A Language-Based Framework For Analyzing Service Representation Models and Service Composition Approaches

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

Hamid Reza 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 computing. Existing works on service composition assume different representation models with various expressiveness for component services as well as for the composition logic. This has made understanding these composition approaches and their applications difficult.

This paper presents a novel language-based formal framework that acts as a basis for understanding the expressiveness of service models and composition logics. It covers the three common service representation models, i.e., input/output, precondition/effects, and automaton models. The novelty of this framework is in using the execution language of service composition, characterized by logic formulae and formal languages, as a common ground to analyze service models and composition logics. We have studied the composition capability of each service model in terms of enabling common workflow patterns, and analyzed the computational complexity of composition algorithms of different service models. The framework simplifies the job of understanding service representation models and composition logics for a given service composition task.

Keywords—service composition, language, workflow

I. 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 [10] 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) and academia (see surveys [20], [28], [8]).

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 service description effort and result in computationally expensive composition algorithms. A component service typically consists of a set of operations. Currently there are three categories of service models that describe these operations: (i) Input/Output (I/O): an operation is modeled as a pair of input and output sets, which are identified by the data schema; (ii) Precondition/Effect (P/E): an operation is modeled as a pair of precondition and effect sets, which are logic literals representing typically the state of the component service; and (iii) Automaton (stateful): a component service uses finite automata to describe its state, and the precedence relationships or dependencies among its operations.

Despite the existence of an extensive amount of work on automated composition methods, service composition is performed often manually today. WSDL is a widely accepted standard for describing service interfaces (the input/output model), however, there are much more information relevant to how to compose services that is found in human-readable documentation but not in model-based descriptions. Besides the need for advancing enabling technology, we believe a main obstacle for service providers is understanding various service composition approaches from the perspective of the expressiveness of the service representation models as well as their composition capabilities. The difficulty arises from the fact that there are various representation models for component services as well as the composition logics, for which no existing work presents a common ground to analyze and understand.

In this paper, we aim to fill this gap by presenting a language-based framework that enables us to explain the capabilities and limitations of service models in terms of their expressiveness and the opportunities for service com-
position. Analyzing different composition methods directly by service model or the composition logic is difficult because they use different modeling frameworks and different composition logic (workflow). We propose to use the execution traces of a composite service, which are sequences of operation invocations in component service (e.g., an operation in WSDL), as the basis for the analysis. All possible execution traces form an execution language, which provides a common ground to analyze service composition approaches (see Fig. 1).

The novel contributions of this paper include: (i) we introduce a formal framework based on the execution languages of the composite services supported by each class of the service model. Our framework characterizes the three common service models using logic formulae and language properties; and (ii) we analyze common workflow patterns [30] and present how they are supported in the composition by various service models.

The framework benefits both service vendors and service users. Given target workflows or execution languages, service vendors may use our framework to identify a suitable service model that is sufficient for the composition. Given a service model in our framework, service users understand exactly the capabilities and complexity of the corresponding composition algorithms and also the space of control flow structures in the composite services.

The rest of the paper is organized as follows. Section II provides background on service composition, service models, and temporal logic. Section III introduces our formal framework for service models and Section IV uses it to compare service models along several dimensions. Section V extends the framework to variants of service models and Section VI uses the framework to analyze workflow patterns. Section VII reviews related work, and Section VIII concludes.

II. BACKGROUND AND DEFINITIONS

Numerous variants of each service model exist. Here we present definitions of I/O, P/E, and automaton 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 models.

A. Service Composition

Given a set of operations in 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 operation is atomic. The atomicity assumption means that one operation’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 operation that it invokes. Therefore, a composite service induces a collection of execution strings describing all possible execution sequences of its operations.

Stateful service models define states and transitions within component services, and the composition method defines synchronization among these stateful models. Typically transitions of these automata are operations in the component service. Since these transitions are atomic too, a stateful model essentially defines execution dependencies among its operations (transitions). 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 operations. We define the language of a composite service in terms of these execution sequences.

Definition 1: (Execution Language) Given a set \( \Sigma \) of operations, and a string \( s \in \Sigma^* \) (Kleene star), \( s \) is a valid execution sequence if the serial execution of \( s \) complies with the permitted uses of \( \Sigma \). The set of all valid execution sequences of \( \Sigma \) is a language, denoted as \( L^{eq}_\Sigma \).

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 the execution sequences of composites are consistent with \( L^{eq}_\Sigma \). Different service models define different execution languages, and this is the focus of our analysis. For example, a loan offer operation may follow a credit report operation, 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.

B. Input/Output Service Model

The input/output model for an operation defines a set of input data needed to execute the operation, 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 \( \Sigma \) of operations and set \( D \) of data types, an input/output model for an operation \( \alpha \in \Sigma \) is a pair \( (I_\alpha, O_\alpha) \), where \( I_\alpha, O_\alpha \subseteq D \), and \( I_\alpha \cap O_\alpha = \emptyset \).

If \( s \) is a string of length \( n \), i.e., \( |s| = n \), the position of an element in \( s \) 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 operation be provided by the outputs of previously invoked operations:

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

Similar to Definition 1, the above definition is independent of composition algorithms, which facilitates comparison. An I/O model based composition algorithm must output execution sequences within its execution language.
Example 1: Consider three operations $\Sigma = \{\alpha, \beta, \gamma\}$, representing user input, credit report, and loan services, respectively. We use I/O models defining recursively as follows, but $\beta$ or $\alpha\gamma$ does not.

C. Precondition/Effect Service Model

The P/E service model defines the preconditions and effects of operations using logic formulae whose literals describe service state. An operation’s preconditions must be satisfied before the service can execute; immediately after its execution, service state is consistent with the operation’s effects. We formalize the P/E service model along the lines of the classical representation of AI planning system [24].

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 $\text{At}(\text{goods, warehouse})$ are allowed but quantifiers $\forall$ or $\exists$ are not permitted. First order literals must be ground and function-free, i.e., $\text{At}(x, y)$ or $\text{At}(\text{Bob}, \text{home})$ are not allowed. As a result, any P/E schema for a set of operations can be propositionalized, i.e., turned into a finite collection of purely propositional formulae with no variables [24]. For simplicity, we consider only propositional literals in our P/E model.

The precondition of a P/E operation is a conjunction of (positive) literals. The effects 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 V.

Definition 4: (P/E Model) Given a set of literals $L$, the P/E model of an operation $\alpha \in \Sigma$ is a triple $(P_\alpha, E_\alpha^+, E_\alpha^-)$, where $P_\alpha \subseteq \Sigma$ is the precondition, $E_\alpha^+ \subseteq L$ is the positive effect, and $E_\alpha^- \subseteq L$ is the negative effect. $P_\alpha \cap E_\alpha^+ = \emptyset$ and $E_\alpha^+ \cap E_\alpha^- = \emptyset$.

If all literals in $P_\alpha$ are true in the current state $T$, i.e., $P_\alpha \subseteq T$, service $\alpha$ may execute. After execution, literals in $E_\alpha^+$ are added to $T$ and literals in $E_\alpha^-$ are removed from $T$, i.e., the resulting state $T' = T \cup E_\alpha^+ - E_\alpha^-$. 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 $s \in \Sigma^*$ is a valid execution sequence under the P/E model if state $T_i$ before executing $s[i]$ satisfies $P_{s[i]} \subseteq T_i$. State $T_i$ is defined recursively as follows $T_0 = \emptyset$, $T_i = T_{i-1} \cup E_{s[i-1]}^+$.

Example 2: An online storage system has three operations $\Sigma = \{\alpha, \beta, \gamma\}$, representing copy, backup, and hosting services, respectively. We define P/E models $\alpha = (\emptyset, \{\text{copy}\}, \emptyset)$, $\beta = (\{\text{copy}\}, \{\text{backup}\}, \{\text{copy}\})$, and $\gamma = (\{\text{copy, backup}\}, \{\text{available}\}, \emptyset)$. That is, $\alpha$ copies the data, $\beta$ marks a copy as a backup, and when both a copy and a backup are ready, $\gamma$ hosts the data online. Strings $\alpha, \alpha\beta, \alpha\beta\alpha\gamma$ all belong to $L_{\Sigma}^{PE}$, but $\alpha\gamma$ does not, because $\beta$ negates precondition $\text{copy}$ needed by $\gamma$.

D. Automaton Service Models

Based on the discussion in Section II-A, we define an automaton service model as a set of automata whose transitions are operations in component services.

Definition 6: (Automaton Service Model) Given a set $\Sigma$ of operations, the automaton service model defines a set $G$ of finite automata. An automaton $g \in G$ is a triple $(Q_g, \Sigma_g, \delta_g)$, where $Q_g$ is the (finite) set of states, $\Sigma_g \subseteq \Sigma$ is the set of transitions, and $\delta_g : Q_g \times \Sigma_g \rightarrow Q_g$ is the partial transition function.

The initial and final states of the automata are typically defined by the composition task and goal. Service composition methods using automaton models define the composition semantics that glue automata together, usually based on common transitions. Pistore et al. [26] advocate the parallel product of local automata, which synchronizes all local automata that share a common transition; Berardi et al. [4] propose a central orchestrator that can delegate an operation to one local automaton; Bultan et al. [7] 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 $g \parallel h = (Q_g \times Q_h, \Sigma_g \cup \Sigma_h, \delta_g \parallel h)$ where $\delta_g \parallel h | h$ is defined as

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

The above definition extends to more than two automata in a natural way. The execution language of the automaton service model is defined by the parallel product automaton, which remains regular. Other composition semantics proposed in the literature often produce regular execution languages as well. Therefore, we consider 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 [3], workflows [2], 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.

E. Linear Temporal Logic

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

The semantics of LTL is defined over a string \( s \) of characters in \( \Sigma \). We still use \( s[i] \) to denote the \( i \)-th character in \( s \). If \( s \) is a string of length \( n \) and \( 0 \leq i < j < n \), then \( s[i,j] \) denotes the string \( s[i]s[i+1]...s[j] \). Further, \( s[i,\ast] \) denotes the suffix \( s[i,n-1] \). We denote \( s = \phi \) if the temporal formula \( \phi \) holds in the string \( s \), which is defined inductively as follows.

- for every symbol \( a \in \Sigma \), \( s = a \) if \( s[0] = a \)
- \( s = X\phi \) if \( |s| > 1 \) and \( s[1,\ast] = \phi \)
- \( s = G\phi \) if \( \exists i \) with \( 0 \leq i < |s| \) s.t. \( s[i,\ast] = \phi \)
- \( s = \phi U \psi \) if \( \exists i \) with \( 0 \leq i < |s| \) s.t. \( s[i,\ast] = \psi \) and for any \( j \) with \( 0 \leq j < i \), \( s[j,\ast] = \phi \)

Intuitively \( \phi \psi \) means \( \phi \) is satisfied at every step until the step where \( \psi \) is satisfied. From that point on, there is no further restriction on either \( \phi \) or \( \psi \). 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.:

\[
\phi W \psi = (\phi U \psi) \lor G \phi.
\]

For example, string \( abc \) satisfies formulae \( Xb \), \( G(a \lor b \lor c) \), and \( aUb \). With the semantics of LTL, we can define a language corresponding to an LTL formula as follows.

**Definition 8:** (LTL Language) Given an alphabet \( \Sigma \) and an LTL \( \phi \), a language \( L_\phi \) is the set of strings that satisfy \( \phi \), i.e., \( L_\phi = \{s | s = \phi \} \). A language \( \mathcal{L} \) is definable in LTL iff there exists an LTL formula \( \phi \) s.t. \( \mathcal{L} = L_\phi \).

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, whereas 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.

**Theorem 1:** [9] A language is definable by an LTL formula iff it is a star-free regular expression.

A star-free regular expression is built up from symbols in the alphabet \( \Sigma \), and operators \( \cdot, \cup, \cap, \neg \) denoting “concatenation”, “union”, “intersection”, and “complementation (with respect to \( \Sigma^* \))”, respectively. Kleene (or star) closure “*” is not allowed. Star-free regular expressions are a strict subset of regular expressions.

Since LTL formulae define languages, we can compare execution languages with languages characterized by LTL formulae. Next, we represent execution languages of different service models in LTL formulae.

III. Logic For Service Models

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

A. Logic Representation for I/O Models

According to the I/O service model in Definition 2 and its execution semantics in Definition 3, an operation \( \alpha \in \Sigma \) cannot take place until all of its input are available. Input \( p \) of \( \alpha \), i.e., \( p \in I_\alpha \), is available if another operation \( \beta \) outputs \( p \), i.e., \( p \in O_\beta \). We translate this semantics into the following LTL constraint using the weak until operator.

\[
\bigwedge_{p \in I_\alpha} \left( \neg \alpha W \bigvee_{p \in O_\beta} \beta \right)
\]

(1)

In words, Formula (1) states that operation \( \alpha \) cannot take place “until” operation \( \beta \) happens first, whose execution generates input \( p \) for \( \alpha \). An execution string over \( A \) satisfies (1) iff a set of operations that generates all input of \( \alpha \) has occurred before \( \alpha \) (or \( \alpha \) never occurs). For each input \( p \) of \( \alpha \), there could be a set of operations that all output \( p \). Therefore we have the disjunctive clause after the \( W \) operator. To summarize, \( \bigvee_{p \in O_\alpha} \alpha \) “achieves” input \( p \) for \( \alpha \), and the Conjunctive Normal Form (CNF) \( \bigwedge_{p \in I_\alpha} \bigvee_{p \in O_\beta} \beta \) “enables” \( \alpha \).

For any valid execution sequence under the I/O model, Formula (1) must be true for every operation in \( \Sigma \). Therefore we have the following equation for the I/O model.

\[
\varphi = \bigwedge_{\alpha \in \Sigma} (1)
\]

(2)

We omit initial input for simplicity. For example, operation \( \alpha \) in Example 1 has no constraint since \( I_\alpha \) is empty; \( \beta \) gives \( \gamma \beta W \alpha \) and \( \gamma \) gives \( (\neg \gamma \wedge (\alpha \lor \beta)) \wedge (\neg \gamma W \beta) \), which is simplified as \( \neg \gamma W \beta \). Together, the service model is defined by \( \neg \gamma W \alpha \wedge \neg \gamma W \beta \), which means \( \alpha \) must occur before \( \beta \) and \( \beta \) in turn occurs before \( \gamma \). We establish the relationship between these constraints and execution languages under I/O models by the following theorem.

**Theorem 2:** Given a set \( \Sigma \) of operations, an I/O model for each operation in \( \Sigma \), and \( \varphi \) in (2), we have \( \mathcal{L}_\Sigma^{IO} = \mathcal{L}_\varphi \).
The above theorem states that if we have an I/O model, there is an LTL formula equivalent to its execution language. On the other hand, given an execution language $L^\Sigma_{\Sigma'}$ over $\Sigma$ we want to comply with, if it is equivalent to an LTL formula in the form of (2), we can use I/O models to describe operations in $\Sigma$ and achieve exactly $L^\Sigma_{\Sigma'}$.

### B. Logic Representation for P/E Models

Similar 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 operations in $\Sigma$, 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 an operation. Therefore, we have the following formula similar to (1).

$$\bigwedge_{l \in P_\alpha} \left( \bigvee_{\beta \in E^+_\alpha} \neg \alpha \text{ W} \bigvee_{l \in E^-_\beta} \beta \right) \tag{3}$$

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

$$\bigwedge_{l \in P_\alpha} \left( \bigvee_{\beta \in E^+_\alpha} \gamma \rightarrow \left( \bigvee_{l \in E^-_\beta} \gamma \rightarrow \left( \bigwedge_{l \in E^+_\alpha} \neg \alpha \text{ W} \bigvee_{l \in E^-_\beta} \beta \right) \right) \tag{4}$$

where “$\rightarrow$” is the logic implication operator.

The above formula states that if operation $\gamma$ takes place and $\gamma$ negates $l$ that is a precondition of $\alpha$, $\alpha$ cannot take place from now on until some operation $\beta$ “regenerates” $l$ as its positive effect. Formula (3) is the condition needed to execute $\alpha$. It must be satisfied at the beginning of the execution sequence. While Formula (4) describes the consequence of negative effects, i.e., when a negative effect occurs, the clause in Formula (3) must be satisfied again (at that step), since the precondition needed by $\alpha$ 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.

$$\varphi = \bigwedge_{\alpha \in \Sigma} (3) \land G(4) \tag{5}$$

For example, operation $\gamma$ in Example 2 gives the formula $\gamma l \lnot \alpha \land \lnot \gamma W \beta$. In addition, the negative effect data caused by $\beta$ gives formula $G(\beta \rightarrow (\gamma l \lnot \alpha))$.

These formulae are consistent with the partial-order planning method [24]. An operation $\beta$ with positive effect $l$, i.e., $l \in E^+_\beta$ “achieves” $l$ for $\alpha$, called a causal link, denoted as $\beta \triangleright l \alpha$, corresponding to Formula (3). If $l$ is a negative effect of $\gamma$, i.e., $l \in E^-_\gamma$, $\gamma$ conflicts with this causal link, and must not occur between $\beta$ and $\alpha$, corresponding to Formula (4). The following theorem establishes the relationship between the above formulae and the execution language of the P/E model.

**Theorem 3:** Given set $\Sigma$ of operations, P/E model $(P_\alpha, E^+_\alpha, E^-_\alpha)$ for each operation $\alpha \in \Sigma$, and formula $\varphi$ in (5), we have $L^\Sigma_{\Sigma'} = L^\Sigma_{\Sigma'}$.

The execution language of a P/E model is equivalent to $\varphi$ in (5). On the other hand, if a given execution language $L^\Sigma_{\Sigma'}$ can be represented by an LTL formula in the form of (5), we can derive a P/E model to compose exactly $L^\Sigma_{\Sigma'}$.

### IV. ANALYSIS 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 expressiveness of the service model, and the computational complexity of composition algorithms; see Table I.

#### A. Complexity of Description

Describing services using I/O models is relatively easy because the widely accepted WSDL standard already includes input and output data types of each operation. The P/E model requires annotating preconditions and effects of each operation, 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 operation must follow a credit report operation, 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 operation in an I/O model or specify literals that describe states before and after each operation’s execution in a P/E model, perhaps with the aid of an ontology [25].

#### B. Execution Languages

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, Theorem 1 implies that the P/E model is less expressive than the automaton model.

#### C. Complexity of Composition Algorithms

Under the I/O service model, the set of output data grows monotonically as more operations are executed. Therefore, we have a transitive closure style composition algorithm.
It executes an enabled operation 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 operation 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 operations.

Deciding the existence of a plan for a planning model is known to be PSPACE-complete [24]. 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.

V. 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: conditional P/E models and data-centric models. We keep the discussion at a high level due to space limit.

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 an operation. Each outcome has its own sets of positive and negative effects. For example, a credit card payment operation may result in a successful charge or a failure. Naturally, conditional P/E models are more expressive than unconditional P/E models. However, languages captured by conditional P/E models remain a strict subset of regular languages. We outline the proof as follows.

An operation with conditional effects can be represented by a set of unconditional operations. Each corresponds to one conditional effect of the original conditional operation. 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 operations back to the symbol that represents the original conditional operation. This is called language projection. We have shown that the execution language of a P/E model is equivalent to a star-free regular expression. A star-free regular expression remains star-free regular after any projection, since the projection does not introduce Kleene $*$ operator or other non-regular operator. As a result, the execution language of a conditional P/E service model is definable by a star-free regular expression—a strict subset of regular languages.

Recently data-centric, or artifact-centric, business processes have received increasing attention [6], [12]. 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. All three service models, I/O, P/E, and automaton, have data-centric variants. For example, a data-centric P/E model associates state literals with data objects [12], e.g., literal shipped becomes structured as order.shipped. The structured literals connect data objects with services that operate on them. A data-centric automaton model may define each automaton as the life cycle of some data object [5]. A transition in an automaton moves its associated object from one state to another.

Analogous to the comparison between process-oriented programming and object-oriented programming, data-centric service models offer better modularity, reusability and possibly easier description. But in terms of expressiveness or composition complexity, they are equivalent with the process based models.

VI. WORKFLOW PATTERNS

We have discussed the execution language of each service model. However, given an execution language itself, finding an appropriate service model for composition is not easy. Here we consider a different model selection criterion that is based on control flow patterns of the goal composite service. Control flow pattern is an important criterion to evaluate the expressiveness of workflow languages [30]. Similarly we use these patterns to evaluate the composition capability of different service models, and give the minimally expressive service model that supports the composition of each pattern.

Under the I/O service model, as discussed in Section III-A, an operation is “enabled” by a CNF logic formula. 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 introduces negative effects that can “disable” an operation. A control flow structure directly related is the milestone pattern [30]. This pattern states that an
operation 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 operation cannot be composed under I/O service models, but is supported by P/E (see Example 2).

Conditional effects are closely related to the deferred choice pattern [30], which is a deferred XOR where the choice is made by the environment at runtime. I/O and P/E models both have deterministic result and therefore do not support this pattern. Conditional P/E model allows the composition of this pattern by matching different outcomes with different choices. In addition, arbitrary cycle pattern [30] are permitted by conditional effects but not with I/O or unconditional P/E models. For example, an online shopping operation may put the “add to cart” action in a loop until the customer completes his/her shopping list. We cannot compose loops with operations that have deterministic outcomes.

Next we examine all other control flow patterns in the order they appear in [30]. The sequential pattern is supported by every service model. The parallel pattern AND and choice pattern XOR are supported by I/O model based on the above discussion. Here we assume that the composition algorithm always uses an AND whenever multiple operations can be executed in parallel, and uses an XOR whenever multiple alternative operations are available. A less optimized composition algorithm may output a serial workflow as the solution, which does not imply that the service model supports only sequential patterns. With the automaton model, we assume that branches inside one automaton is equivalent to XOR, while the execution of different component automata can be parallelized using AND.

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 implicit termination pattern is essentially a terminal node, which is irrelevant of the service model.

The interleaved parallel pattern indicates the sequential execution of a set of operations in arbitrary order. The subtle distinction between this pattern and the parallel pattern cannot be captured by I/O or P/E models. The automaton model allows this pattern by explicitly enumerating all possible execution ordering. Other workflow patterns include the group of multiple instance patterns and the group of cancelation patterns. These patterns are irrelevant to service models.

Table II 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, 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.

VII. RELATED WORK

Existing work in service composition can be studied into two broad categories: those presenting frameworks for manual composition (surveyed in [8], [20]) and languages such as WS-BPEL, and those providing automated techniques (surveyed in [28], [20]). 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 models include [29], [16], [14], [18], P/E models include [1], [19], [27], [22], and automaton models include [13], [21], [15]. Some approaches mix the I/O and P/E service models [17]. A graph dependency-based approach for service composition is presented in [11], which can be analyzed under automaton model. The P/E model has been applied to the RESTful framework recently [32], which is similar to the data-centric variant of P/E model. Finally, we note that process algebra characterizes concurrent processes more precisely than execution traces [31], while the latter is sufficient for our comparison.

VIII. CONCLUSIONS

This paper presents a formal language-based framework for analyzing service models and composition logics. It covers the three common service models and characterizes their expressiveness, composition capability, and the
computational complexity of composition. The framework proposes to use the execution language of composition logic as the common ground for analyzing and comparing service composition approaches. We have used the framework to analyze service models for service composition. We have also analyzed the service models and their composition capabilities in terms of common workflow patterns.

This framework and the presented results facilitates understanding and adoption of automated service composition approaches by aiding service providers in offering a model-based description of services and the selection of the appropriate service model for desired composition capabilities as well as the service consumers in understanding the capabilities and complexities of service composition approaches.

REFERENCES


