On the Optimal Petri Net Representation for Service Composition

Yin Wang
Hewlett-Packard Labs
Palo Alto, CA
yin.wang@hp.com

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

Ram Swaminathan
Hewlett-Packard Labs
Palo Alto, CA
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. 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.**

I. 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 detailed documentation and wide adoption. Manually composed workflows are poorly optimized, and so 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 [18], [19], precondition/effect (P/E) models [15], [17], and stateful (e.g., automaton) models [7], [16]. 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 the output composite service is often represented by some ad-hoc workflows that organize services as a straight line or with simple AND/OR structures. These workflows may not present all possible paths to achieve composition goals, and describe little or no concurrency.

Because of the poor quality of both manually and automatically composed workflows, in this paper, we aim to find the optimal representation for service composition. First, we propose an automaton-based composition framework that incorporates I/O, P/E and stateful service models by automatically converting them into component automata. Composition goals specified in different service models are translated into goal states in component automata, and our composition algorithm selects relevant components for composition. We use the *parallel product* operation for the integration. Parallel product includes all feasible paths that achieve the goal, but it can be very large and difficult to understand, and it does 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* [13], [11], [5]. The Petri net synthesized is optimal in the sense that it preserves all feasible paths in the parallel product, and it allows independent transitions (services) of different component automata to run in full concurrency.

Comparing with automata, Petri nets are much more compact and capture concurrency. 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 using only AND/OR structures 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.

Our contribution is on the optimal Petri net representation for service composition. In particular, we propose an

automaton-based composition framework that incorporates all popular service models to facilitate the application of Petri net synthesis. Our framework converts different service models into automata preserving the semantics. 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 strives to reduce the number of places and the number of arcs in the synthesized Petri net, therefore reducing the size of the composite. To execute the composed service, we have implemented a lightweight Petri net execution engine in our Web2Exchange framework [20]. We note that while the theory of region has been applied to process mining [8], 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, with 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]. Inspired by the principle of artifact-centric design [6], we automatically extract automata that represent the life cycles of objects in workflows. After recomposition and synthesis, the Petri nets constructed by our algorithms often exhibit better concurrency yet preserve the original semantics, mostly because it is difficult for developers to reason about complex workflows and fully exploit parallelism. 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 II discusses the background related to automaton and Petri net synthesis. Section III describes how to model and compose services using automata. Based on the theory of regions, Section IV presents our Petri net synthesis algorithm, and Section V presents our experimental results for Google checkout and BPEL workflow samples. Section VI concludes the paper with a summary of the results.

II. Background

A. Automaton and Parallel Product

We assume readers are familiar with finite state automaton. An automaton g is defined as $(S_g, \Sigma_g, \Delta_g, s_{0g})$, 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.

B. Petri net

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. Formally, a Petri net is represented as (P, Π, A, W, M_0), 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, $W : A \rightarrow \mathbb{N}$ represents arc weight, and for each $p \in P$, $M_0(p)$ is the initial number of tokens in p.

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. 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 $W(p,\alpha)$ tokens. When an enabled transition α fires, it removes $W(p,\alpha)$ tokens from every input place p, and adds $W(\alpha,q)$ to every output place q of α, i.e., $(\alpha,q) \in A$.

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 an n-dimensional column vector M with non-negative integer entries: $M = \left[M(p_1) \cdots M(p_n)\right]^T$. As defined above, M_0 is the initial marking. 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. Otherwise it is called *bounded Petri net*. We consider only bounded Petri nets in this paper. If the arc weight is always one and every reachable marking has no more than one token in each place, it is called *elementary Petri net*. Given a Petri net (P, Π, A, W, 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 (P, Π, A, W, M_0) is *unlabeled*. We can add set Ψ of labels and labeling function $L : \Pi \rightarrow \Psi$. 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 $L(\beta) = \alpha$.

C. Theory of Regions

The problem of Petri net synthesis is to construct a Petri net 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 theory started with the synthesis of elementary Petri net (P, Π, A, W, M_0), where $\Pi = \Sigma$. The core idea is the concept called *region*, which is a set of states in S that maps to a place in P. We call the state set a *set region*. A set region satisfies the property that identically labeled transitions in the automaton must connect to states in the region in one of the following ways: i) all "enter" the region, ii) all "leave" the region, and iii) none "enters" or "leaves" the region. In the synthesized Petri net, event labels that enter the region become the input transitions of the corresponding place, and event labels that leave the region become the output transitions. A place p has one token in some marking M if and only if the automaton state corresponding to M in the reachability graph is in the region corresponding to p. Various algorithms have been proposed to find these regions [13], [11]. Some go one step further to characterize the conditions needed to synthesize a Petri net with the minimum number of places [10]. However, not every automaton can be converted into an elementary Petri net. A generalized notion of region was developed to synthesize bounded Petri net [5]. While a set 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. We call this *vector region*. During synthesis, the vector region still maps to a place p in the Petri net, and its vector represents arc weights between p and all transitions. As bounded Petri nets are more general than elementary Petri nets, vector region converts a broader class of automata than set region does, but it is difficult to reduce the number of places in the synthesized Petri net. In both cases, the synthesized Petri nets are unlabeled. Not every automaton can be converted into an unlabeled Petri net. In this case, we can either relabel conflicting transitions and synthesize a labeled Petri net [10] or prune the automaton until it is isomorphic to the reachability graph of some unlabeled Petri net. Relabeling reduces concurrency, and pruning loses both alternative paths and concurrency.

D. Related Work

Existing service composition methods are based on I/O models, P/E models, and stateful models. The detailed comparison of these models and the relevant literature can be found in our study [23]. In the workflow domain, Petri nets are widely used to model and analyze workflows [21]. Net synthesis techniques have been applied to process mining to construct workflows from event logs [8], but not for service composition. Other than Petri nets, Process Algebra has been widely used to model and analyze concurrent systems [14]. Other concurrent models include trace structures, Hoare structures, event structures, prime algebraic domains, and asynchronous transition systems [12].

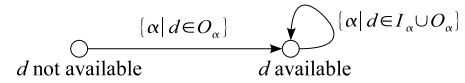


Figure 1: The automaton for data d in I/O models

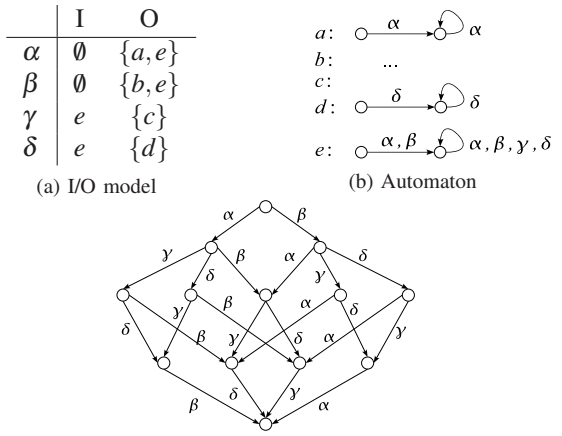
III. Automaton-based Composition

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. Our previous study shows that these models are increasingly more difficult to construct and the composition algorithm has higher computation complexity, in exchange for better expressiveness [23]. Based on this study, we propose 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.

A. Service Modeling

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 Fig. 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 re-



(a) I/O model

(b) Automaton

(c) Parallel product of the five automata

Figure 2: Example 1 and its automaton models

quire 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 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 example to illustrate the idea.

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 IV.

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. There are still two states, representing false and true values of literal l, respectively. 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 types, instead of boolean values, are often used for preconditions and effects. For example, Google Checkout Service allows an order to have status such as "chargeable," "charged," and "shipped". Instead of encoding enumeration types into boolean variables, it is more efficient to use states in the automaton to represent all possible values. More specifically, each state represents a possible value of the enumeration type, and transitions may change its value; see Section V for a real example.

Many stateful services are modeled by automata already, while others can be translated into automata in a straightforward way. For example, reachability graph can capture the semantics of services modeled by Petri nets. Next we consider workflows as stateful services and discuss how to extract component automata from them that preserve the semantics. The technique will be used in our experiments in Section V-B.

Workflows organize various operations into structures such as AND, OR, and sequence. Each workflow defines a set of objects, manipulated by its operations that include service invocation, user interaction, value assignment, and utility functions. Inspired by the artifact-centric design principle [6], we consider these objects as artifacts, and build their life-cycle automata based on the semantics of the workflow. Relevant operations in the workflow become transitions in the automaton. After we obtain the repository of these life-cycle automata, service composition is obtained using parallel product and Petri nets are constructed using our net synthesis algorithm. This artifact-centric design provides strong composability over process-oriented designs. Our net synthesis algorithm has the added benefit that the optimal representation can be constructed automatically.

B. 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 object, the composition task is naturally specified as pairs of initial and goal states for a subset of component automata, denoted as $G' \subseteq G$. 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 automaton can be added. This is our basic service selection procedure. Optionally we

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:    for all s ∈ E do
5:       solve event_seperation_linear_programming(s, α)
6:       if feasible solution found then
7:          add the solution vector region to ℛ
8:       else
9:          split_event(α) and start all over
10:      end if
11:   end for
12: end for
13: remove redundant regions and map to Petri net N
```

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. 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 the optimal 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 reduces the number of automata in the final composite.

IV. Petri Net Synthesis

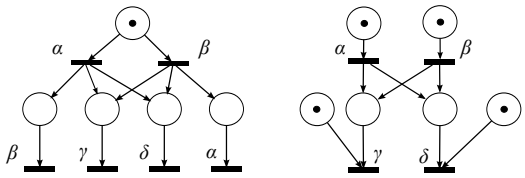
The theory of regions is a well-studied body of work. Section II-C briefly discussed the relevant work and the two popular concepts of regions, set region and vector region. Based on the requirements of our synthesis task, we developed a customized Petri net synthesis algorithm that combines the benefits of both regions. More specifically, the algorithm synthesizes both elementary and bounded Petri nets, and it strives to reduce the size of the synthesized net. Due to space limit, we try to avoid much of the notation and development, and instead restrict out attention to the intuition and the practical implication; see our technical report [22] for the full development and proofs.

Both set region and vector region algorithms try to satisfy the *event separation* condition. This condition requires that for every state in the automaton, if transition α is not allowed, there must exist a place in the synthesized Petri net such that α is disabled by the place at the corresponding marking. Our key observation is that this place can be synthesized using either set region or vector region, thus it allows us to combine the two algorithms and benefit from both. Algorithm 1 describes the procedure. It is similar to the algorithm in Fig. 10 of [10], which is based on set region. The key difference is at lines 5-7, where the event separation condition cannot be satisfied by set regions alone, our algorithm seeks vector regions first before splitting event. The algorithm we use to find vector regions is based on the linear programming formulation described in [5].

We apply Algorithm 1 to Example 1, and Fig. 3b 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 Fig. 3a as an example outcome from a popular Petri net synthesis tool. It is interesting to note that if we use only structures AND, OR for this example, the result will not be fully concurrent as Fig. 3b is. For example, a typical solution puts α and β in an AND structure, and γ and δ in a succeeding AND structure. 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 typical workflows with AND/OR structures do not allow full concurrency even for I/O service models.

For the event splitting at line 9 and the redundant region removal at line 13, we followed the same strategy in [10]. The linear programming formulation at line 5 is slightly different from the one described in [5]. Instead of a dummy objective function, we added an optimization goal that is to minimize the number of non-zero entries in the vector region, therefore minimizing the number of arcs connected to the corresponding place. This optimization reduces the size of the Petri net synthesized. In practice, this optimization often results in simple Petri nets that can be converted into workflows using AND/OR structures only. However, with an objective function, the formulation becomes an integer programming problem rather than linear programming. In addition to this integer programming step, line 2 has exponential complexity as the number of pre-regions can be exponential in the number of states. In practice, the workflows we collected are small enough such that the extra optimization is affordable. At last, we present the following result regarding the concurrency of the Petri nets synthesized. Both the proof and the correctness of Algorithm 1 can be found in [22].

Theorem 1: If the Petri net synthesized by Algorithm 1 is unlabeled and self-loop free, then the net is maximally concurrent. Furthermore, parallel product of I/O service models always result in a composite automaton that can be converted into a maximally concurrent unlabeled Petri net.



(a) Petrify [9] (b) Algorithm 1

Figure 3: Petri nets synthesized for Example 1

Financial State	Valid Actions	Description
REVIEWING	None	Reviewing the order
CHARGE-ABLE	authorize-order, cancel-order, charge-and-ship	The order is ready to be charged
CHARGING	None	Go to CHARGED or PAYMENT_DECLINED
CHARGED	refund-order	The order is charged

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

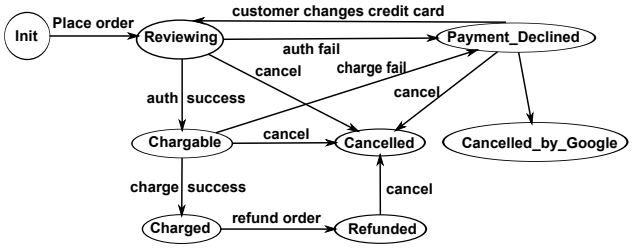


Figure 4: The automaton for Google financial status variable

V. Experiments

As WSDL is limited to describe the input and output of services, most WSDL services perform simple data look up tasks [4]. Semantic-rich services 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 composite obtained exhibits better parallelism for almost all non-trivial workflows.

A. 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 can handle simple lump sum payments, as well as complicated per item order processing operations such as credit authorization, declined payment, partial charge, back order, shipping, return, and refund. Because of this 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 describe their own order processing systems, and our composition engine handles the integration.

Many checkout APIs provide simple stateless calculation, which can be captured by I/O 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 model is not sufficient for their semantics. Table I is a part of the financial status summary table taken from the online reference, 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 III-A. The automaton translated from the P/E model is displayed in Fig. 4. This model construction can be automated if there are proper structure and syntax added to the documents.

To illustrate the 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. 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 [9], 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.

B. Oracle BPEL Workflow Samples

We apply the model extraction technique described in Section III-A to real BPEL workflows, and evaluate the benefit of our Petri net synthesis algorithm for better concurrency. The model extraction has to preserve the semantics of the workflow so the recomposed workflow produces the same result. We follow two rules for this purpose. First, each read of a variable in the workflow must see the same write. Second, invocations of an external service, called *partner links* in BPEL, must follow the same order. The second rule is conservative especially if the external service is stateless. We require this rule as we assume no knowledge about external services.

Following the two rules, we build an automaton for each variable and partner link in a BPEL workflow. The transitions of these automata are activities in the workflow. 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., `getVariableData('input'...)`. These activities also specify whether the access is a read or write. Structured activities define the ordering relationship of basic activities, which is captured by the automaton model. For example if two accesses to the same variable occur in a sequence and at least one of them is a write, we add these two activities as consecutive transitions in the automaton. If both of them are reads, it is safe to include all interleaving of the two activities in the automaton. Branches in the automaton capture `switch`, while `flow` is safely ignored as concurrent activities must access different objects.

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]. These samples are divided into categories including "demos", "references," "tutorials," 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 life-cycle 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` that contains value assignments to many different variables that can be arbitrarily interleaved, hence state explosion.

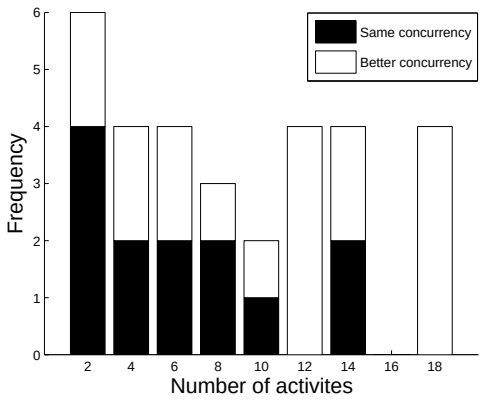


Figure 5: 31 Oracle BPEL examples in the "demos" category

Our Petri net synthesis algorithm successfully converted all 192 composite automata to Petri nets using only set regions. The linear programming function for vector region was never invoked. Due to the small size of these examples, the calculation takes less than a second for all cases. 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. The result is displayed as histogram in Fig. 5. There are 18 cases where our Petri nets are more concurrent, and the rest 13 are exactly the same. These 13 cases are mostly trivial examples with less than 10 activities organized as one sequence. For the other 18 cases, the most common source for better concurrency 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 in parallel. 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 Fig. 6. 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. Therefore it can take place in parallel with the preceding

```
<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 6: Code snippet of one Oracle BPEL example

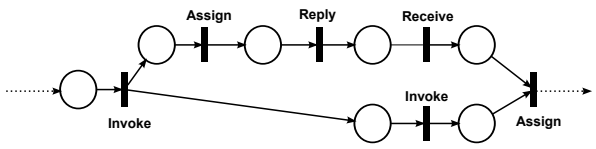


Figure 7: Petri net (partial) of the Oracle BPEL example

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 7 shows the part of the final synthesis result that corresponds to this code snippet.

VI. Conclusion

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.

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

[5] 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.

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

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

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

[9] 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.

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

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

[12] V. Diekert and G. Rozenberg. *The book of traces*. World Scientific Pub Co Inc, 1995.

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

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

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

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

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

[18] A. Riabov and et al. Wishful search: interactive composition of data mashups. In *WWW*, pages 775–784, 2008.

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

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

[21] 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.

[22] Y. Wang and et al. Finding the optimal representation for service composition using the theory of regions. Technical Report HPL-2010-191, HP Labs, Palo Alto, CA, 2010.

[23] 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.