Protocol-level Service Composition Mismatches: A Petri Net Siphon Based Solution

PengCheng Xiong[1], Calton Pu[1] and MengChu Zhou[2]
[1]College of Computing, Georgia Institute of Technology
[2]Department of ECE, New Jersey Institute of Technology
xiong@gatech.edu, calton@cc.gatech.edu, zhou@njit.edu

ABSTRACT:

Protocol-level mismatch is one of the most important problems in service composition. The state-of-the-art method to deal with protocol mismatch is to generate adaptors to check deadlock-freeness based on a reachability graph. When this property is violated, the generation process will repeat itself until no deadlock state is found. The main drawback of this method is that it does not take into account the future deadlock state and requires many interactions with a developer. Thus it suffers from low efficiency. In this paper, we model multiple web service interaction with a Petri net called Composition net (C-net). The protocol-level mismatch problem is transformed into the empty siphon problem of a C-net. By this transformation, we not only take future deadlock states into consideration but also find the optimal solution that involves fewest interactions with a developer. The proposed method is proved to achieve higher efficiency for resolving protocol-level mismatch issues than traditional ones.

KEY WORDS:
Service composition, protocol level mismatch, Petri nets

INTRODUCTION

In web service composition, when multiple web services are developed by different groups or vendors, they often fail to invoke each other because of mismatches. Service composition mismatches can be divided into interface and protocol-level ones (Nezhad, 2007). Table 1 gives a detailed taxonomy of already known service composition mismatches.

Table 1. Taxonomy of service composition mismatches

Service composition mismatches	Interface mismatches	message signature mismatches
		message split/merge mismatches
		message missing/extra mismatches
	Protocol level mismatches	unspecified reception
		mutual waiting mismatches
		non-local choice mismatches

Interface mismatches

Interface mismatches include message signature mismatches, message split/merge and message missing/extra mismatches (Benatallah, 2005; Benatallah, 2006) as shown in Fig. 1.

Message signature mismatches can be defined as: a service client's message has a different name and/or data type and/or data range from that of a service provider. For example, the message with the name "ClientOrder" on the service client's site may correspond to the message with the name "Order" on the provider's site. Note that, the case when the provider's message has a different data range from that of the client is also defined as parameter constraint mismatch in (Benatallah, 2005).

Message split mismatch can be defined as: a service client requires multiple messages to achieve certain functionality while a service provider can offer only a single message. For example, a provider's message *mp* contains *n* parts, i.e., mp_1, mp_2, …, and mp_n while a client considers mp_1, mp_2, …, and mp_n as *n* separate messages.

Message merge mismatch can be defined as: a service client requires a single message to achieve certain functionality while a service provider can offer multiple messages. This is the reverse case of message split mismatch. For example, a client requires message *mc* that is a combination of messages mc_1, mc_2, …, and mc_n from a provider.

Missing message mismatches can be defined as: a service client expects a message that a service provider does not issue. For example, a client expects an acknowledgement message while although a provider receives the previous message, it does not issue an acknowledgement. Note that, the missing message should not affect the semantics of the client. Extra message mismatches are defined as: a provider issues an extra message that a client does not expect. This is the opposite case of missing message mismatches. For example, a provider is sending an acknowledgement message that a client does not expect. Fig. 1 shows the cases for interface mismatches.

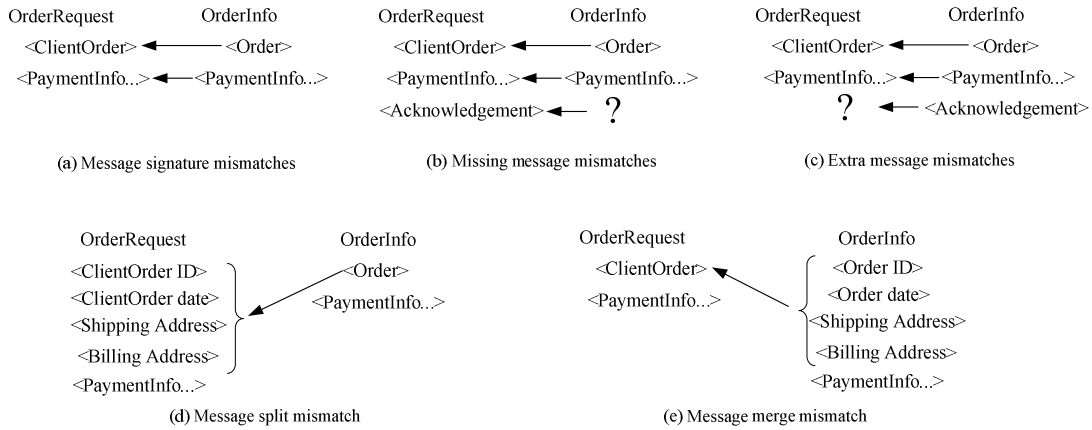


Figure 1. Illustration for interface mismatches

Services can be composed if there is no interface mismatches. There are significant research results towards service interface level mismatches such as service adaptation-based method (Benatallah, 2005; Benatallah, 2006), schema matching-based method (Nezhad, 2007), information retrieval techniques (Wang, 2003) and clustering-based approach (Dong, 2004). Service adaptation generates a service (the adaptor) that mediates the interactions among two services with interface mismatches such that two services are interoperable. For example, Nezhad *et al.* (2007) propose a service adaptor that is based on matching of schemas of messages to solve the service interface mismatch problem. They firstly obtain the XML schemas of all the messages on a service provider and a service client. Secondly, they calculate the similarity between all the possible message pairs considering message types and message orders. Finally, they pick the pair with the largest similarity. This method is proved to be useful to solve the message signature mismatches. To solve the other kinds of interface mismatches, Benatallah *et al.* (2005) propose a method based on a service adaptor. To solve message split mismatch, the adaptor extracts mp_1, mp_2, …, and mp_n from *mp* and then sends them to a service client. To avoid message merge mismatch, it integrates mc_1, mc_2, …, and mc_n to generate a single message *mc* and sends it to a service client. To avoid missing message mismatch, it generates a new message by using the template of the expected message and sends it to the service client. Finally, to avoid extra message mismatch, it simply intercepts the extra message and discards it.

Protocol level mismatches

Even if the service interface level matches perfectly, there may be protocol-level mismatches causing problems such as unspecified reception and deadlock (Yellin, 1997). Figure 2 shows the cases for unspecified reception, mutual waiting mismatches and non-local choice mismatches, respectively. Unspecified reception mismatches can be defined as: a service client's sending message order is different from a service provider's receiving message order. For example, as shown in Fig.2 (a), the client is sending message *a* first (denoted as *a*!), then message *b* (denoted as *b*!). However, the service provider is expecting to receive message *b* first (denoted as *b*?), then *a* (denoted as *a*?). Their interaction will result in an unspecified reception mismatches for the provider.

Deadlock mainly comes from mutual waiting mismatches (Nezhad, 2007) and non-local choice mismatches (Ben-Abdallah, 1997, Martens, 2003). Mutual waiting mismatches can be defined as: both of a service client and provider are expecting to receive a message from each other. For example, as shown in Fig.2 (b), after the client sends message *a* (denoted as *a*!), it is expecting to receive *c* (denoted as *c*?) from a provider. However, the provider is expecting to receive *b* (denoted as *b*?) from the client. Both of them are waiting for messages from each other.

Non-local choice mismatches can be defined as: a service client and provider make different local choices that should actually be synchronized. For example, as shown in Fig.2 (c), after the service client sends message *a* (denoted as *a*!), it makes a local choice, i.e., either to send a message *b* (denoted as *b*!) or send a message *c* (denoted as *c*!). However, the service provider also makes a local choice, i.e., after it receives message *a* (denoted as *a*?), it waits for message *b* (denoted as *b*?) or message *c* (denoted as *c*?). If the service client and provider happen to make the same choices, e.g., the client is sending *b* and the provider is expecting to receive *b*, then there is no problem. However, if the client is sending message *b* and the provider is expecting to receive *c*, then there is a non-local choice mismatch.

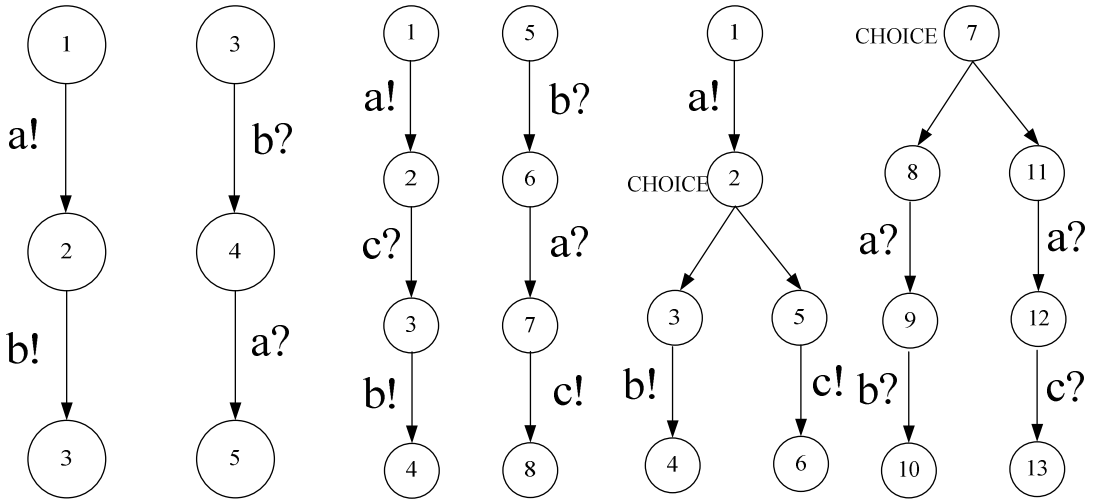


Figure 2. Illustration for protocol mismatches

To solve unspecified reception mismatches, Nezhad *et al.* (2007) propose a service adaptor-based method. The adaptor first buffers the pending message, e.g., message *a* in Fig. 2(a) and then send it to the provider when it is expected. Actually, some commercial products, e.g., the Oracle BPEL engine already have the function to automatically buffer the pending messages for

web service interactions. To solve deadlock mismatches, previously proposed methods (Nezhad 2007) mainly take the following steps: (1) check the deadlock using a state-space based method; and (2) provide some message and repeat testing the state-space until there is no deadlock. In other words, they offer quite limited help in resolving deadlock mismatches. Finding deadlocks involve intense interactions with developers. When protocols become complicated, it is very hard for developers to find the best solutions. For example, it is non-trivial (Nezhad 2007) to find the best solution for the mutual waiting mismatches outlined in Fig. 2(b).

Contributions of this work

The main contribution of the paper is a Petri net-based method to find protocol-level mismatches and then generate solutions to fix the mismatch problems. Our approach is based on an observation of Petri net objects called siphons (see Section 2 for details). A siphon is a subset of Petri net nodes with a property analogous to program safety properties. The number of tokens in a siphon never increases and an empty siphon always remains empty. Our observation is that protocol-level mismatch happens if and only if there is an empty siphon (see Theorems 1 and 2) in Petri net models derived from BPEL composition.

Technically, our approach consists of three steps. First, we adopt Business Process Execution Language for Web Services (BPEL) as the web service composition language. In the first step, the BPEL description of a composite service is translated into a Petri net model. Second, we use a mix-integer programming formulation to detect the maximal empty siphons, which are then used to find protocol-level mismatches. Third, we describe an algorithm to find siphon-based solutions for protocol-level mismatches by adding tokens in siphons to prevent them from becoming empty. Finally, we make time complexity comparisons between our approach and existing ones and prove that the proposed method can achieve higher efficiency for resolving protocol-level mismatching issues.

The rest of the paper is organized as follows. Section 2 shows basic Petri net concepts and how we model web service composition with Petri nets. In Section 3, we show how we detect and solve protocol-level mismatches. Section 4 provides two concrete examples for mutual waiting mismatches and non-local choice mismatches respectively to show how we use the siphon-based algorithms to solve the mismatches. Section 5 makes comparisons between the existing methods and the proposed one from two perspectives, i.e., the coverage of BPEL semantics and time efficiency of solutions. Finally, Section 6 concludes the paper.

MODELING SERVICE INTERACTION WITH PETRI NETS

In this section, we give basic Petri net concepts, simple sequential workflow net (SSN), workflow module net (WMN) and Composition net (C-net) and then show how we model service interaction with a C-net.

Basic Petri nets

Definition 1: A Petri net (Zhou, 1998; Aalst, 1998; Murata, 1989) is a 3-tuple, $N=(P, T, F)$ where:

i. $P=\{p_1, p_2,\ldots, p_m\}$, $m>0$, is a finite set of places pictured by circles;

ii. $T=\{t_1, t_2,\ldots, t_n\}$, $n>0$, is a finite set of transitions pictured by bars, with $P \cup T \neq \varnothing$ and $P \cap T = \varnothing$;

iii. $F \subseteq (P \times T) \cup (T \times P)$ is the incidence relation. Based on F, we can derive the input and output functions that are $m \times n$ matrices. $\boldsymbol{I}: P \times T \to \{0,1\}$ is an input function. $\boldsymbol{I}(p, t)=1$ if $(p,t) \in F$; and 0 otherwise. $\boldsymbol{O}: P \times T \to \{0,1\}$ is an output function. $\boldsymbol{O}(p, t)=1$ if $(t, p) \in F$; and 0 otherwise.

Postset of t is the set of output places of t, i.e., $t^{\bullet}=\{p|\boldsymbol{O}(p, t)\neq 0\}$. Preset of t is the set of input places of t, i.e., ${}^{\bullet}t=\{p|\ \boldsymbol{I}(p, t)\neq 0\}$. Post (Pre) set of p is the set of output (input) transitions of p, denoted by $p^{\bullet}$ and ${}^{\bullet}p$ respectively. $M: P \to Z^{+}$, is a marking where $M(p)$ represents the number of tokens in place p and $Z^{+}=\{0,1,2\ldots\}$. An initial marking is denoted by M_0. Tokens are pictured by dots. (N, M) is called a net system or marked net. p is marked by M iff $M(p)>0$. A place subset $S \subseteq P$ is marked by M iff at least one place in S is marked. The sum of tokens in S is denoted by $M(S)=\sum_{p\in S} M(p)$. A transition $t \in T$ is enabled under M, if and only if $\forall p \in {}^{\bullet}t: M(p)>0$ holds, denoted as $M[t>$. If $M[t>$ holds, t may fire, resulting in a new marking M', denoted as $M[t>M'$, with $M'(p)=M(p)-I(p,t)+O(p,t)$.

M' is reachable from M iff there exists a firing sequence $\sigma = t_{i1}t_{i2}\ldots t_{ik}$, such that $M[t_{i1}>M_1[t_{i2}>\ldots M_{n-1}[t_{ik}>M'$ holds. The set of markings reachable from M_0 in N is denoted as $R(N, M_0)$. Given a marked net (N, M_0) and $N=(P, T, F)$, a transition $t\in T$ is live under M_0 if $\forall M \in R(N, M_0)$, $\exists M' \in R(N, M)$, $\ni M'[t>$ holds. N is dead under M_0 if $\forall t \in T$, $M_0[t>$ cannot hold. (N, M_0) is live if $\forall t \in T$: t is live under M_0. A Petri net N is said to be deadlock-free if at least one transition is enabled at every reachable marking.

$N_X=(P_X, T_X, F_X)$ is a subnet of the Petri net $N=(P, T, F)$ iff $P_X \subseteq P$, $T_X \subseteq T$ and $F_X=F\cap((P_X\times T_X)\cup(P_X\times T_X))$. N_X is generated by P_X iff $T_X={}^{\bullet}P_X \cup P_X^{\bullet}$ (where the presets and postsets are taken w.r.t. F). Note that $\forall Q \subseteq P$, ${}^{\bullet}Q=\bigcup_{p\in Q} {}^{\bullet}p$ and $Q^{\bullet}=\bigcup_{p\in Q} p^{\bullet}$. A Petri net is called a state machine if $\forall t \in T$, $|{}^{\bullet}t|=|t^{\bullet}|=1$. An asymmetric choice net or AC net is a Petri net such that $\forall p_1, p_2 \subseteq P: p_1^{\bullet}\cap p_2^{\bullet}\neq\varnothing \Rightarrow p_1^{\bullet}\subseteq p_2^{\bullet}$ or $p_1^{\bullet}\supseteq p_2^{\bullet}$.

A P-vector is a column vector $X: P \to Z$ indexed by P and a T-vector is a column vector $Y: T \to Z$ indexed by T, where Z is the set of integers. The incidence matrix of N is a matrix $[N]=\boldsymbol{O}-\boldsymbol{I}$. X is a P-invariant (place invariant) if $X\neq 0$ and $X^T[N]=\mathbf{0}^T$ hold. Y is a T-invariant (transition invariant) if $Y\neq 0$ and $[N]Y=\mathbf{0}$ hold. $\|X\|=\{p\in P\mid X(p)\neq 0\}$ and $\|Y\|=\{t\in T\mid Y(t)\neq 0\}$ are called the support of X and Y, respectively.

A nonempty place set $S \subseteq P$ is called a siphon if ${}^{\bullet}S \subseteq S^{\bullet}$ holds as shown in Fig. 3. A siphon is minimal if there does not exist a siphon such that $S' \subset S$. A siphon has a property: a siphon free of tokens at a marking remains token-free. Marking M is a dead marking if no transition is enabled at M. A siphon S that eventually becomes empty is called potential deadlock. A Petri net is called potential deadlock free if all the siphons are always marked.

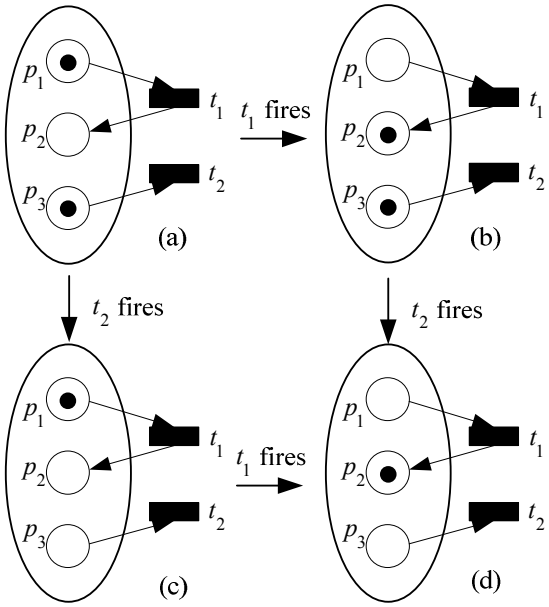


Figure 3. Illustration of a Petri net siphon

Definition 2: A simple sequential workflow net (SSN) is a Petri net $N = \{P \cup \{\alpha, \beta\}, T, F\}$ if:

i. *N* has two special places: α and β where α is a source place, i.e., ${}^\bullet\alpha = \varnothing$, and β is a sink place, i.e., $\beta^\bullet = \varnothing$;

ii. If we add a new transition *t* to *N* which connects α with β, i.e., ${}^\bullet t = \beta$ and $t^\bullet = \alpha$, then the resulting extended net $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$ where $\overline{P} = P$, $\overline{T} = T \cup \{t\}$, and $\overline{F} = F \cup \{(\beta, t), (t, \alpha)\}$, is a strongly connected state machine;

iii. Every circuit of *N* contains *t*; and

iv. $M_0(\alpha) = 1$ and $M_0(p) = 0$, $\forall p \neq \alpha$

State α is defined as $M(\alpha) = 1$ and $M(p) = 0$, $\forall p \in P \setminus \{\alpha\}$ while state β is defined as $M(\beta) = 1$ and $M(p) = 0$, $\forall p \in P \setminus \{\beta\}$. Note that SSN is a special kind of workflow net (Aalst, 1998) and is close to the definition of simple sequential processes in (Ezpeleta, 1995). SSN has a sound property:

i. For every state *M* reachable from state α, there exists a firing sequence leading from *M* to β;

ii. State β is the only state reachable from α with at least one token in β; and

iii. There are no dead transitions in it.

Definition 3: A workflow module net (WMN) is an extended Petri net $N=\{P \cup P_I \cup P_O, T, F\}$, where:

i. The subnet generated by *P* is an SSN;

ii. $M_0(\alpha) = 1$ and $M_0(p) = 0$, $\forall p \neq \alpha$;

iii. P_I and P_O denote the input and output interfaces for the workflow module respectively, satisfying

a) $P_I \neq \varnothing$, $P_O \neq \varnothing$, $P_I \cap P_O = \varnothing$ and $(P_I \cup P_O) \cap P = \varnothing$;

b) $\forall p \in P_I$, $p^{\bullet\bullet} \cap P \neq \varnothing$ and $|p^{\bullet}| = 1$;

c) $\forall p \in P_O$, ${}^{\bullet\bullet}p \cap P \neq \varnothing$ and $|{}^{\bullet}p| = 1$; and

d) $\forall p \in P_I \cup P_O$, ${}^{\bullet}p \cap p^{\bullet} = \varnothing$.

We define P as the set of process places and $P_I \cup P_O$ as the set of interface ones. The initial marking of a WMN follows that of its SSN according to ii of Definition 3.

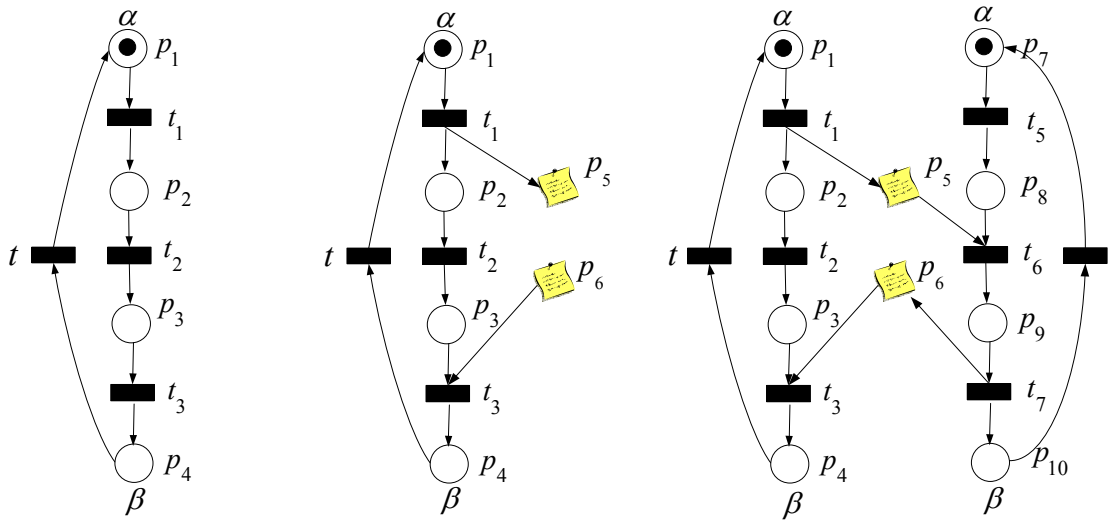
Definition 5: Two WMNs $N_j = \{P_j \cup P_{I_j} \cup P_{O_j}, T_j, F_j\}$, $j \in \{1,2\}$ are composable, if $P_1 \cap P_2 = \varnothing$, $T_1 \cap T_2 = \varnothing$ and $(P_{I_1} \cap P_{O_2}) \cup (P_{I_2} \cap P_{O_1}) = P_E \neq \varnothing$. For every $p \in P_E$, if $x \in {}^{\bullet}p$ and $y \in p^{\bullet}$, we call (x, p, y) an information channel. The 2-member Composition net (C-net) denoted as $N = N_1 \oplus N_2$ is defined as follows:

i. $P=P_1 \cup P_2$,

ii. $P_I = \{(P_{I_1} \cup P_{I_2}) \setminus P_E\}$, $P_O = \{(P_{O_1} \cup P_{O_2}) \setminus P_E\}$,

iii. $T=T_1 \cup T_2$, and

iv. $F=F_1 \cup F_2$.



(a) SSN (b) WMN (c) C-net

Figure 4. Illustration for Petri net siphon

Two WMN N_1 and N_2 are composable when they interact through a set of common places. For example, if $P_{I_2} \cap P_{O_1} = P' \neq \varnothing$, N_1 sends information through the set of interfaces P' that is received by N_2.

Definition 6: An n-member C-net denoted as $N = \oplus_{i=1}^{n} N_i$ is defined recursively following Definition 5. A C-net is a complete net iff $P_I = \varnothing$ and $P_O = \varnothing$.

For example, Figs. 4(a-c) show an SSN, WMN and C-net respectively. In an SSN, place α (i.e., p_1) and β(i.e., p_4) denote the start and end status, respectively and $p_{2\text{-}3}$ are internal places. A WMN is formed by adding two interface places and their corresponding arcs to an SSN. In Fig. 4(b), p_5 and p_6 are output and input places respectively. A WMN can be considered as a single-member C-net. Two or more WMNs can be composed to form a C-net. A complete C-net does not have additional input and output interfaces for external interaction. Fig. 4(c) shows a complete C-net composed of two WMNs. One of the WMN is shown in Fig. 4(b) which has an output and input place, which are denoted as p_5 and p_6 respectively. The other one has an input and output place, which are denoted as p_5 and p_6 respectively. Thus, they can be composed to form a complete C-net.

Model web service interaction with C-net

We then show how we model web service interaction with C-nets. We assume that no service interface level mismatches exist, i.e., the message signature and number of interfaces in both parties match. We divide the basic structures in BPEL, i.e., *receive, reply, invoke, assign, throw, terminate, wait*, *empty* and *link* into two categories. The first category is internal control logic that includes *assign, terminate, wait* and *empty*. The second category is external control logic that includes *receive, reply*, *invoke*, *throw* and *link*. Basic structures in the first category are not related to the interaction between different web services and we model them as internal places and transitions in a WMN. Basic structures in the second category are related to the interaction between different web services and we model them as transitions connected with internal and interface places as shown in Fig. 5. Note that, *invoke* is a combination of *reply* and *receive*, and *link* is modeled as an information channel.

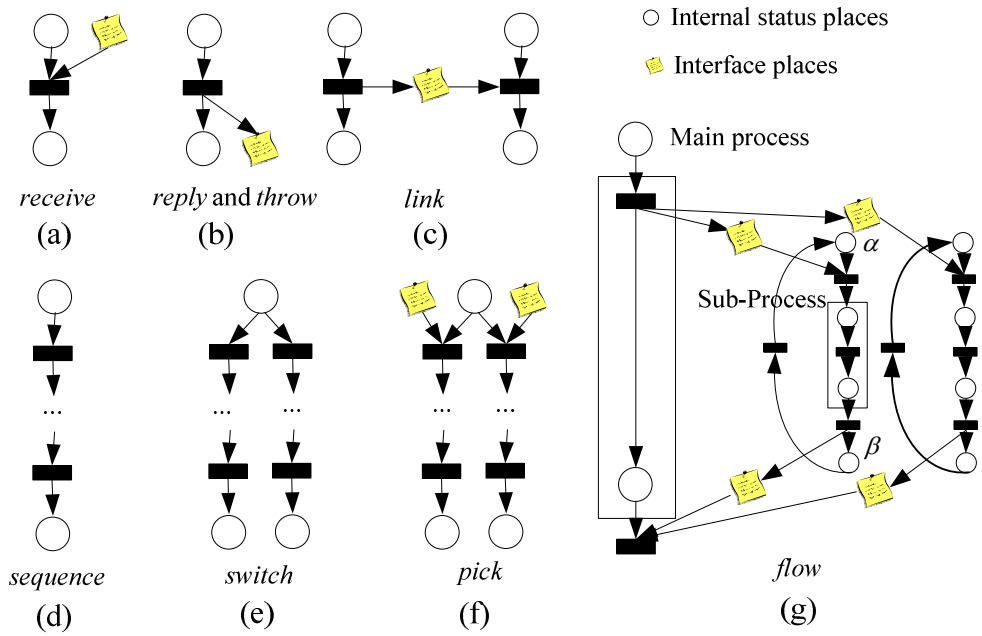


Figure 5. Transforming BPEL into WMNs

There are *sequence, flow, pick, switch* and *while* structured activities in a BPEL process. Based on basic structures, a WMN can cover the *sequence, pick* and *switch* structures. The semantics of *while* structure are similar to while-loop in programming languages like *Java*. Here we approximate the number of loops in a finite *while* structured activity and transform the activity to a *sequence* activity by expanding cycles (Ezpeleta, 1995). We can transform the processes that

are executed in parallel in the *flow* structure into the same processes that are invoked simultaneously in the *invoke* structure while maintaining the business logic. For example, we can divide the processes that are executed in parallel into separate BPEL processes while maintaining the business logic as shown in Fig. 5(g).

For example, assume that we have two web services as shown in Fig. 6, i.e., a customer service and an online shop service. The customer service first sends an order message, waits for delivery and then sends a payment message; while the online shop service waits for the order and payment, and then delivers the product.

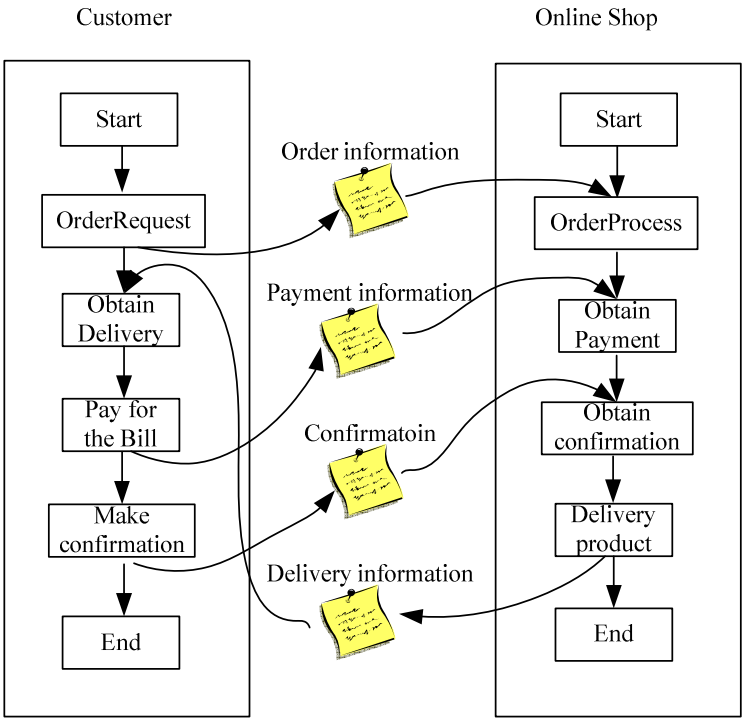


Figure 6. A customer service and an online shop service

We model the action of sending order information of a customer service in Fig. 7(a). Here we model the order information message as p_{11} and the customer service status before and after sending the message as p_1 and p_2, respectively. We also model the action of receiving order information of an online Shop service in Fig. 7(b). We model the order information message as the same p_{11} and the online shop service status before and after receiving the message as p_6 and p_7, respectively.

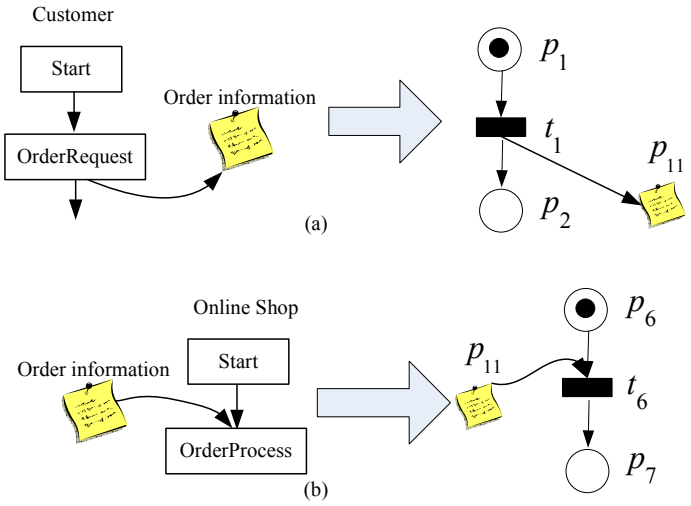


Figure 7. Modeling the case in Fig. 6 as C-net

Note that the interface places do not have tokens initially because no message is created. They can have tokens if and only if some transition wants to send a message through the information channel while they can lose a token if and only if some transition wants to receive a message through the information channel. A token in them models the situation when the required message is ready. We assume that the maximum number of tokens that an interface place can hold is one. Otherwise a BPEL process is not correct.

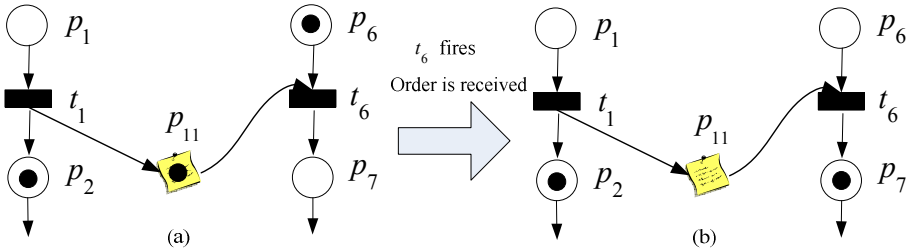


Figure 8. Modeling the interaction of web services

The firing of transitions in a C-net simulates the interaction of web services. For example, order information is ready in Fig. 8(a). After t_6 fires, the order information is received by the online shop service in Fig. 8(b). Following the modeling method, the final C-net for two web services is shown in Fig. 9(a).

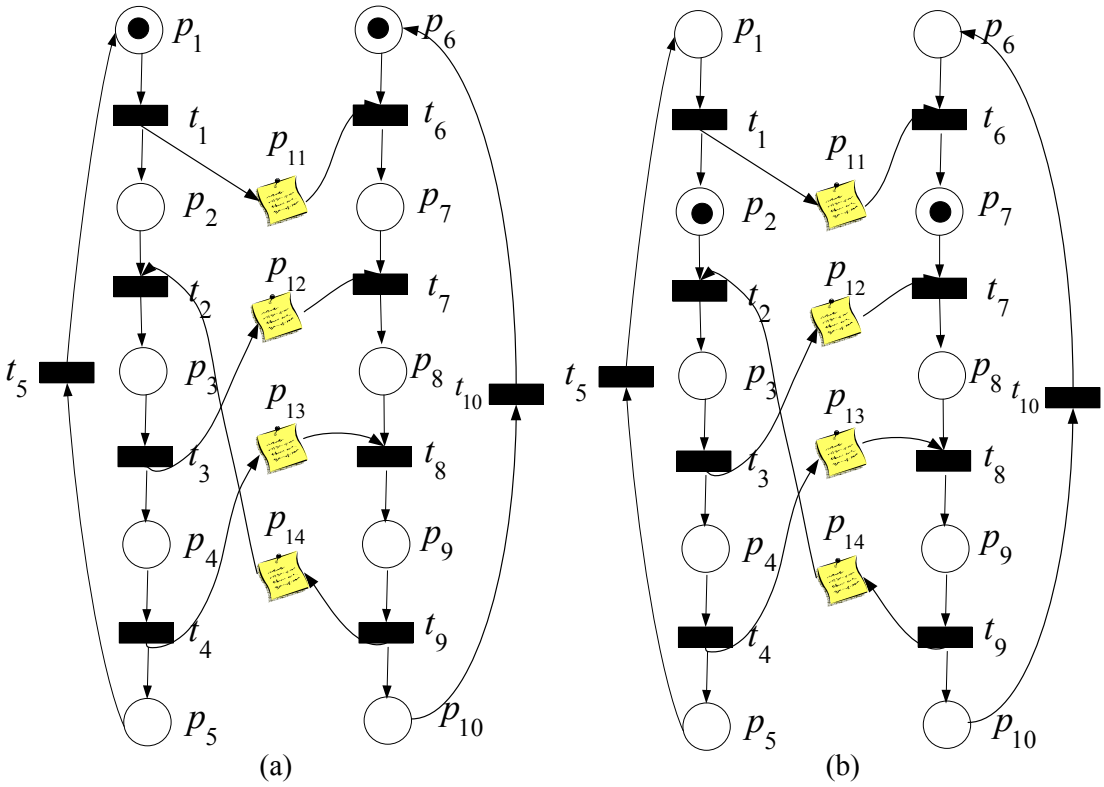


Figure 9. Modeling the interaction of web services in Fig.6

PROTOCOL-LEVEL MISMATCH ANALYSIS

As stated in Section 1, mismatches in protocol-level mainly come from mutual waiting mismatches and non-local choice mismatches. In this section, we propose a detection method, a core algorithm, as well as a comprehensive solution method for protocol-level mismatch.

Protocol-level mismatch detection

Definition 7: A C-net $N = \oplus_{i=1}^{n} N_i$ matches at protocol-level if N is live.

Note that, according to Theorem 4 in (Murata, 1989), a state machine N is live iff N is strongly connected and M_0 has least one token. Following this theorem, since an SSN is a strongly connected state machine with one token, it is live. However, this theorem is not true for C-nets.

Theorem 1: A C-net is live iff it is potential deadlock free.

Proof:

(1) A C-net is an asymmetric choice net.

$\forall p_1, p_2 \subseteq$ C-net and $p_1^{\bullet} \cap p_2^{\bullet} \neq \varnothing$, there are four cases.

Case 1: $p_1 \in P_E$ and $p_2 \in P_E$. According to Definition 3, we have $p_1^{\bullet} \cap p_2^{\bullet} = \{t\}$.

Case 2: $p_1 \in P_E$ and $p_2 \notin P_E$. We have $p_1^{\bullet} \subseteq p_2^{\bullet}$.

Case 3: $p_2 \in P_E$ and $p_1 \notin P_E$. We have $p_1^{\bullet} \supseteq p_2^{\bullet}$.

Case 4: $p_1 \notin P_E$ and $p_2 \notin P_E$. However, since the extended SSN is a strongly connected state machine, we have $\forall t \in T$, $|{}^{\bullet}t| = |t^{\bullet}| = 1$. This case is impossible.

For all the case, we have $p_1^{\bullet} \cap p_2^{\bullet} \neq \varnothing \Rightarrow p_1^{\bullet} \subseteq p_2^{\bullet}$ or $p_1^{\bullet} \supseteq p_2^{\bullet}$. Then a C-net is an asymmetric choice net.

(2) According to (Chu, 1997), an asymmetric choice net is live iff it is potential deadlock free.

Based on (1) and (2), a C-net is live iff it is potential deadlock free. □

According to the definition of potential deadlock, a siphon S eventually becomes empty when a C-net is not live and all the siphons are always marked when a C-net is live.

Theorem 2: A C-net $N = \oplus_{i=1}^{n} N_i$ matches at protocol-level iff all the siphons are always marked, i.e., $\forall M \in R(N, M_0)$, $\forall$ (minimal) siphon S, $M(S) \neq 0$.

Proof: Obvious from Definition 7 and Theorem 1. □

For instance, there are 5 minimum siphons in Fig. 6(a), i.e., S_1={$p_{1\text{-}5}$}, S_2={$p_{6\text{-}10}$}, S_3={p_1, $p_{3\text{-}5}$, $p_{7\text{-}9}$, p_{11}, p_{14}}, S_4={ p_3, $p_{8\text{-}9}$, p_{12}, p_{14}}, S_5={ $p_{3\text{-}4}$, p_9, $p_{13\text{-}14}$}. $M_0(S_{1\text{-}3})$=1 and $M_0(S_{4\text{-}5})$=0. Since there are initial empty siphons, i.e., S_4 and S_5, there exists protocol-level mismatching. This is true because after t_1 and t_6 fire, there is a deadlock as shown in Fig. 9(b).

Thus the problem of protocol-level mismatching of web service interaction is transformed to the problem of empty minimal siphons in a C-net. We can use the mix-integer programming algorithm to detect the maximal empty siphon (Chu, 1997).

In the aspect of detecting protocol-level mismatch, neither of the reachability analysis based method and mix-integer programming has clear computational advantage over the other because both of them have exponential complexity (Li, 2004).

Suppose that we have a C-net $N = \oplus_{i=1}^{n} N_i$ where some minimal siphons can become empty. Our main goal is to introduce into the system a solution to guarantee that no empty minimal siphons are reachable during the evolution of the new C-net, i.e., the new C-net at protocol-level matches.

Core algorithm

As stated in Theorem 2, a C-net $N = \oplus_{i=1}^{n} N_i$ matches at protocol-level iff $\forall M \in R(N, M_0)$, $\forall$ (minimal) siphon S, $M(S) \neq 0$. According to the property of a siphon, the number of tokens in a siphon will never increase and an empty siphon will always remain empty.

From a modeling perspective, the most intuitive way is to add tokens to the C-net in order to make one or more tokens to interface places to make all of the empty siphons marked. For example, adding one token to p_{11} in Fig. 9 means that the order information is ready while adding to p_{12} means that the payment information is ready. From a service perspective, the customer is waiting for the delivery information while the online shop is waiting for the payment information. Thus, developers must be involved to provide additional information at the deadlock point, i.e., to ask the customer to provide the payment information or to ask the online shop to provide the delivery information.

In the above cases, there are at least 4 choices for a developer:

Choice 1(C1): provide nothing

Choice 2(C2): provide the payment information
Choice 3(C3): provide the delivery information
Choice 4(C4): provide the payment and delivery information

But developers may not always make a good decision. A good decision implies correctness and efficiency of the solution.

Correctness means, the solution should resolve the current protocol level mismatch. C1 is obviously not a correct solution since both the customer and online shop web services are still waiting. C2 is also not a correct solution although it can solve the current waiting status for the online shop. After choosing C2, payment information is provided and t_7 can fire. However, the system goes to another mutual waiting state, i.e., the online shop is waiting for confirmation and the customer is waiting for delivery. This is because siphon S_5 is empty.

Efficiency means, the solution should require the smallest amount of information to solve as many as future protocol-level mismatches. C4 is also not an efficient solution because it needs 2 kinds of messages. For example, payment, confirmation and delivery information is three different kinds of messages. Choices 1-4 require 0,1,1 and 2 kinds of messages respectively. Then although C3 and C4 are all correct solutions, C3 is more efficient than C4 because it requires fewer kinds of messages.

We propose an algorithm to choose the correct and also the most efficient solution by linear programming.

Core Algorithm:

INPUT: (1) n-member C-net N with minimum siphon set $\Omega=\Omega_N \cup \Omega_Y$. Ω_N denotes the non-empty siphon set, and $\Omega_Y = \{S_1, S_2, \ldots S_i\}$ denotes the empty siphon set when a protocol mismatch happens. (2) Interface place set $P_E=\{p_1, p_2, \ldots p_j\}$

OUTPUT: A list of messages that should be provided. We denote the list as a $j \times 1$ vector L where $L(j)=1$ if $p_j \in L$; and 0 otherwise.

BEGIN:

Step 1. /* Calculate the contribution matrix of every message to the siphon*/

Constitute an $i \times j$ matrix A, where $A(i, j)=1$ if $p_j \in S_i$; and 0 otherwise.

Step 2. /* Optimization*/

Compute the following linear programming problem:

Minimize $\mathbf{1}*L$

s.t. $A*L=\mathbf{1}^{\mathrm{T}}$

Step 3. /* Return result*/

Return L

END

The time complexity for the algorithm is linear. We explain the idea underlying this algorithm as follows:

Firstly, the constraint function $A*L=\mathbf{1}^{\mathrm{T}}$ can return the correct solutions. As shown in Fig. 10, because we have $A(i, j)=1$ if $p_j \in S_i$; and 0 otherwise, and $L(j)=1$ if $p_j \in L$; and 0 otherwise, the solutions of the constraint function guarantee that each empty siphon is marked by exactly one token. Moreover, if the constraint function is not satisfied, there is at least one empty siphon.

Secondly, the objective function of the linear programming formulation can return the best solution. In the contribution matrix, the more siphons the message p_j is involved in, the more **1**'s it has in the jth column. Since the objective function calculates the sum of messages, the solution has the smallest total number of messages if the proposed objective function is minimized.

Finally, we can claim that, through the algorithm, we can provide a correct and also efficient solution to protocol-level mismatch.

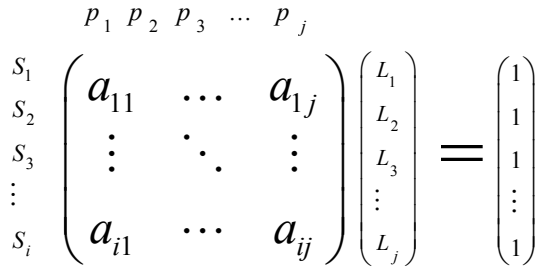


Figure 10. Illustration of linear programming

Comprehensive solution to protocol-level mismatch

Based on the core algorithm, we propose a comprehensive solution to protocol-level mismatch as follows. The input of the solution is web services and their interaction described in BPEL. We assume that there is no interface mismatches for these web services' interaction.

Comprehensive solution:
BEGIN:
Step 1. Transform web service interaction into Petri net models as simple sequential workflow net (SSN), workflow module net (WMN) and Composition net (C-net).
Step 2. Check if there is/will be empty siphon by using mix-integer programming according to "protocol-level mismatch detection". If none of them can become empty, go to end.
Step 3. Find the set of minimal siphons. Monitor all the minimal siphons until all the web services terminate properly, i.e., for each SSN, state β is reached from state α. During the monitoring period, if one of the minimal siphons becomes empty, denote the empty siphon set as Ω_Y. Go to step 4. When monitoring period ends, go to end.
Step 4. Take Ω_Y as the input of core algorithm. Run the core algorithm. Provide the list of messages that are returned by the algorithm. Go back to step 3.
END

We explain the idea underlying the solution as follows:

Firstly, according to the taxonomy of service composition mismatches, we may have both the interface and protocol mismatches. Since interface matches are a prerequisite to achieve protocol matches, before using the solution, we have to guarantee that there is no interface mismatches.

Secondly, we model web service interaction by Petri net models so that we can identify and check the minimal siphons. If we cannot detect any empty minimal siphon, then there is no protocol mismatches and we end. Otherwise, we have to monitor those minimal siphons until all the web services terminate properly. Before they terminate, when there is an empty siphon, we can use the core algorithm to obtain the list of messages that should be provided.

Finally, we provide those messages and solve the protocol mismatches.

We make an overview of the solution. As mentioned previously, there are three kinds of protocol level mismatches, i.e., unspecified reception, mutual waiting mismatches and non-local choice mismatches. The cause for unspecified reception is a wrong sequence of sending messages and it can be solved by buffering previous messages. The cause for mutual waiting mismatches is that the web services follow a conflict "send and receive" sequence and they are waiting for each other to send messages. The cause for non-local choice mismatches is that the web services follow a conflict "send and receive" choice and wait for each other to send messages. In both of the cases, the missing messages are responsible for the mismatch. By using our core algorithm,

we are not only trying to find those missing messages but also find the smallest set of those missing ones. Thus, after we provide those messages, we are able to resolve the protocol mismatches.

We outline a complexity analysis of the solution step by step.

Step 1 is doing transformation tasks. Thus its complexity linearly depends on the size of BPEL.

Step 2 is answering a "Yes or No" question, i.e., whether or not there is an empty siphon at any reachable markings. Then its complexity depends on the time when the algorithm will find an empty siphon. In one case, if there is a mutual waiting mismatch, generally speaking, there is at least one empty siphon at the initial marking. For example, there is a mutual waiting mismatch in Fig.6 and there is an empty siphons at the initial marking, i.e., S_4={ p_3, $p_{8\text{-}9}$, p_{12}, p_{14}}. Using the method to detect empty siphon at the initial marking, the maximal empty siphon that contains S_4 can be detected in polynomial time (Chu, 1997). In another case, if there is a non-local choice mismatch, generally speaking, there is no empty siphon at the initial marking but there will be an empty siphon at a future marking. This is because a non-local choice mismatch will happen only when services make conflicting choices. In other words, it may be "hidden" when services happen to make the same choices. To detect an empty siphon for a non-local choice mismatch at a future marking has exponential complexity in the worse case. Fortunately, since we already propose an approach to resolve the mismatch, the answer to whether or not there is an empty siphon at any reachable markings is not very important.

Step 3 is generating the minimal siphons. It is a prerequisite for Step 4. The generation of minimal siphons has exponential complexity in the worse case. Fortunately, since siphons are a kind of special structure of Petri nets, they are not related with the states and we can generate them offline.

Step 4 is trying to control the empty siphons. Because it is using linear programming, the complexity is polynomial.

EXAMPLES

As previously mentioned, protocol mismatch includes mutual waiting mismatch and non-local choice mismatches. In this section, we use two concrete examples to illustrate how to use the method mentioned above, especially the core algorithm.

The scenario in Fig. 6 can be classified as mutual waiting mismatch. After t_1 and t_6 fire, there is a deadlock. We have Ω_N={$S_{1\text{-}3}$}, Ω_Y ={$S_{4\text{-}5}$} where S_4={ p_3, $p_{8\text{-}9}$, p_{12}, p_{14}}, S_5={ $p_{3\text{-}4}$, p_9, $p_{13\text{-}14}$}, and P_E={$p_{11\text{-}14}$}. Because $p_{12}\in S_4$, $p_{14}\in S_4$, $p_{13}\in S_5$, $p_{14}\in S_5$, we have A=((0 1 0 1) (0 0 1 1)). The result is L=(0 0 0 1)^T. It means that a developer should ask the online shop to provide the delivery information (the token in the interface place p_{14} denotes the delivery information). Moreover, if we check the method that the developer asks the customer to provide the payment information, i.e., L'=(0 1 0 0)^T, we find that this method will fail. This is simply because L' is not a solution of the linear programming problem. Because $A*L'$=(1 0)^T, although S_4 is marked, S_5 is still empty.

We use a scenario mentioned in (Ben-Abdallah, 1997; Martens 2003) as shown in Fig. 11 to illustrate the non-local choice mismatches. There are three web services, denoted as WS1-WS3. We first denote their interaction through Message Sequence Charts (MSC). WS1 first sends "Dreq" to WS2, and then WS2 sends "DA" to WS3. After WS1 first sends "Dreq" to WS2, it is the first web service to decide whether to MSC2 or MSC3. For example, if it decides to proceed by following MSC3, it will send "Creq" to WS2, WS2 sends "RC" to WS3 and then WS1 waits for WS3 to send back "Crep".

We model the messages "Dreq", "DA", "Creq", "RC", "Crep", "DC", "Dind" and "Drep" with p_{17}, p_{20}, p_{18}, p_{22}, p_{23}, p_{21}, p_{19} and p_{24} respectively. Then we build the C-net for the scenario as shown in Fig.12(a). There will be no non-local choice mismatches if WS1 and WS3 can make the same choices, i.e., following MSC2 or MSC3. However, if they follow different MSCs, then there is a non-local choice mismatch as shown in Fig.12(b). For example, if WS1 decides to proceed following MSC3 and WS3 decides to proceed following MSC2, then after t_1, t_3, t_7, t_8, t_{10}, t_{13}, t_{14}, t_{11}, and t_{18} fire, there will be a deadlock. The situation will be: WS1 is waiting for "Crep" to fire t_5; WS2 is waiting to fire t_9 because "DC" arrives; WS3 is waiting for "Drep" to fire t_{16}.

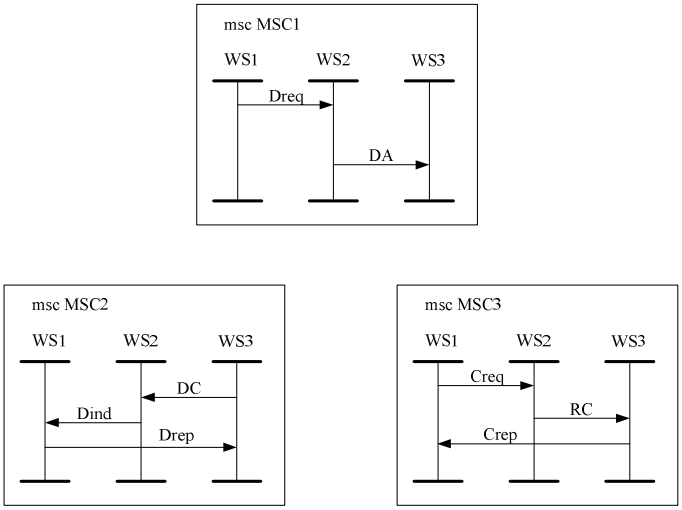


Figure 11. Non-local choice protocol mismatch

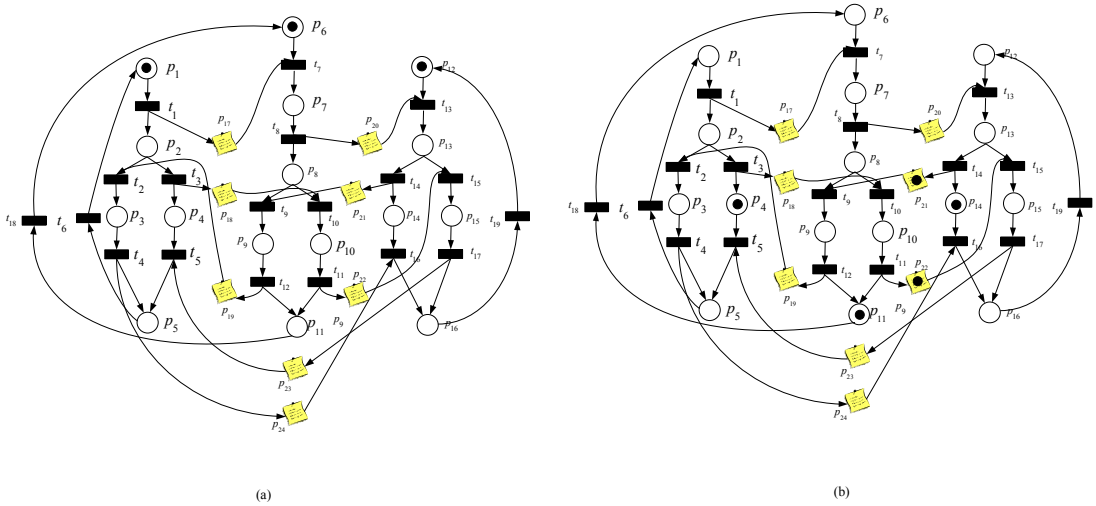


Figure 12. The C-nets for the non-local choice protocol mismatch in Fig. 11

The three web services are waiting for each other. There are 15 minimal siphons in the C-nets. We have $\Omega_N=\{S_{1\text{-}11}\}$, $\Omega_Y=\{S_{12\text{-}15}\}$, and $P_E=\{p_{17\text{-}24}\}$. For simplicity, we only list the siphons in Ω_Y, i.e., $S_{12}=\{p_1, p_3, p_5, p_{7\text{-}9}, p_{13}, p_{15}, p_{17}, p_{19\text{-}20}, p_{23}\}$, $S_{13}=\{p_1, p_3, p_5, p_{7\text{-}9}, p_{12\text{-}13}, p_{15\text{-}17}, p_{19}, p_{23\text{-}24}\}$, $S_{14}=\{p_{1\text{-}3}, p_5, p_7, p_{13}, p_{15}, p_{17}, p_{20}, p_{23}\}$, $S_{15}=\{p_{1\text{-}3}, p_5, p_{12\text{-}13}, p_{15\text{-}16}, p_{23\text{-}24}\}$. The result is $L=(0\ 0\ 0\ 0\ 0\ 0\ 1\ 0)^T$. Here p_{23} denotes "Crep". It means that the developer should ask WS3 to provide "Crep".

RELATED WORK

Web Services Business Process Execution Language (BPEL) (Curbera, 2006; Wohed, 2003) is an OASIS standard executable language for specifying interactions with web services. It is becoming the de facto standard for implementing business processes using web service technology. However, it lacks a formal semantics, which makes the web service composition mismatches hard to analyze. In order to avoid this disadvantage, several attempts have been made to translate BPEL into formal models. By using them, it is able to detect inconsistencies in the BPEL specification and formally analyze BPEL processes using techniques of computer aided verification.

For example, Fisteus *et al.* (2004) use finite state machines, Ferrara (2004) uses process algebra Fahland and Reisig (2005) use abstract state machines, and Ouyang *et al.* (2005), Hinz *et al.* (2005) and Lohmann *et al.* (2007) use Petri nets. Other methods, like pi-calculus (Lucchia, 2007), message sequence charts (Foster, 2005) and conversations (Fu, 2004) are also used.

Because Petri nets are appropriate for both modeling and analyzing BPEL, in this article, we propose our own semantics for BPEL translation to Petri nets, e.g., SSN, WMN and C-net and our own analysis methods based on siphons. We focus on the other Petri net-based methods and make comparisons among them.

Coverage of Petri net semantics

Different researchers propose different Petri net semantics for BPEL. Ouyang *et al*. (2005) transform BPEL into Petri nets represented in the Petri Nets Markup Language (PNML) by BPEL2PNML. They model a BPEL process as a work flow net (WFN) (Aalst 1998). Lohmann adopts open workflow nets (oWFNs) (Lohmann 2006) for modeling BPEL processes and developed a compiler BPEL2oWFN. Both BPEL2PNML and BPEL2oWFN implement reduction techniques to reduce the model after their generation. Our semantics cover the main components of BPEL process, e.g., the basic structures and structured activities instead of covering all the elements in BPEL. For example, we model the control flow by undistinguishable black tokens and we provide little data information (e.g., data values and message content). In this case, C-net is a kind of low-level Petri nets and it is more like a WFN than an oWFNs. The most distinguished modeling advantage for the C-net is that it enables the useful conclusions for analyzing protocol mismatches.

Efficiency of discovering and solving protocol mismatches

After transforming BPEL into Petri nets represented in PNML, Ouyang *et al.* (2005) use WofBPEL to support three types of analysis, i.e., reachability analysis, competing message-consuming activities and garbage collection of queued messages by generating the full state space. Similarly, König *et al.* (2008) uses Fiona to automatically analyze the interactional behavior of a given oWFN. Martens (2003) proposes a BPEL annotated Petri nets (BPN) and presents a decision algorithm for the controllability of a BPN model based on the communication graph (c-graph). The examination of interactions between the composed BPEL processes is transformed into the verification of deadlock-freeness of a BPN. Nezhad *et al*. (2007) generate a mismatch tree to handle deadlock situations. The basis of such tree is similar to the reachability graph in Petri nets. Although the current methods provide useful insights into the problem by adopting Petri net based analysis methods, e.g., WofBPEL, Fiona, c-graph and mismatch tree, their analysis is mainly based on a reachable state space and they do not propose an effective solution to resolve the protocol-level mismatch issues.

For example, following the mismatch tree approach that is proposed by Nezhad *et al*., we have choices 1-4 for the scenarios in Fig. 4. For each choice, we have to search all the reachable state space and find out if it results in a future deadlock. Note that, the complexity for generating the reachable state space is exponential. Normally, assume that there are w web services and each of them is waiting for a message, the number of solutions to consider should be $O(2^w)$. Assume

that there are k nodes in a Petri net. Then the complexity for generating the reachable state space is $O(Ce^k)$ for each solution. Thus the total complexity to find the best solution would be $O(C2^w e^k)$.

Compared with their work, the proposed one is more efficient. For example, to solve the same problem in Fig. 4, after generating the siphons, it simply uses the linear programming to find the best solution. Assume that there are k nodes in the Petri net. Then the complexity for generating the minimal siphons is exponential, i.e., $O(Ce^k)$. Because the complexity to solve linear programming is polynomial-time, the total complexity is still $O(Ce^k)$.

Moreover, because siphons depend on only the structure of Petri nets, they can be generated off-line. Thus, the generation process can run in parallel with web services' interaction. That is to say, we can generate the minimal siphons before the protocol mismatch appears and we can run the linear programming right after the protocol mismatch appears. However, the reachable state space method is strongly based on the input status. Thus, only when the protocol mismatch really happens, we can obtain the input status and then we can test the candidate solutions in reachable state space method. A detailed comparison is shown in Fig. 13.

Our previous work (Xiong, 2010) also proposes a method to solve non-local choice mismatches. There are two major differences between this work and (Xiong, 2010). (1) The protocol-mismatches that this work can solve include both mutual-waiting and non-local choice mismatches. Our previous work (Xiong, 2010) can only solve non-local choice mismatches. Compared with previous work, this work can solve more general protocol-mismatches. (2) Although both of the works can solve non-local choice mismatches, they are using different methods. The previous work uses a method based on adding information links and needs to modify BPEL source code. It does not allow non-local choice mismatches to happen. Compared with it, this work proposes a method based on providing missing messages. It can allow non-local choice mismatches to happen and then it provides those messages to resolve protocol-mismatches.

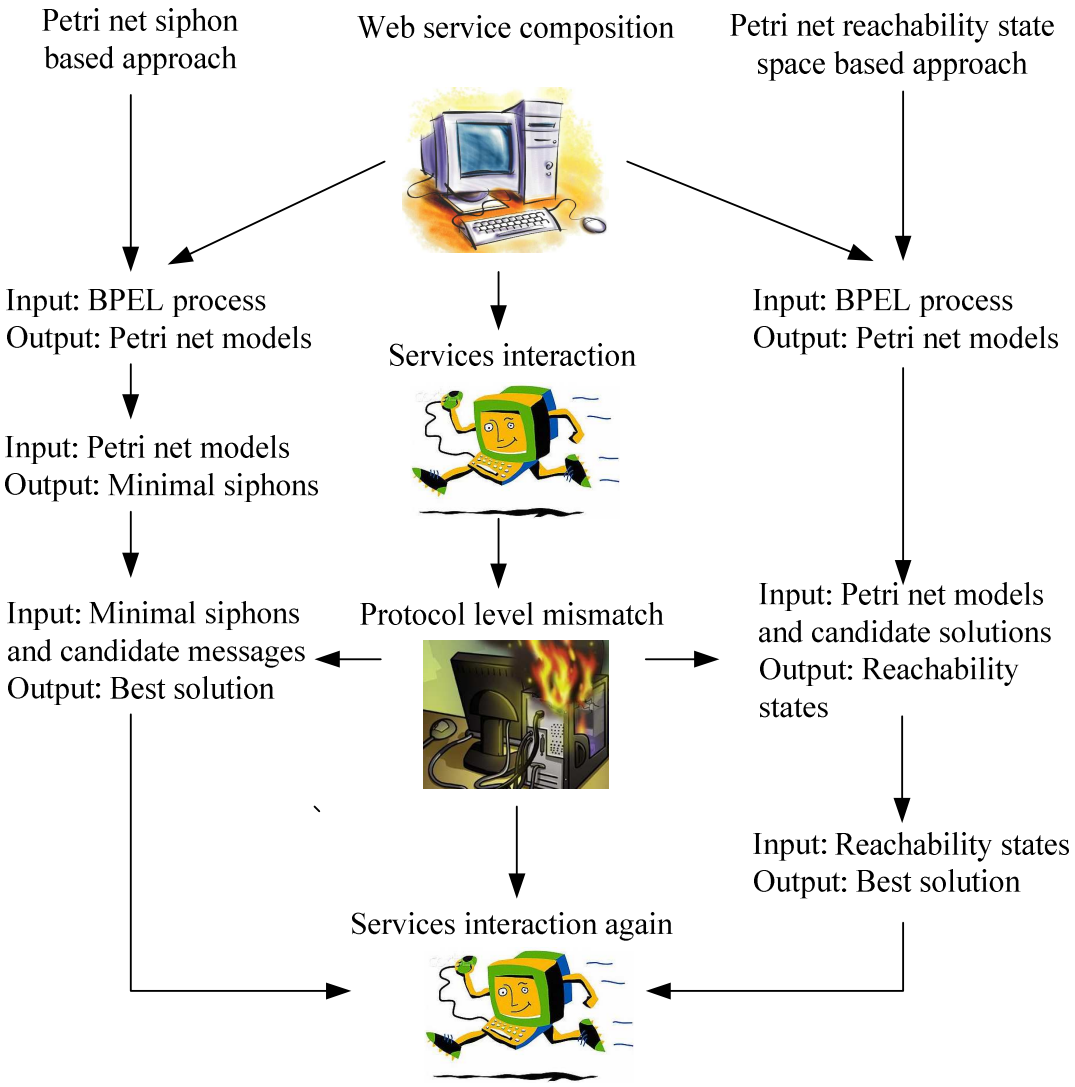


Figure 13. The comparison between Petri net siphon based and reachability state space based methods

CONCLUSION

In service composition, services interact with each other through a sequence of messages to achieve a certain business goal. However, mismatches at the interface and protocol levels may render the composite service unusable. There are a plethora of works on service interface level mismatches and most of the interface level mismatch can be solved by using adaptors. However, the mismatches at protocol levels are not well addressed. Existing studies based on adaptors and reachability analyses fails to provide an efficient solution.

The main contribution of this paper is to propose a Petri net siphon-based solution to protocol level mismatches. The whole approach follows the steps of modeling, analysis and solution. Firstly, the web service composition is modeled by a set of Petri nets, i.e., SSN, WMN, and C-net. Secondly, the protocol level mismatch problem is analyzed by Petri net siphons. We make a useful conclusion that, there should be a protocol level mismatch if and only if there is an empty

siphon in the C-net. Finally, based on this conclusion, we use a linear programming method to find the best solution. Our approach provides an optimized and also automatic solution for correcting protocol-mismatches. This approach greatly reduces the amount of interactions with developers.

In the future, first, we would like to extend our approach to address the protocol discovery from message logs. Our current approach is based on the already known BPEL. We would like to extend our approach to deal with the situation when BPEL is unknown and conversations between two or more services should be discovered by message logs. Data mining and machine learning technique may help to accomplish this goal (Aalst 2008). Second, we would like to extend our approach to address the dynamically evolving service protocols. As mentioned in (Ryu, 2007; Ryu, 2008), service protocols can dynamically adapt to changes in business needs and/or regulations. For example, Ryu *et al.* (2008) propose a comprehensive approach to dynamic protocol evolution management using a finite state machine model. In this paper, we only target at static service protocols. We plan to adapt our approach to deal with dynamic protocol evolution and find out how changes in protocol can impact the analysis and solution. Finally, although the search for siphons can be performed offline and the computation of minimum siphons is simple, in some complex structured C-net, such computation can be expensive. Some polynomial complex algorithms to find and control siphons should be explored for C-nets by making full use of their special structural information. Some recent advance (Wang, 2009; Li, 2009; Hu, 2009; Wu, 2008) may provide good help along this direction.

ACKNOWLEDGMENT

This research has been partially funded by National Science Foundation grants ENG/EEC-0335622, CISE/CNS-0646430, CISE/CNS-0716484, AFOSR grant FA9550-06-1-0201, NIH grant U54 RR 024380-01, IBM, Hewlett-Packard, Wipro Technologies, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

REFERENCES

Nezhad H., Benatallah B., Martens A., Curbera F. and Casati F. (2007), "Semi-Automated Adaptation of Service Interactions", Proc. of the 16th Intern. World Wide Web Conf., Banff, Alberta, Canada, pp. 993-1002.

Benatallah B., Casati F., Grigori D., Nezhad H. and Toumani F. (2005), "Developing Adaptors for Web services Integration", Proc. of CAiSE, Porto, Portugal, pp.415-429.

Benatallah B., Casati F., and Toumani F. (2006), "Representing, analysing and managing web service protocols", Data Knowl. Eng., 58(3):327–357.

Wang Y. and Stroulia E. (2003), "Flexible interface matching for web-service discovery," Proc. of the 4th Intern. Conf. on Web Info. Sys. Eng., Rome, Italy, pp. 147-156.

Dong X., Halevy A., Madhavan J., Nemes E. and Zhang J. (2004), "Similarity Search for Web Services", Proc. of the 30th Intern. Conf. on Very Large Data Bases, Toronto, Canada, pp. 372-383.

Yellin D. and Strom R. (1997), "Protocol specifications and component adaptors", ACM Trans. on Prog. Lang. and Sys., Vol. 19, Iss. 2, pp. 292-333.

Zhou M. and Venkatesh K. (1998), Modeling, Simulation and Control of Flexible Manufacturing Systems: A Petri Net Approach, World Scientific, Singapore.

Aalst W.M.P. (1998), "The application of Petri Nets to workflow management", The Journal of Circuits, Systems and Computers, Vol. 8, No. 1, pp. 21-66.

Murata T. (1989), "Petri nets: Properties, analysis and applications", Proc. of IEEE, 77(4):541–580

Ezpeleta J., Colom J. and Martinez J. (1995), "A Petri Net based Deadlock Prevention Policy for Flexible Manufacturing Systems," IEEE Trans. on Robotics and Automation, Vol. 11, No. 2, pp. 173–184.

Xiong P.C., Fan Y.S. and Zhou M.C. (2010), "A Petri Net Approach to Analysis and Composition of Web Services", IEEE Trans. on Sys., Man and Cybern., Part A, 2009. Vol. 40, Iss. 2, pp. 376–387.

Chu F. and Xie X.L. (1997), "Deadlock analysis of Petri nets using siphons and mathematical programming," IEEE Trans. on Robotics and Auto., vol. 13, no. 6, pp. 793–804.

Li Z.W.and Zhou M.C. (2004), "Elementary Siphons of Petri Nets and Their Applications to Deadlock Prevention in Flexible Manufacturing Systems," IEEE Trans. on Sys., Man and Cybern., Part A, Vol. 34, No. 1, pp. 38-51.

Ben-Abdallah H. and Leue S. (1997), "Syntactic detection of process divergence and non-local choice in message sequence charts", Proc. of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Enschede, Netherlands, pp. 259-274.

Martens A. (2003), "Usability of Web services", Proc. of the 4th Intern. Conf. on Web Info. Systems Eng. Workshops, Rome, Italy, Dec. 2003, pp. 182-190.

Curbera F., Mukhi N., Nagy W. and Weerawarana S. (2006), "Implementing BPEL4WS: the Architecture of a BPEL4WS Implementation", Concurrency and Computation: Practice & Experience, Vol. 18 , Iss. 10, pp 1219-1228.

Wohed P., Aalst W.M.P., Dumas M. and Hofstede A. (2003), "Analysis of Web services composition languages: The case of BPEL4WS", Proc. of 22nd International Conference on Conceptual Modeling, Chicago, IL, pp. 200-215.

Fisteus J., Fernández L. and Kloos C. (2004), "Formal verification of BPEL4WS business collaborations", Proc. of the 5th International Conference on Electronic Commerce and Web Technologies, Zaragoza, Spain, pp. 76-85

Ferrara A. (2004), "Web Services: a Process Algebra Approach", Proc. of the 2nd International Conference on Service Oriented Computing, New York City, NY, USA, pp. 242-251.

Fahland D. and Reisig W. (2005), "ASM-based semantics for BPEL: The negative control flow", Proc. of the 12th International Workshop on Abstract State Machines, Paris, France, pp. 131-151.

Ouyang C., Verbeek E., Aalst W.M.P., Breutel S., Dumas M. and A. Hofstede (2005), "WofBPEL: A Tool for Automated Analysis of BPEL Processes", Proc. of the 3rd Intern. Conf. on Service Oriented Computing, Amsterdam, Netherlands, pp. 484–489.

Hinz S., Schmidt K. and Stahl C. (2005), "Transforming BPEL to Petri Nets", Proc. of the 3rd International Conference on Business Process Management, Nancy, France, pp. 220-235.

Lohmann N. (2007), "A feature-complete Petri net semantics for WS-BPEL 2.0", Proc. of the 4th International Workshop on Web Services and Formal Methods, Brisbane, Australia, pp. 77-91.

Lucchia R. and Mazzara M. (2007), "A pi-calculus based semantics for WS-BPEL", Journal of Logic and Algebraic Programming, Vol. 70, Iss. 1, pp. 96-118

Foster H., Uchitel S, Magee J. and Kramer J. (2005), "Tool support for model-based engineering of Web service compositions", Proc. of the 2005 IEEE International Conference on Web Services, Orlando, Florida, USA, Vol. 1, pp. 95-102.

Fu X., Bultan T. and Su J. (2004), "Analysis of Interacting BPEL Web Services", Proc. of the 13th International World Wide Web Conference, New York, NY, pp. 621-630.

Lohmann N., Massuthe P., Stahl C., and Weinberg D. (2006), "Analyzing Interacting BPEL Processes", Proc. of the 4th International Conference on Business Process Management, Vienna, Austria, pp. 17–32.

König D., Lohmann N., Moser S., Stahl C. and Wolf K. (2008), "Extending the Compatibility Notion for Abstract WS-BPEL Processes", Proc. of the 17th International World Wide Web Conference, Beijing, China, pp. 785-794.
Martens A., Hamadi R. and Benatallah B. (2003), "A Petri Net based Model for Web Service Composition", Proc. of the 14th Australian Database Conf., Adelaide, Australia, pp. 191-200.
Aalst W.M.P. and Verbeek H. (2008), "Process Mining in Web Services: The WebSphere Case", IEEE Data Eng. Bull. 31(3): 45-48.
Ryu S., Casati F., Skogsrud H., Benatallah B. and Saint-Paul R. (2008), "Supporting the dynamic evolution of Web service protocols in service-oriented architectures", ACM Transactions on the Web, 2(2)
Ryu S., Saint-Paul R., Benatallah B. and Casati F. (2007), "A Framework for Managing the Evolution of Business Protocols in Web Services", Proc. of the 4th Asia-Pacific Conference on Conceptual Modelling, Ballarat, Victoria, Australia, pp. 49-59
Wang A.R., Li Z.W., Jia J.Y. and Zhou M.C. (2009), "An Effective Algorithm to Find Elementary Siphons in a Class of Petri Nets," IEEE Trans. on Sys., Man and Cybern., Part A, 39(4).
Li Z.W. and Zhou M.C. (2009), Deadlock Resolution in Automated Manufacturing Systems: A Novel Petri Net Approach, Springer, New York.
Hu H., Zhou M.C. and Li Z.W. (2009), "Liveness Enforcing Supervision of Video Streaming Systems Using Non-sequential Petri Nets," IEEE Transactions on Multimedia, 11(8), pp. 1446-1456.
Wu N.Q., Zhou M.C., and Li Z.W. (2008), "Resource-Oriented Petri Net for Deadlock Avoidance in Flexible Assembly Systems," IEEE Trans. on Systems, Man, and Cybernetics: Part A, 38(1), pp. 56-69.