Finding the Optimal Representation for Service Composition Using the Theory of Regions

Yin Wang, Ahmed Nazeem, Ram Swaminathan

HP Laboratories
HPL-2010-191

Keyword(s):

Workflow, Service Composition, Automaton, Petri net, Theory of Regions

Abstract:

Service composition has received significant attention in the research community, but the focus has been on service semantics and composition algorithms, and the problem of representation of the composition outcome has been largely ignored. Ad hoc workflows are often employed, which typically sacrifice alternative paths and parallelism for the sake of simple representation. In this paper, we show how theory of regions, which was originally developed to derive Petri nets from finite state automata, can be applied to find the optimal representation of composition. To apply the theory, we first propose an automaton-based composition framework that incorporates most existing composition techniques without changing the service semantics or its description language. Then based on the special requirements of the composition representation, we develop our own Petri net synthesis algorithm that combines the benefits of two well known algorithms from the theory of regions. More specifically, our algorithm converts an automaton to an unlabeled Petri net, whenever such a conversion exists, and reduces the number of places in the net. We demonstrate that workflow-based representation can limit the concurrency even for simple input/output based service composition, while our Petri net-based representation is optimal in terms of flexibility and parallelism. Our experimental evaluations include a case study on composing Google Checkout Service, and the study on Oracle BPEL samples, for which our algorithm obtains better concurrent representations for almost all non-trivial cases.

External Posting Date: February 4, 2011 [Fulltext] Approved for External Publication

Internal Posting Date: November 21, 2010 [Fulltext]

Finding the Optimal Representation for Service Composition Using the Theory of Regions

Yin Wang
Hewlett-Packard Labs
1501 Page Mill Rd
Palo Alto, CA, USA
yin.wang@hp.com

Ahmed Nazeem
Georgia Institute of Technology
Department of ISyE
Atlanta, GA, USA
anazeem@gatech.edu

Ram Swaminathan
Hewlett-Packard Labs
1501 Page Mill Rd
Palo Alto, CA, USA
ram.swaminathan@hp.com

ABSTRACT

Service composition has received significant attention in the research community, but the focus has been on service semantics and composition algorithms, and the problem of representation of the composition outcome has been largely ignored. Ad hoc workflows are often employed, which typically sacrifice alternative paths and parallelism for the sake of simple representation. In this paper, we show how *theory of regions*, which was originally developed to derive *Petri nets* from finite state automata, can be applied to find the optimal representation of composition. To apply the theory, we first propose an automaton-based composition framework that incorporates most existing composition techniques without changing the service semantics or its description language. Then based on the special requirements of the composition representation, we develop our own Petri net synthesis algorithm that combines the benefits of two well known algorithms from the theory of regions. More specifically, our algorithm converts an automaton to an *unlabeled Petri net*, whenever such a conversion exists, and reduces the number of places in the net. We demonstrate that workflow-based representation can limit the concurrency even for simple input/output based service composition, while our Petri net-based representation is optimal in terms of flexibility and parallelism. Our experimental evaluations include a case study on composing Google Checkout Service, and the study on Oracle BPEL samples, for which our algorithm obtains better concurrent representations for almost all non-trivial cases.

1. INTRODUCTION

In the Service Oriented Architecture (SOA), workflows are widely used to organize and orchestrate services to achieve business objectives. A workflow language, e.g., BPEL [3], defines a set of activities, including service invocation, user interaction, and value assignment. These activities are arranged by constructs such as sequence, AND fork/join, OR fork/join, and loops. Workflows are often constructed manually, which is a tedious, time-consuming, and error-prone process. For example, Google estimates up to four weeks to integrate its checkout service with a merchant order processing system, despite its ample documentation and wide adoption. Manually composed workflows are poorly optimized, and maintaining these workflows is even more difficult.

As services become increasingly abundant, especially due to the recent boom in cloud services, automated service composition becomes the key to scale. To address this challenge, numerous composition methods have been proposed in the literature. Automated service composition relies on service models that describe the semantics of services. Existing service models can be largely divided into three categories: input/output (I/O) models [33, 34, 21, 23], precondition/effect (P/E) models [29, 4, 25, 31], and stateful (e.g., automaton) models [12, 9, 30, 7, 6, 20, 27, 22]. Different service models result in different composition algorithms that generate the composite service to achieve a given goal. Most of the existing work focus on service models and composition algorithms, and usually, the output composite service is often represented by some ad-hoc workflows. These workflows may not present all possible paths to achieve composition goals, and thus describe little or no concurrency.

Because of the poor quality of both manually and automatically composed workflows, in this paper, we restrict our focus to the optimal representation for service composition. We propose a generic automaton-based composition framework that incorporates I/O, P/E and stateful service models by automatically converting them into component automata. For the composition, our framework uses an automaton integration operation, called *parallel product*. First, composition goals specified in different service model languages are translated into our framework as goal states in component automata, and then, our composition algorithm selects and integrates relevant components into one composite automaton. This automaton preserves all feasible paths that achieve the goal, but it can be very large and difficult to understand. Furthermore, automaton models do not express concurrency explicitly. Therefore, our last step converts the composite automaton into an *unlabeled Petri net* using a synthesis tool we develop based on the *theory of regions*.

The theory of regions [17, 16, 8], developed in the 90s, is well known for its ability to derive Petri nets from automata. Comparing with automata, Petri nets are much more compact and are concurrent in nature. Comparing with workflows that use limited constructs, Petri nets are more expressive. In the application of service composition, we show that even with simple I/O models, workflows do not allow full concurrency in general. We also prove that when the conversion from an automaton into an unlabeled Petri net exists, the synthesized net is maximally flexible and maximally concurrent, i.e., it preserves all possible paths and allows all possible concurrent executions as the composite automaton permits.

Our contribution is on the optimal representation for service composition. In particular, we propose an automaton-based composition framework that incorporates all popular services models to facilitate the application of Petri net synthesis. Our framework converts different service models into automata preserving the semantics yet allowing maximum flexibility and concurrency. For the Petri net synthesis, based on the special requirement of the composition representation, we develop a customized algorithm using the theory of regions. In addition to finding the optimal representation, our algorithm also reduces the number of places and the number of arcs in the synthesized Petri net. To execute the com-

posed service, we have implemented a Petri net execution engine in our Web2Exchange framework [35]. We note that while the theory of region has been applied to process mining [13], this is the first paper to apply this theory to service composition.

Since manual composition is the norm today, to demonstrate the value of our method, we designed two service composition scenarios. The first one is for services with detailed but unstructured descriptions, and here we use Google Checkout Service [1] as a case study. We show that these service descriptions naturally map to I/O or P/E service models that capture the semantics. The second scenario is based on existing manually composed workflows, for which we study Oracle online BPEL samples [2]. We follow the principle of artifact-centric design [11], and construct automata that represent the life cycles of data objects in these workflows. After composition and synthesis, the Petri nets constructed by our algorithms often exhibit better concurrency yet preserve the original semantics. We therefore argue that developers should follow the artifact-centric design and develop these life-cycle automata instead, and let our tool handle the composition task. As we obtain these life-cycle automata automatically from existing workflows, as a byproduct, our tool can be used to optimize manually composed workflows. These two sets of experiments show that our composition framework is flexible enough to incorporate real world complicated services, and that our synthesis algorithm scales to composition problems of practical size.

The rest of the paper is organized as follows. Section 2 discusses the background related to automaton and Petri net synthesis. Section 3 describes how to model and compose services using automata. Based on the theory of regions, Section 4 presents our Petri net synthesis algorithm and its proof of correctness, and Section 5 presents our experimental results for Google checkout and BPEL workflow samples. Section 6 discusses related work and Section 7 concludes the paper with a summary of the results.

2. BACKGROUND

In this section, we define parallel product and Petri net, and discuss theory of regions. Due to limited space, we keep the discussion at a high level and focus on the intuition and the relevance to our method.

2.1 Automaton and Parallel Product

We assume readers are familiar with finite state automaton. An automaton g is a triple $(S_g, \Sigma_g, \Delta_g, s_{0g})$, where S_g is the (finite) set of states, Σ_g is the set of event labels, partial function $\Delta_g : S_g \times \Sigma_g \rightarrow S_g$ is the transition function, and s_{0g} is the initial state. Assuming component services are modeled by automata, service composition is based on the *parallel product* operation.

DEFINITION 1. *The parallel product of automata g and h is an automaton $g||h = (S_g \times S_h, \Sigma_g \cup \Sigma_h, \Delta_{g||h}, (s_{og}, s_{0h}))$*

$$\Delta_{g||h} : (s,t) \times \alpha \rightarrow \begin{cases} (s', t) & \text{if } \Delta_g(s,\alpha) \text{ is defined} \\ (s, t') & \text{if } \Delta_h(t,\alpha) \text{ is defined} \\ (s', t') & \text{if both are defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $s' = \Delta_g(s,\alpha)$, $t' = \Delta_h(t,\alpha)$ are successor states of s and t, respectively.

The above definition extends to the parallel product of more than two automata in a natural way. We drop the subscripts g and h hereafter when the discussion involves only one automaton.

2.2 Petri net

We present the basic definition of Petri net and related concepts here; see [28] for an excellent tutorial. Petri nets are bipartite directed graphs with two types of nodes: circles represent *places* and solid bars represent *transitions*. *Tokens* in places are shown as dots.

DEFINITION 2. *A Petri net $N = (P, \Pi, A, M_0)$ is a bipartite graph, where P is the set of places, Π is the set of transitions, $A \subseteq (P \times \Pi) \cup (\Pi \times P)$ is the set of arcs, and for each $p \in P$, $M_0(p)$ is the initial number of tokens in place p.*

A transition α in a Petri net is enabled if every input place p of α, i.e., $(p,\alpha) \in A$, has at least one token in it. When an enabled transition α fires, it removes one token from every input place p of α, and adds one token to every output place q of α, i.e., $(\alpha, q) \in A$. The Petri net in Definition 2 is *ordinary*. A non-ordinary Petri net $N = (P, \Pi, A, W, M_0)$ assigns weight to each arc $W : A \rightarrow \mathbb{N}$. In this case, the firing of a transition α will take $W(p,\alpha)$ tokens from each input place p, and replenish $W(\alpha, q)$ tokens to each output place q.

Let $P = \{p_1, \cdots, p_n\}$, the marking (i.e., state) of a Petri net, which records the number of tokens in each place, is represented as a column vector M of dimension $n \times 1$ with non-negative integer entries, using a fixed order for the set of places: $M = \left[M(p_1) \cdots M(p_n)\right]^T$, where T denotes transpose. As defined above, $M_0(p_i)$ is the initial marking of p_i. A self-loop in a Petri net is a pair p, α such that $(p,\alpha), (\alpha, p) \in A$. We consider only self-loop-free Petri nets in this paper, called *pure* in the literature.

The reachable state space of a Petri net is the set of all markings reachable by transition firing sequences starting from M_0. This state space may be infinite if one or more places contain an unbounded number of tokens. We consider only Petri nets with a bounded number of tokens in this paper, called *bounded Petri net*. Given a Petri net $N = (P, \Pi, A, M_0)$, we can construct a reachability graph that is an automaton (S, Σ, Δ, s_0), where S represents all reachable markings of N from M_0, $\Sigma = \Pi$, and Δ captures the dynamics of N, such that $\Delta(M_1, \alpha) = M_2$ *iff* $\alpha \in \Pi$ is enabled at marking M_1, and the firing of α at M_1 leads to the marking M_2.

The Petri net in Definition 2 is *unlabled*. We can add labeling function L, s.t. $N = (P, \Pi, A, L, \Psi, M_0)$ where $L : \Pi \rightarrow \Psi$. The dynamics of a labeled Petri net is the same as an unlabeled one, but the reachability graph is slightly modified as $\Sigma = \Psi$ and $\Delta(M_1, \alpha) = M_2$ *iff* $\beta \in \Pi$ changes the marking M_1 to M_2 and $L(\beta) = \alpha$.

2.3 Theory of Regions

The problem of Petri net synthesis is to construct a Petri net (P, Π, A, M_0) whose reachability graph is isomorphic to a given automaton (S, Σ, Δ, s_0). In this regard, the *theory of regions* is the most extensively studied approach. The core idea of this theory is the concept called *region*. A region is a set of states in an automaton where every set of transitions with the same event label must be one of the following: i) all "enter" the region, ii) all "leave" the region, and iii) none "enters" or "leaves" the region. Let us consider first the synthesis of unlabeled Petri net, and let $\Pi = \Sigma$. Then a region maps to a place in the Petri net, where event labels that enter the region become its input transitions, and event labels that leave the region become its output transitions. Therefore, a place p has one token in some marking M if and only if the automaton state corresponding to M is in the region corresponding to p. Various algorithms have been proposed to find these regions [17, 16]. Some goes one step further to characterize the conditions needed to synthesize a Petri net with the minimum number of places [15]. These algorithms synthesize only *elementary Petri nets* that are unlabeled and safe, i.e., no more than one token in one place in any

reachable marking. In addition, one can synthesize unlabeled and bounded (not necessarily safe) Petri nets using a generalized notion of region [8]. While the state set based region maps every event label to one of the three cases, "enter," "leave," and "irrelevant," the generalized region maps a label to an arbitrary integer, i.e., it is represented as an integer vector over all event labels. During synthesis, the vector region still maps to a place p in the Petri net, and its vector represents p's connectivity with all transitions. Bounded unlabeled Petri net is a superset of elementary net, but its reachability graphs are still a strict subset of all automaton models. Therefore, the conversion from an automaton to a bounded unlabeled Petri net, or an elementary net, may not exist. In this case, we can either relabel conflicting transitions and synthesize a labeled Petri net [15] or prune the automaton and still synthesize an unlabeled Petri net. Relabeling reduces concurrency, even though the reachability graph remains isomorphic to the given automaton. Pruning loses both flexibility and concurrency.

3. AUTOMATON-BASED SERVICE COMPOSITION FRAMEWORK

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. A component service typically consists of a set of atomic *operations*. Automated composition approaches are based on service models that characterize component services and their operations, which can be divided into the following three categories: (i) *Input/Output* (I/O): an operation of a service is modeled as a pair of input and output sets, which are identified by the data schema; (ii) *Precondition/Effect* (P/E): an operation of a service is modeled as a pair of precondition and effect sets, which are logic literals representing typically the state of the component service; and (iii) *Stateful*: a component service is modeled by stateful models, e.g., finite state automata, to describe its state, where its operations are transitions in the automata that change its states. We list the above three categories in the order of increasing expressive power, as we will see next that more expressive models can capture the semantics of less expressive models [37]. However, expressive models are hard to construct and their composition algorithms in general have higher computation complexity. In practice, there is no widely adopted standard for the purpose of service composition. Table 1 summarizes the tradeoff.

Next we briefly discuss how to translate different service models into automata and use parallel product for the composition, the result of which is consistent with the semantics the underlying service model encapsulates. Therefore, instead of designing different Petri net synthesis algorithms for various service models separately, we can focus on the synthesis problem of automaton models.

3.1 Input/Output Model

In the input/output service model, an operation α of a service is define by an I/O pair (I_α, O_α), where I_α and O_α, are the input and the output data set, respectively. The execution semantics of the I/O model is such that in order to execute α, I_α must be generated by services executed preceding α. After its execution, O_α is added to the set of available data. To compose I/O services using automata, we construct an automaton for each data type as Figure 1 shows.

An automaton for a data type, say d, has two states representing the availability of d. The initial state represents the unavailability of d, where operations that generate d as an output can take place and move the automaton to the final state representing the availability of d. Operations that require d as input can take place only at this state. In addition, operations that output d can still occur in the final state since otherwise they would be prohibited after the parallel product operation. The way we model I/O services by automata guarantees that an operation generating some data will always precede any operation that requires the data. Moreover, parallel product preserves this ordering precedence. We use the following running example to illustrate the idea.

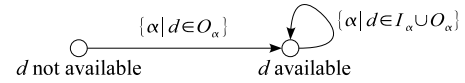
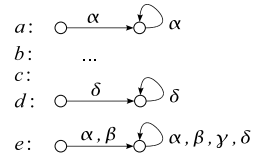


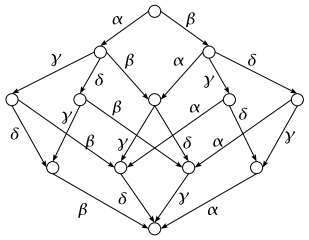
Figure 1: The automaton for data d in I/O models

	I	O
α	$\emptyset$	$\{a,e\}$
β	$\emptyset$	$\{b,e\}$
γ	e	$\{c\}$
δ	e	$\{d\}$

(a) I/O model



(b) Automaton



(c) Parallel product of the five automata

Figure 2: Example 1 and its automaton models

Example 1. There are four operations $\alpha, \beta, \gamma, \delta$, with I/O pairs described in Fig. 2a. The automata for the five data types a, b, c, d, e are shown in Fig. 2b (automata for b, c are similar to those for a, d and therefore omitted). The parallel product of the five automata is displayed in Fig. 2c, where self-loops are omitted. The parallel product preserves not only the ordering precedence as defined by the I/O model, but also all the feasible paths and parallelism, as we will see later in Section 4.

3.2 Precondition/Effect Model

The precondition/effect(P/E) model describes the semantics of services using propositional literals. Formally, the P/E model of an operation α is a triple $(P_\alpha, E_\alpha^+, E_\alpha^-)$, where P_α is the set of literals representing preconditions, E_α^+ and E_α^- represent positive and negative effects, respectively. We separate positive and negative effects to facilitate the use of set operations.

The execution semantics of the P/E model is defined as follows. We assume that the current state T is defined as a set of literals that are true in the state. Literals not in T are assumed to be false (*closed-world* assumption). Operation α can take place in T if $P_\alpha \subseteq T$, and once α takes place, the next state is defined as $T \cup E_\alpha^+ - E_\alpha^-$. In other words, to execute an operation α, all literals in P_α must be true in the current state. After its execution, E_α^+ is added to the state, while E_α^- is removed.

The automaton for P/E service model is similar to the automaton representing I/O models, as shown in Figure 3. There are still two states, representing false and true values of literal l, respectively.

Model	Features	Service Description	Standard
Input/Output	interface only, service has no semantics	input and output data schema	WSDL
Precondition/Effect	semantics only, service is stateless	preconditions and effects, situation calculus	OWL-S (draft)
Stateful	service has states, most expressive	states and transition function	

Table 1: Summary of service models for composition

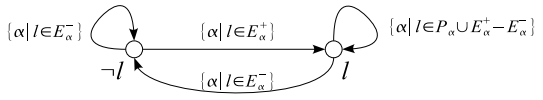


Figure 3: The automaton for literal l in P/E models

Operations that have l in their positive or negative effect set will move the automaton to the corresponding states. Operations that require l as a precondition can only take place when l is true.

In practice, enumeration data types, instead of boolean values, are often used for expressing preconditions and effects. For example in Google Checkout Service, if the order status is "chargeable," the merchant can issue the "charge" operation and the effect is that the status becomes "charged." With an enumeration type, instead of encoding it into boolean formulae, it is more efficient to use one automaton for the variable. More specifically, each state (possibly more than two) in the automaton represents a possible value of the enumeration type, and transitions may change its value; see Section 5 for a real example.

P/E service models in practice often have conditional effects. For example, a "charge" operation may result in a successful charge or a declined payment. We capture conditional effects by branches in the automaton model. More specifically, we split one operation into one transition that represents its invocation, and a set of successor transitions that represent different effects.

3.3 Stateful Models

As we mentioned in the beginning of this Section, a service typically consists of multiple atomic operations. While both I/O and P/E models describe each operation separately, stateful models consider the service as one system with states, and its operations are transitions that can change its state. For example, in the credit card charge example, instead of using precondition "chargeable" and effect "charged" to describe the operation, we may directly write down the automaton with states "chargeable," "charged," and transition "charge" in between. Therefore, the stateful service model shares the same set of transitions with the automata we use for I/O and P/E services models, and it is merely a different way to capture the service semantics. This is the basis of our service composition framework, which allows us to compose services of different models together and use one algorithm for composition and the optimal representation.

Recently data-centric, or artifact-centric, business processes have received increasing attention [11, 19, 10]. The data-centric design centers around data objects and their life cycles, which are typically modeled by automata. We follow this design principle and briefly discuss its role in workflow designs. This method is applied in our experiments in Subsection 5.2, where we show that even process-oriented workflows can be recomposed using data-centric design and achieve better concurrency.

Workflows are high level scripting languages that organize various operations into structures such as AND, OR, and sequence. Each workflow defines a set of variables, manipulated by its operations that includes service invocation, user interaction, value assignment, and utility functions. Following the data-centric design, we build one automaton for each variable to be used in the workflow. The automaton includes relevant operations as transitions, and captures the life cycle of the variable. After we obtain the repository of these variable automata, service composition is obtained using parallel product and Petri nets (as workflows) are constructed using net synthesis algorithms. This data-centric design provides strong composability over process-oriented designs. Building life cycle automata for each variable is considerably easier than constructing the monolithic workflow, and also, these automata can be reused for different workflows. Our net synthesis algorithm has the added advantage that the optimal representation can be constructed automatically.

3.4 Service Selection for Composition

Our service composition algorithm takes a composition goal as the input, selects relevant automaton models from the service repository, and uses parallel product to build the composite service that achieves the given goal. This subsection describes this process in detail.

Let G denote the set of component automata in the service repository. Since each component automaton in the service repository represents the life cycle of some data object, the composition task is naturally specified as pairs of initial and goal states for a subset of component automata, denoted as $G' \subseteq G$. Since this subset must be included in the composition, we start with G' and expand the set until all relevant component automata are included. Parallel product synchronizes automata on shared events, therefore all automata that share events with those in G' must be included, i.e., $G' = G' \cup \{g | g \in G \setminus G', \exists h \in G', \Sigma_g \cap \Sigma_h \neq \emptyset\}$. We continue expanding G' until no new automata can be added to G'. This is our basic service selection procedure. There is an optional pruning step that can reduce the number of component automata included in exchange for less flexible solutions. For example, we can prune *dead states* in each component automaton, which are states not reachable from the initial state or states that cannot reach the goal state. This pruning does not reduce alternative paths in the composite to reach the goal, but the composite may become undefined should the execution lead to those dead states unexpectedly. In addition, we can sacrifice alternative paths for a small composite automaton. For example with a composition task like map navigation, we may want only a simple solution rather than numerous alternatives.

The computational complexity of the above algorithm depends on the size of the final composite. The parallel product constructs the Cartesian product for the state sets of all automata involved in the operation, which dominates the computation. With many shared events among components, in practice, the state space is much smaller than the full Cartesian product. Pruning further reduce the number of automata in the final composite.

4. PETRI NET SYNTHESIS

The theory of regions is a well-studied body of work. Section 2.3 briefly discussed the relevant work and the two popular concepts of regions. Here we provide a unified view of the two types of regions, which enables us to combine the benefits of the respective net synthesis algorithms. Due to the need of our specific application domain, we also discuss the existence of the conversion and the

concurrency of the synthesized Petri net. Since our focus is on the application of theory of regions, we try to avoid much of the notation and development, and instead restrict out attention to the intuition and the practical implication; see [16, 15, 8] for the full details of the theory.

The rest of the section is organized as follows. Subection 4.1 provides our own interpretation and a unified view of the two types of regions in the literature. Based on this view, Subection 4.2 combines synthesis algorithms for different region types to obtain both a broader class of the synthesized Petri net, and the compactness. Subection 4.3 discusses the properties of the synthesized Petri net.

4.1 Regions

First we give the precise definitions of the two types of regions in the literature, the one on set based and the other on vector based.

DEFINITION 3. *(Set Region) In automaton (S,Σ,Δ,s_0), a set of states $R \subset S$ is a region iff for any pair of equally labeled transitions $\Delta(s,\alpha)=s', \Delta(t,\alpha)=t'$, the following holds:*
if $s \in R$ and $s' \notin R$ then $t \in R$ and $t' \notin R$, and
if $s \notin R$ and $s' \in R$ then $t \notin R$ and $t' \in R$.

Essentially a region is a set of states where every event in Σ has one of the three possibilities: "leaves" the region, "enters" the region, and irrelevant ("inside" the region or "outside" the region or both). If an event α "leaves" region R, we call R a pre-region of α. If α "enters" R, we call R a post-region of α. The set of all pre-regions of α is denoted as ${}^\circ\alpha$, and α° for all post-regions. Given a set $\mathscr{R}$ of regions for automaton (S,Σ,Δ,s_0), we construct an unlabeled Petri net (P,Π,A,M_0) as follows. First let $P=\mathscr{R}$ and $\Pi=\Sigma$, i.e., add one place per region and one transition per event label. Then $(p,\alpha)\in A$ iff the region of p is a pre-region of α, and $(\alpha,p)\in A$ iff the region of p is a post-region of α. A place has one initial token in M_0 iff its region contains the initial state s_0. The following theorem establishes the relationship between a set of regions and the constructed Petri net.

THEOREM 1. *[16] A set $\mathscr{R}$ of set regions map to an elementary Petri net whose reachability graph is isomorphic to (S,Σ,Δ,s_0) iff the following two conditions hold:*

$$\forall s,t \in S, \exists R \in \mathscr{R} \text{ such that } s \in R, t \notin R \text{ ors} \notin R, t \in R \quad (1)$$

$$\forall s \in S, \alpha \in \Sigma, \Delta(s,\alpha) \text{ undefined} \rightarrow \exists R \in \mathscr{R}, R \in {}^\circ\alpha, s \notin R \quad (2)$$

Equation (1) is the *state separation* condition, which guarantees that different states in the automaton map to different markings in the Petri net. This is achieved by finding a region that includes one state but not the other. Therefore its corresponding place will have one token in one state (marking) and zero token in the other. The state separation is unnecessary for our problem domain and therefore we do not include it in our algorithm.[1] Equation (2) is the *event separation* condition, which guarantees that an event α not defined at a state s will be prohibited by some place in the marking that corresponds to s. This is achieved by a pre-region of α that does not include s. This region (place) is connected to α but it does not have any token in the marking that corresponds to s, and therefore α is prohibited at s.

Although the event separation condition is simple and intuitive, its implementation is not very practical because Equation 2 requires one region for each undefined pair (s,α). Sparsely connected automata would actually have more places in the converted Petri net. There is an alternative event separation condition that enables efficient place reduction techniques [15]:

$$\forall \alpha \in \Sigma,\ {}^\circ\alpha \neq \emptyset \text{ and } \bigcap_{R \in {}^\circ\alpha} R = \text{pre-states of } \alpha \quad (3)$$

where pre-states of α are states for which α is defined.

Equation (3) is equivalent to (2). However, we can now reduce the number of places by first finding all pre-regions of an event, and then selecting the minimum combination of regions that still satisfies Equation (3). The final result is still not the minimum-size Petri net though, since it is possible to combine pre-regions of different events and further reduce the number of places. However, the minimization is not a good optimization goal here because combining pre-regions of different events can destroy the nice structures in the Petri net and make it unnecessarily complex; see Fig.8 in [15]. A more serious problem with the overall approach is that it is limited to elementary Petri net only. The conversion to such a net may not exist even for the simplest I/O model-based composition. For example, the parallel product automata in Figure 2c cannot be converted into an elementary net. The event separation condition does not hold for the starting state and event γ (or δ). The solution within the set region framework is to split event labels where the event separation condition is violated, and convert the automata into a labeled Petri net. Event splitting reduces the concurrency in the converted Petri net. In the extreme case, if we give every transition in the automaton a different label, the Petri net generated would be totally sequential, as the automaton model is. Figure 4a shows the labeled Petri net generated by the popular synthesis tool Petrify, using the algorithm in [15]. Events α and β are split and therefore cannot take place in parallel after the conversion.

DEFINITION 4. *(vector region) A vector region of an automaton (S,Σ,Δ,s_0) is a mapping $V:\Sigma\rightarrow\mathbb{Z}$.*

A set $\mathscr{R}$ of vector regions of (S,Σ,Δ,s_0) maps to a non-ordinary Petri net (P,Π,A,W,M_0) as follows. Again, we first let $P=\mathscr{R}$ and $\Pi=\Sigma$. Each vector region $V\in\mathscr{R}$ represents the connectivity of its corresponding place, say p, with the transitions in the Petri net, i.e., $(\alpha,p)\in A$ and $W(\alpha,p)=V(\alpha)$ if $V(\alpha)$ is positive, or $(p,\alpha)\in A$ and $W(p,\alpha)=-V(\alpha)$ if $V(\alpha)$ is negative, otherwise there is no arc between p and α if $V(\alpha)=0$. With the above Petri net construction, we observe that if we follow a path u of transition firing sequence, the token change in a region (place) V would be the weighted sum of the integer vector of V, where the weight of event label α is the number of occurrences of α in u. With this important observation, we can find a set of regions to achieve the synthesis goal through linear programming. More specifically, first we want to guarantee that if we follow different paths to reach a state in the automaton, each region would have exactly the same number of tokens. We achieve this by finding all undirected *elementary cycles* in the automaton, and formulate them as equality constraints in the linear programming formula. The constraint states that each cycle evaluates to zero in the weighted sum of a vector region, i.e., going through a cycle does not change the number of tokens in a place. In addition, we need to guarantee that each place has nonnegative number of tokens in every reachable state. Therefore we add an inequality constraint to the linear programming, stating that the weighted some of a path to a state is nonnegative. With this linear programming formulation, we present a different characterization of the state separation and event separation conditions below using vector regions. For the sake of simplicity we denote the number of tokens in a vector region V in state s as V_s.

[1] Technically speaking, without (1), the reachability graph is *bisimilar* to the given automaton. Bisimulation is an equivalence relationship that is weaker than isomorphism [26]. Here we consider only automata without any bisimilar states, for which (1) is not necessary [15]. In practice, we have not seen any automaton in our service composition tasks that has bisimilar states.

THEOREM 2. *[8] A set $\mathscr{R}$ or vector regions maps to a bounded Petri net whose reachability graph is isomorphic to (S,Σ,Δ,s_0) iff the following two conditions hold:*

$$\forall s,t \in S, \exists V \in \mathscr{R} \text{ such that } V_s \neq V_t \quad (4)$$

$$\forall s,\alpha,\Delta(s,\alpha) \text{ undefined} \rightarrow \exists V \in \mathscr{R}, V_s + V(\alpha) < 0 \quad (5)$$

The state separation condition in (4) ensures that for each pair of states, there is at least one place such that its number of tokens in the two corresponding markings would be different, i.e., different states cannot map to the same marking. Again this condition is unnecessary for our problem domain. The event separation condition in (5) states that for every undefined state event pair (s,α), there is a place in the Petri net with insufficient number of tokens to disable α at the marking corresponding to s.

Vector region and its linear programming method can convert automata into unbounded Petri nets, which is a superset of elementary Petri nets. However, it suffers from the same problem with the conditions in (1-2) that the number of places generated can be large. The place reduction technique corresponding to (3) does not apply to vector regions. Moreover, the vector obtained from the linear programming method may have too many non-zero entries, which map to numerous arcs connected to the place, and thus complicate the Petri net representation. We believe the best solution is to start with the set region algorithm that is based on condition (3) and resort to vector region and the linear programming method when the automaton cannot be converted into a Petri net. Before delving into the details of our algorithm, we present a unified view of the two types of regions to conclude this subsection.

The notion of vector region is a generalization of the notion of set region. One can convert a set region R to a vector region V as follows, $V(\alpha) = 1$ if R is a post-region of α, $V(\alpha) = -1$ if R is a pre-region of α, otherwise $V(\alpha) = 0$. Therefore, the Petri net generated by the translated vector regions is the same as the Petri net generated by the set regions directly. Moreover, from the unified view of set regions and vector regions, we have a simple event separation condition.

THEOREM 3. *An automaton $g = (S,\Sigma,\Delta,s_0)$ is isomorphic to the reachability graph of some unlabeled and bounded Petri net if for each pair of s,α, if $\Delta(s,\alpha)$ is undefined, there exists a set or a vector region that satisfies (2) or (5), respectively.*

PROOF. (sketch) We prove the result by constructing an unlabeled Petri net $N = (P,\Pi,A,W,M_0)$ from the regions obtained for all pairs of s,α where $\Delta(s,\alpha)$ is undefined. Let $\Pi = \Sigma$, we map each set or vector region to a place in the Petri net according to the methods described in this subsection. Then we show that the reachability graph of N is isomorphic to g. More specifically, if $\Delta(s,\alpha)$ is not defined, the corresponding region will map to a place that disables α at the marking corresponding to s. Otherwise if $\Delta(s,\alpha)$ is defined, no place prevents the transition from firing. □

4.2 Our Synthesis Algorithm

Theorem 3 is the basis for our synthesis algorithm that combines both set regions and vector regions. Furthermore, we can apply the optimization techniques based to condition (3) to reduce the number of set regions. Algorithm 1 displays this procedure. It utilizes the same framework as the algorithm in Figure 10 of [15]. The framework first finds all pre-regions of an event, and checks whether condition (3) is satisfied. If not it splits the event and tries to resolve the conflict. The procedure iterates until (3) is satisfied for all the events. Our new contribution is at lines 6-8, where for each violation of the event separation condition, we first check if there is a vector region that can satisfy the condition. The check is performed by the linear programming formulation described in [8].

Algorithm 1 Petri net synthesis algorithm

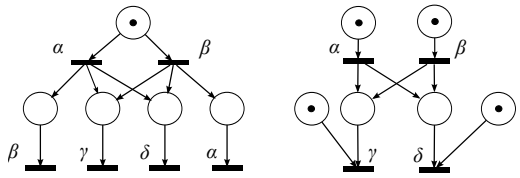
Input: Automaton $g = (S,\Sigma,\Delta,s_0)$
Output: Petri net $N = (P,\Pi,A,M_0)$, where $\Pi = \Sigma$, and the reachability graph is isomorphic to g.

```
1: for all α ∈ Σ do
2:    R = all minimal pre-regions of α
3:    E = ⋂_{R∈R} R − {pre-states of α}
4:    if E ≠ ∅ then
5:       for all s ∈ E do
6:          solve event_seperation_linear_programming(s, α)
7:          if feasible solution found then
8:             add the solution vector region to R
9:          else
10:            split_event(α) and start all over
11:         end if
12:      end for
13:   end if
14: end for
15: remove redundant regions and map to Petri net N
```

The correctness of Algorithm 1 relies on and Theorem 3.4 in [15] and Theorem 3 in the previous subsection. Here we provide a sketch. Assume $\mathscr{R}$ is the set of pre-regions of α obtained at line 2, and $\mathscr{V}$ is the set of vector regions of α obtained at lines 5-12. For any state s where α is not defined, if $s \notin E$, there must exist a set region $R \in \mathscr{R}$ such that $s \notin R$. Since R is a pre-region of α, event separation for (s,α) is achieved by R. If $s \in E$, there will be a vector region synthesized for (s,α) in $\mathscr{V}$. Therefore the event separation condition is satisfied for all pairs of s,α where $\Delta(s,\alpha)$ is undefined. The theorem is true as a result of Theorem 3.

We apply Algorithm 1 to Example 1, and Figure 4b shows the synthesized Petri net. It allows fully concurrent execution. If we use only set regions, event splitting is necessary and the concurrency is reduced; see Figure 4a as an example outcome. It is interesting to note that if we use only structures AND, OR for this example, the result will not be fully concurrent as Figure 4b is. For example, a typical solution puts α and β in an AND structure, and γ and δ in another AND structure that succeeds the first AND. In this case γ and δ have to wait until both α and β finish, a significant performance penalty especially if one of α and β is much slower than the other. This example shows that if we use typical workflow representation for the composition result, we cannot get full concurrency even for the simplest I/O service model.

For the event splitting at line 10 and the redundant region removal at line 15, we followed the same strategy in [15]. The linear programming formulation at line 6 is slightly different from the one described in [8]. Instead of a dummy objective function, we added an optimization goal that is to minimize the connection of region V with transitions in the Petri net, i.e., minimizing the number of non-zero mappings in V. This way, the Petri net obtained has fewer arcs and is much simpler. In many cases, our Petri net synthesized is essentially a workflow net that can be converted into workflow languages like BPEL straightforwardly. However, with the new objective function, the formulation becomes an integer programming problem rather than linear programming. In practice, the examples we have are small enough so we can afford the extra computation complexity. In addition to the integer programming step, line 2 needs exponential computation time in the worst case as the number of pre-regions can be exponential in the number of states. In



(a) Petrify [14] (b) Algorithm 1

Figure 4: Petri nets synthesized for Example 1

practice, however, the number of pre-regions is usually very small. This observation is shared in [15] as well.

4.3 Properties

Existing work on the theory of regions focus on the correctness of the synthesis and concurrency properties of the synthesized Petri net have not been discussed. As we have shown that event splitting can reduce concurrency, in this subsection, we want to precisely define concurrency and give conditions on when the synthesized Petri net is maximally concurrent. First we define exactly how we compare the concurrency of two Petri nets whose reachability graphs are isomorphic. Then we show that if a Petri net is unlabeled, it is maximally concurrent.

The isomorphism of reachability graphs define naturally an equivalence relationship among all labeled and unlabeled Petri nets. We denote the class of Petri nets whose reachability graphs are isomorphic to automaton g as I_g. The Petri nets in I_g, however, may exhibit different concurrent behavior. More specifically, at a marking M in the reachability graph, even though every Petri net has the same set of transitions enabled, due to isomorphism, the subsets of transitions that can execute concurrently can be different. Therefore we introduce the notation $M[\stackrel{N}{\Rightarrow}\Lambda$ for that all transitions in Λ can fire concurrently at M in Petri net N. Now we compare the concurrency of two Petri nets as follows. Given two Petri nets $N_1, N_2 \in I_g$, N_1 is no less concurrent than N_2 if for any reachable marking M in I_g, and any subset of transitions Λ, $M[\stackrel{N_2}{\Rightarrow}\Lambda \rightarrow M[\stackrel{N_1}{\Rightarrow}\Lambda$. A Petri net $N \in I_g$ is maximally concurrent if $\forall N' \in I_g$, N is no less concurrent than N'. Note that maximally concurrent Petri nets may have different representations as the set of places can be different.

We claim that if Algorithm 1 synthesizes an unlabeled Petri net then it is maximally concurrent. To prove this result, we need the following lemma.

LEMMA 1. *Consider a marking M and set Λ such that $M[\stackrel{N}{\Rightarrow}\Lambda$, then starting from M, the transitions in Λ can be executed in all possible permutation orders in N.*

PROOF. Since all transitions in Λ can fire concurrently at M, the input places of these transitions must all have sufficient tokens in M, i.e., no less than the summation of the tokens needed by each transition. Different firing sequences drain exactly the same number of tokens in these places, and therefore must be enabled. □

THEOREM 4. *For any $N \in I_g$, if N is unlabeled and self-loop free, N is maximally concurrent.*

PROOF. Proof by contradiction. Assume that N is not maximally concurrent. Let $N' \in I_g$ be a more concurrent Petri net. Therefore, there exists a set Λ and a marking M such that $M[\stackrel{N'}{\Rightarrow}\Lambda$ but $M[\stackrel{N}{\not\Rightarrow}\Lambda$. That is, there exists a transition $\alpha \in \Lambda$ that can not fire concurrently with the rest of transitions in Λ at M in N. Therefore there must exist one or more places in N that connect to p and to some other transitions in Λ, with insufficient tokens at M in N. For simplicity, let us consider one such place only, say p. Place p connects to $\Gamma \subseteq \Lambda, \alpha \in \Gamma$, such that Γ cannot fire concurrently at M. Note that p is not an output place for any transition in Γ due to our self-loop free assumption. By Lemma 1, we can pick any permutation sequence of transitions in Γ and fire them one by one in N'. Due to isomorphism, the same firing sequence must be possible in N as well. However, as p does not gain any token throughout the firing sequence, p will prevent some transition in Γ at a certain step, which is a contradiction. □

THEOREM 5. *With I/O service models, the composite automaton of any composition task can be converted to an unlabeled Petri net with isomorphic reachability graph.*

This theorem can be proved by the construction of an unlabeled Petri net for a given I/O model in a similar way as Figure 4b shows. This result proves that Algorithm 1 always synthesizes a maximally concurrent Petri net for I/O model-based compositions.

5. EXPERIMENTS

Currently there are very few services with well defined semantics available for the purpose of composition. The only exception is services with WSDL specifications, which can be viewed as I/O model. However, because of the limited semantics of the I/O model, most WSDL services are data look up services [5], for which the composition is trivial. Semantic-rich services, however, are usually described by plain text.

To evaluate the full capability of our composition framework and the synthesis algorithm, we designed two service composition scenarios. First, through the case study of Google checkout service, we show that the text documents for these semantic-rich services closely resemble I/O and P/E service models, which are easily translated into automaton for automated composition. The second scenario is for manually composed workflows, for which we experimented with Oracle online BPEL samples [2]. We built a BPEL parser to automatically extract automata that represent life cycles of data objects in these workflows, and we show that the composition obtained exhibit better parallelism for almost all non-trivial workflows. For this reason, we argue that workflow developers should follow the data-centric design principle [11], and define life cycles of data objects rather than the workflow itself.

5.1 Case Study: Google Checkout

Google Checkout is an online payment processing service that helps merchants manage their order payments. It has around 20 APIs that communicate to the merchant through HTTP PUT and GET commands. The parameters of each API can be sent through name value pair in the HTTP request, or in a separate XML message. These APIs are designed with extreme flexibility such that merchants of various size and complexity can use the service. The simplest case could be a lump sum payment for each order. The complicated case involves per item order processing that handles operations such as credit authorization, declined payment, partial charge, back order, shipping, return, and refund. Exactly because of the flexibility, there is a steep learning curve on using these APIs. Google estimates up to four weeks to integrate its checkout service with the merchant's shopping portal [1]. Different order processing systems result in different integration, and the checkout service itself is evolving. This makes the whole system extremely complex and hard to maintain. Our goal is to model the checkout service in our composition framework, such that merchants only need to

Financial State	Valid Actions	Description
REVIEWING	None	Google is reviewing the order
CHARGE-ABLE	authorize-order, cancel-order, charge-and-ship	The order is ready to be charged
CHARGING	None	May result in CHARGED or PAYMENT_DECLINED state
CHARGED	refund-order	The order has been charged

Table 2: Financial order states table (partial) from [1]

specify their specific order processing systems, and our composition engine handles the integration.

Google checkout APIs are described by reference documents in plain text. However, we show that these documents closely resemble the I/O and P/E service models, and the translation is straightforward. We believe the translation can be automated if there are proper structure and syntax added to the documents. Many APIs provide simple stateless calculation, which can be captured by I/O service models. For example, shipping cost and tax calculations are stateless APIs where the input is the shopping cart and the output is the cost for shipping and tax, respectively. These I/O models can be constructed automatically through the analysis of XML schema. Order processing and financial command APIs are the core of the checkout service. Simple I/O models cannot capture their semantics. Table 2 is a part of the financial status summary table taken from the API reference online, with simplified descriptions. It shows financial states side by side with the list of commands available in a state. Precondition/effect models capture these enumeration data type well, as discussed in Section 3.2. The automaton translated from the P/E model is displayed in Figure 5.

For the purpose of composition, we constructed a basic merchant order handling process as follows. After charging the order, the merchant checks inventory to see if the items are available, if not, it may cancel or mark the order as back ordered. Otherwise the order will be shipped. After shipping, upon receiving the order return notification, the merchant will refund the customer and cancel the order. Figure 6 shows the automaton that models the process. In addition, we have a few WSDL-based data lookup services that calculate taxes, shipping cost and coupons, to be integrated together with the merchant and the checkout services. These services are captured by I/O models, and subsequently translated into automata model.

In the service composition phase, our service selection algorithm picked twelve Google checkout APIs that are necessary for the completion of the aforementioned basic merchant process. There are a total of 20 component automata used for the parallel product integration. The composite automaton has 98 states, 134 transitions and 31 event labels. It turns out that this automaton cannot be converted into an unlabeled Petri net for maximum concurrency. Algorithm 1 had to split events and iterate. The final result contains 43 transitions rather than the original 31 events. The overall computation takes a few seconds. In comparison, Petrify, the set region based tool, generates a Petri net with 49 transitions, which is less concurrent. This case study shows that our composition framework incorporates real services well, and that our Petri net synthesis algorithm adds the benefit of better concurrent representation.

5.2 Oracle BPEL Workflow Samples

Section 3.3 discussed the principles of data-centric design, and how we recompose process-oriented workflows based on the design. Here we apply the technique to real BPEL workflows and discuss implementation details and BPEL specific issues.

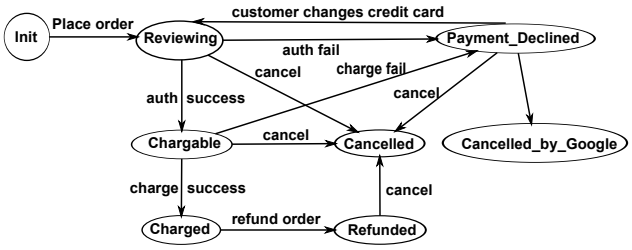


Figure 5: The automaton for Google financial status variable

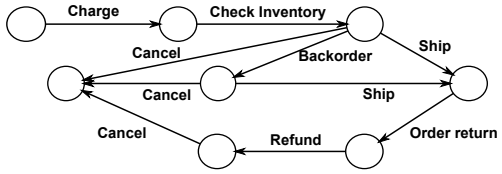


Figure 6: The automaton for merchant order processing

Our model extraction method has to ensure that after recomposition, the composite service produces the same result. We follow two rules for this purpose. First, each read in the workflow must see the same write. This analogous to the concept of *view serializability* in the database literature [32]. Second, invocations of an external service, called *partner links* in BPEL specification, must follow the same order. The second rule is conservative especially if the external service is stateless, in which case the invocation order does not matter. However, we require this rule as we assume no knowledge about external services.

Based on the above two rules, we extract models for both variables and partner links in a BPEL program. The transitions of these automata are activities used in the program. There are around 20 activities defined in BPEL specification [3], including basic activities such as `receive`, `reply`, `invoke`, and `assign`, and structured activities such as `sequence`, `flow`, and `switch`. In addition, some examples include Oracle's BPEL extensions. Basic activities access variables either explicitly through an attribute, e.g., `variable="replyInput,"` or through functions in an expression, e.g., `bpws:getVariableData('input',...)`. These activities also specify whether the access is a read or write. Structured activities define the ordering relationship of basic activities. With this understanding, the modeling is straightforward in most cases. For example in a sequence structure, if a read follows a write, or a write follows a read, we need to put the two activities in a sequence in the automaton model for the accessed variable, otherwise the read could see a different write. If multiple reads occur consecutively, we include all possible interleavings of these reads in the automaton. Less obvious is when multiple writes occur consecutively, and in this case, we still include all possible interleavings in the automaton. Because these writes are in fact modifying different segments of complex data types, which can proceed in parallel.

We implemented a model extraction tool for BPEL by parsing the XML file. We applied the tool to 194 BPEL samples downloaded from Oracle BPEL designer website [2].[2] These samples are divided into categories including "demos", "references," "tu-

[2]We collected these distinct samples from various links on Oracle website over the past two years. As the Oracle BPEL designer software and its supplementary online contents are constantly changing, one may find only a subset of these samples online to repeat our experiments.

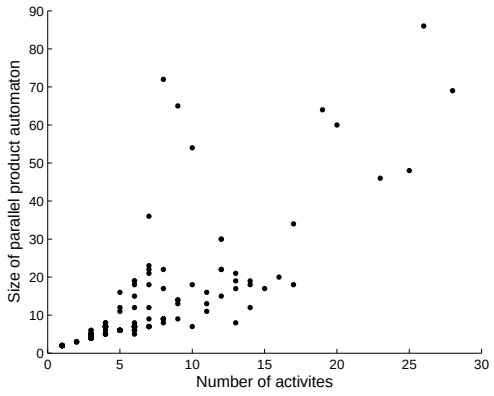


Figure 7: Automaton size of 192 Oracle BPEL examples

torials," and "utils." Our model extraction tool successfully parsed 192 of them. One example caused a `SAXParseException` in the XML parser, and another contains a `link` structure that we do not handle yet. Most of these samples are very small, with no more than 10 activities, and even less variables. After extracting lifecycle automata for variables and partner links, the parallel product of these automata contains less than 100 states, except one example with a state size of 1,186. It is an `XPath` example in the "reference" category, which contains value assignments to many different variables that can be parallelized. All possible interleavings of the parallel access caused the state explosion. We remove the outlier and draw the scatter plot in Figure 7.

Our Petri net synthesis algorithm successfully converted all 192 composite automata to Petri nets using only set regions. The linear programming function for vector regions was never invoked. Due to the small size of these examples, the calculation takes less than a second in each case. To compare our results with the original workflow, we manually examined all 31 examples in the "demos" category, which contains some of the largest examples in our collection. There are 18 cases where our Petri nets are more concurrent, and the rest 13 is exactly the same. These 13 cases are mostly trivial examples with less than 10 states in the composite automaton, organized as a sequence. For the 18 cases that get better concurrency, the most common source of optimization is value assignments to different variables or different portions of the same variable that can take place in parallel. Another common pattern is a generic reply message that does not depend on any computation, and therefore can be sent early. There are a few cases where different service invocations can happen simultaneously. Interestingly, we discovered a case where we believe that the programmer forgot to put an output variable in a service invocation to store the result. Therefore, the service invocation becomes independent with the subsequent activities, and our synthesis tool fully exploited the optimization opportunity.

For the purpose of illustration, we picked one example, called "CheckoutFlow," in the "demos" category. The code snippet is displayed in Figure 8. The whole workflow contains one big sequence structure with 14 activities in total. The figure shows the middle part with 6 activities. The first three activities, `invoke`, `assign`, and `reply`, must take place in order because the output of the previous activity is the input of the next. The next activity, `receive`, has to follow `reply` as well since they both invoke the same partner link `client`. The next activity, the second `invoke`, only depends on the first `invoke` as they use the same partner link.

```
<sequence name="main">
...
<invoke partnerLink="CRMService" ... inputVariable=
  "crmRequest" outputVariable="crmAddressResponse"/>

<assign><copy>
  <from variable="crmAddressResponse" part="payload"/>
  <to variable="replyInput" part="payload"/>
</copy></assign>

<reply partnerLink="client" ... variable="replyInput"/>
<receive partnerLink="client" ... variable="continue"/>

<invoke partnerLink="CRMService" ... inputVariable=
  "crmRequest" outputVariable="crmCreditCardResponse"/>

<assign><copy>
    <from variable="continue" part="payload"/>
    <to variable="input" part="payload"/>
  </copy><copy>
    <from variable="crmCreditCardResponse" part="..."/>
    <to variable="replyContinue" part="payload"/>
</copy></assign>
...
</sequence>
```

Figure 8: Code snippet of one Oracle BPEL example

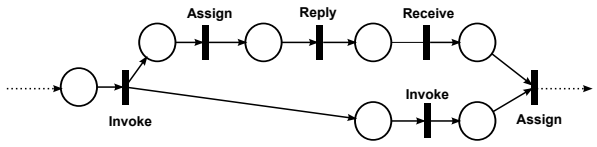


Figure 9: Petri net (partial) of the Oracle BPEL example

Therefore it can take place in parallel with the preceding four activities. The last `assign` activity must wait for all the preceding activities to finish, as it has two input variables that depend on both branches. All these ordering constraints are enforced by the parallel product of component automata that models these variables and the partner links, in a similar fashion as Example 1 demonstrates. Figure 9 shows the part of the final synthesis result that corresponds to this code snippet.

There are additional optimization opportunities in Fig. 8, which our implementation does not include currently. For example, as mentioned earlier in this subsection, we enforce the same invocation ordering for any external service. Here in this example, obviously the two `invoke` activities can be parallelized as both requests are read only. In addition, the last assignment activity includes two copy operations that operate on different variables. As we do not modify activities, parallelism inside activities is not exploited.

6. RELATED WORK

Existing service composition methods are based on I/O models, P/E models, and stateful models. The detailed comparison of these models can be found in our study [37]. The semantics of I/O and P/E models are relatively consistent in the literature. A few exceptions include the I/O model where the output is consumed by an operation rather than copied for repeated use [34], and another approach that mixes the I/O and P/E models [24]. There is much more variation in the use of stateful models. Parallel product for service composition has been discussed in the literature [30], and there are other composition techniques using asynchronous messages [12] and action delegation [9]. In all these cases, the composition outcome is still an automaton and therefore the net synthesis technique we develop can be applicable. Composition methods

using stateful models other than automaton include Petri nets [7] and workflows [6]. There is also a dependency graph model based composition [18], which can be viewed as a simplified automaton model.

In the workflow domain, Petri nets are widely used to model and analyze workflows [36]. Net synthesis techniques have been applied to process mining to construct workflows from event logs [13], but not for service composition. Finally, we note that Process Algebra has been widely used to model and analyze concurrent systems [26]. The core notion, *bi-simulation* equivalence, is stronger than language equivalence and it captures the behavior of concurrent systems. However, the notion does not capture the concurrency equivalence as we showed that two Petri nets may not be equally concurrent even if their reachability graphs are isomorphic, where isomorphism is even stronger than bi-simulation.

7. CONCLUSIONS

In this paper, we studied the representation problem for service composition and showed how theory of regions, can be applied to find the optimal representation of composition. To apply the theory, we first proposed an automaton-based composition framework that incorporates most existing composition techniques without changing the service semantics or its description language. Then based on the special requirements of the composition representation, we developed our own Petri net synthesis algorithm that combines the benefits of two well known algorithms from the theory of regions. We demonstrated that workflow-based representation can limit the concurrency even for simple input/output based service composition, and we proved that our Petri net-based representation is optimal in terms of flexibility and parallelism. Our experimental evaluations, which include a case study on Google Checkout Service, and the study on Oracle BPEL samples, demonstrates that our algorithm obtains better concurrent representations for almost all non-trivial cases.

8. REFERENCES

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

[2] Oracle online BPEL samples. `http://soasamples.samplecode.oracle.com/`.

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

[4] V. Agarwal and et al. A service creation environment based on end to end composition of Web services. In *WWW*, pages 128–137, 2005.

[5] E. Al-Masri and Q. H. Mahmoud. Investigating Web services on the world wide web. In *WWW*, pages 795–804, 2008.

[6] P. Albert and et al. A constrained object model for configuration based workflow composition. In *BPM Workshops*, pages 102–115, 2005.

[7] P. Álvarez and et al. 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.

[8] E. Badouel and P. Darondeau. Theory of regions. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 529–586, London, UK, 1998. Springer-Verlag.

[9] D. Berardi and et al. Automatic composition of e-services that export their behavior. In *ICSOC*, pages 43–58, 2003.

[10] P. Bertoli and et al. Control flow requirements for automated service composition. In *ICWS*, pages 17–24, 2009.

[11] K. Bhattacharya and et al. Towards formal analysis of artifact-centric business process models. In *BPM*, pages 288–304, 2007.

[12] T. Bultan and et al. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW*, pages 403–410, 2003.

[13] J. Carmona and et al. A region-based algorithm for discovering petri nets from event logs. In *BPM*, pages 358–373, 2008.

[14] J. Cortadella and et al. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, 80:315–325, 1997.

[15] J. Cortadella and et al. Deriving petri nets from finite transition systems. *IEEE Trans. Comput.*, 47(8):859–882, 1998.

[16] J. Desel and W. Reisig. The synthesis problem of petri nets. *Acta Inf.*, 33(4):297–315, 1996.

[17] A. Ehrenfeucht and G. Rozenberg. Partial (set) 2-structures. *Acta Informatica*, 27:315–368, 1990.

[18] R. Eshuis and et al. Structured service composition. In *BPM*, pages 97–112, 2006.

[19] C. Fritz and et al. Automatic construction of simple artifact-based business processes. In *ICDT*, pages 225–238, 2009.

[20] G. D. Giacomo and et al. Automatic workflows composition of mobile services. In *ICWS*, pages 823–830, 2007.

[21] Z. Gu and et al. Automatic service composition based on enhanced service dependency graph. In *ICWS*, pages 246–253, 2008.

[22] R. R. Hassen and et al. Protocol-based Web service composition. In *ICSOC*, pages 38–53, 2008.

[23] R. Hewett and et al. Scalable optimized composition of Web services with complexity analysis. In *ICWS*, pages 389–396, 2009.

[24] S. Kona and et al. Generalized semantics-based service composition. In *ICWS*, pages 219–227, 2008.

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

[26] R. Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[27] S. Mitra and et al. Automated choreographer synthesis for Web services composition using I/O automata. In *ICWS*, pages 364–371, 2007.

[28] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

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

[30] M. Pistore and et al. Automated synthesis of composite BPEL4WS Web services. In *ICWS*, pages 293–301, 2005.

[31] A. Ragone and et al. Fully automated Web services orchestration in a resource retrieval scenario. In *ICWS*, pages 427–434, 2005.

[32] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2007.

[33] A. Riabov and et al. 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] V. Srinivasmurthy and et al. Web2exchange: A model-based service transformation and integration environment. pages 324 –331, Sept. 2009.

[36] W. M. P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[37] Y. Wang and et al. A language-based framework for analyzing service representation models and service composition approaches. In *IEEE International Conference on e-Business Engineering*, 2010.