *ICSOC*, pages 43–58, 2003.

[7] P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, and M. Wagner. Control flow requirements for automated service composition. In *ICWS*, pages 17–24, 2009.

[8] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *BPM*, pages 288–304, 2007.

[9] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW*, pages 403–410, 2003.

[10] S. Dustdar and W. Schreiner. A survey on Web services composition. *Int. J. Web Grid Serv.*, 1(1):1–30, 2005.

[11] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. 1990.

[12] T. Erl. *SOA Principles of Service Design*. Prentice Hall PTR, 2007.

[13] R. Eshuis, P. W. P. J. Grefen, and S. Till. Structured service composition. In *BPM*, pages 97–112, 2006.

[14] C. Fritz, R. Hull, and J. Su. Automatic construction of simple artifact-based business processes. In *ICDT*, pages 225–238, 2009.

[15] G. D. Giacomo, M. de Leoni, M. Mecella, and F. Patrizi. Automatic workflows composition of mobile services. In *ICWS*, pages 823–830, 2007.

[16] Z. Gu, J. Li, and B. Xu. Automatic service composition based on enhanced service dependency graph. In *ICWS*, pages 246–253, 2008.

[17] R. R. Hassen, L. Nourine, and F. Toumani. Protocol-based Web service composition. In *ICSOC*, pages 38–53, 2008.

[18] R. Hewett, P. Kijsanayothin, and B. Nguyen. Scalable optimized composition of Web services with complexity analysis. In *ICWS*, pages 389–396, 2009.

[19] S. Kona, A. Bansal, M. B. Blake, and G. Gupta. Generalized semantics-based service composition. In *ICWS*, pages 219–227, 2008.

[20] Z. Liu, A. Ranganathan, and A. Riabov. Modeling Web services using semantic graph transformations to aid automatic composition. In *ICWS*, pages 78–85, 2007.

[21] A. Marconi, M. Pistore, P. Poccianti, and P. Traverso. Automated Web service composition at work: the amazon/mps case study. In *ICWS*, pages 767–774, 2007.

[22] H. Meyer and M. Weske. Automated service composition using heuristic search. In *BPM*, pages 81–96, 2006.

[23] P. Mika, D. Oberle, A. Gangemi, and M. Sabou. Foundations for service ontologies: aligning OWL-S to Dolce. In *WWW*, pages 563–572, 2004.

[24] N. Milanovic and M. Malek. Current solutions for Web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[25] S. Mitra, R. Kumar, and S. Basu. Automated choreographer synthesis for Web services composition using I/O automata. In *ICWS*, pages 364–371, 2007.

[26] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *WWW*, pages 993–1002, 2007.

[27] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW*, pages 77–88, 2002.

[28] D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[29] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma. Meteor-S Web service annotation framework. In *WWW*, pages 553–562, 2004.

[30] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS Web services. In *ICWS*, pages 293–301, 2005.

[31] A. Ragone, T. D. Noia, E. D. Sciascio, F. M. Donini, and S. Colucci. Fully automated Web services orchestration in a resource retrieval scenario. In *ICWS*, pages 427–434, 2005.

[32] J. Rao and X. Su. A survey of automated Web service composition methods. In *Proc. 1st Int'l Workshop on Semantic Web Services and Web Process Composition*, 2004.

[33] A. Riabov, E. Bouillet, M. Feblowitz, Z. Liu, and A. Ranganathan. Wishful search: interactive composition of data mashups. In *WWW*, pages 775–784, 2008.

[34] Z. Shen and J. Su. On completeness of Web service compositions. In *ICWS*, pages 800–807, 2007.

[35] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[36] H. Zhao and P. Doshi. Towards automated RESTful Web service composition. In *ICWS*, pages 189–196, 2009.