Asynchronous Semantics and Anti-patterns for Interacting Web Services

Yongyan Zheng, Paul Krause
Department of Computing, University of Surrey
Guildford, GU2 7XH, UK
{y.zheng,p.krause}@surrey.ac.uk

Abstract

Web service is an emerging paradigm for distributed computing. First, in order to verify web services rigorously, it is important to provide a formal semantics for the flow-based web service language (WS). A suitable formal model should cover most features of the WS. The existing formal models either abstract from data, cover a simple subset of WS, or omit the interactions between certain components. This paper presents a web service automaton, an extension of Mealy machine, to fulfill the formal model requirements of the web service domain. Second, semantic compatibility checking between web services is another important issue. The existing checking approaches are post-checking, where the compatibility is checked after composition. As a complement to post-checking, we proposes anti-patterns for web service interactions as a pre-checking, so that certain incompatible web services can be modified or re-selected in the earliest stages.

1. Introduction

The web service paradigm provides a flexible, re-usable, and loosely coupled model for distributed computing. Because of these advantages, web service architectures have been actively researched in recent years, and various web service standards have been proposed such as WS-CDL [11] for service choreography, BPEL [2] for service orchestration, and WSDL [4]. WSDL is in the first layer to describe the service interface. BPEL is in the second layer to model the local behaviours of web service interactions. WS-CDL is the third layer to model the global behaviour of BPEL web service interactions. Due to the similarity of BPEL and WS-CDL, we believe a general formal model can provide semantics for both BPEL and WS-CDL. In this paper, we focus on defining the semantics of web service automata WSA (definition 2), and discuss a light-weight approach for checking semantic compatibility. We use the terms machines and automata interchangeably.

In the literature, there exist two machine communication models: 1) synchronous communication model (SC) i.e. rendezvous: a message needs to be sent and consumed synchronously; 2) asynchronous communication model (AC): machines are associated with buffers, such that a message can be sent and consumed asynchronously. The limitation of the SC model is that the partner machines can never be sure that a machine is in a state ready to accept its input. We believe the AC model is more suitable for web services, because web services are designed to be loosely-coupled and distributed, where each web service shall evolve independently. WSA is a variant of Communicating Finite State Machines (CFSM) [3], which abstracts from data and supports the AC model with FIFO queues. Our WSA includes multiple-event transitions, data and internal events. Inspired by Alur [1], we also consider various buffering schemes. We explicitly elicit the conditions for a WSA to ensure that the composition operation be commutative and associative. The discussion of the suitability of WSA for web services will be covered in section 2.

We will also discuss semantic compatibility checking. Existing approaches involve post-checking, where the compatibility is checked after composition. We propose to include pre-checks of the individual machines against a set of proposed anti-patterns over event occurrences, so that certain incompatible web services can be modified or re-selected in the earliest stages. We do not aim to guarantee a composite model to be deadlock or livelock free, but to provide a guideline for selecting the right automata. As a complement to post-checking, the pre-checking has dual advantages. First, less computation effort is required because the verification tools need not to explore the whole composite model to identify incompatible behaviours, but explore the individual models. Second, the event occurrence relation property of an individual machine can be used as a machine profile and re-used for further anti-pattern checking with different machines.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 defines the WSA semantics, composition semantics, and discusses different buffering schemas. Section 4 presents compatibility and anti-patterns for web service interactions. Section 5 concludes the paper.

2. Related Works and Motivation

There are a number of proposals for formal semantics, such as process algebras, petri nets, and automata for BPEL web service models. [13, 10] give good reviews of the existing web services techniques. BPEL consists of basic activities (e.g. receive, invoke, assign, reply) and structured activities (e.g. flow, sequence, while, switch, pick, scope, event handlers, and fault handlers). Structure activities impose constraints on a set of activities contained within them. The idea of mapping BPEL to formal models includes: 1) map each BPEL basic activity to a basic component; 2) map each BPEL structured activity to a composite component, which consists of a set of basic components; 3) map the whole BPEL process to a composite component, which consists of a set of composite components. Here we use component as a general term for either a process, a petri net, or an automaton.

A suitable formal model for BPEL should not only cover most features of BPEL language, but also be able to model the interactions between BPEL internal components as well as the interactions between BPEL processes. The existing models in the literature either abstract from data, internal events, or cover a simple subset of BPEL. Also, most approaches do not consider the interactions between BPEL internal components explicitly, but leave the internal components interact implicitly by shared variables. In the theoretical point of view, we believe it is clearer and simpler to provide a uniform means of interactions, instead of considering both shared variable and message passing mechanisms.

Forster [7] uses MSCs (Message Sequence Chart) as the web service specification, WS-CDL and BPEL as the implementations. First, MSC, WS-CDL, and BPEL are all transformed into FSPs (Finite State Processes). Second, by trace equivalence checking, the LTSA model checker accepts the input FSPs and model-checks the BPEL against the WS-CDL or MSC, or the WS-CDL against MSC. Since FSP abstracts from data and supports synchronous communication, we believe it is not suitable to model interactions of distributed web services. Koshkina [12] proposed a BPE-calculus as the semantics for BPEL, and applies the Concurrency Workbench (CWB) verification tool. The BPE-calculus abstracts from data. CWB has the advantage that it supports not only model checking for μ-calculus properties, but also pre-order checking and equivalence checking. Fu and Bultan [8] proposed a guarded automaton (GA) as the semantics for both BPEL and the Conversation Model. Their conversation model has the same role as WS-CDL. The model checker SPIN is used to verify the GA against LTL properties. Their GA is a non-deterministic CEFSM with constraints on the elements of transitions, without considering internal events. The interactions between BPEL processes is by message-passing, but they omit the interactions between internal components of a BPEL process.

Our WSA model aims to provide suitable semantics for BPEL-like web service language. We consider data and internal events in our model, and also we use message-passing as the only way for both BPEL internal component communications and BPEL process communications. The reason for us to include internal events is that internal events become interesting in the context of composition, where the external events of the individual machines will become internal events to a composite machine. Furthermore, WSA allows a transition to associate with multiple input events, which linked by logical operator conjunction, disjunction, or negation This feature reduces the unnecessary state space of a state machine, and enables WSA to capture most features of BPEL language in a natural way. For instance, a conjunction of events can model the case when BPEL flow activity waits for all its sub-activities to finish. A disjunction of events can model BPEL fault handling, while a conjunction of event A and the negation of an event B can model the case that B has higher priority than A, which is useful for modeling BPEL's interruption feature. The detailed mapping from BPEL to WSA is covered in [17].

For web service semantic compatibility checking, the current approaches [16, 6, 14] only check the compatibility after composition. The whole state space of the composite model needs to be explored for thorough checking, but such post-checking is expensive to find an interaction problem. Instead, we proposed anti-patterns over traces of individual machines, so that certain interaction problems can be identified by looking at the individual machines.

3. Web Service Automaton

3.1. Static Semantics

The static semantics of a web service automaton extends finite state machine with signature, data structure, and message storage schema. The dynamic semantics of a web service automaton includes machine configurations and execution traces. The formal definitions are given below.

Definition 1. We assume that we have available an enumerable infinite set V of variables and sets AX,BX of assignment expressions and Boolean expressions respectively, together with a set D of values. We also assume that we have a set of functions Env where $\epsilon \in Env : V \rightarrow D$

assigns variables of V with values from D. Given an expression exp, we need three functions:

- $def : AX \rightarrow V$, where $def(exp) \in V$ returns the assigned variable, i.e. the variable on the left hand side of the assignment.
- $cuses : AX \rightarrow \wp(V)$, where $\wp(V)$ is the power set of V and $cuses(exp) \subseteq V$ returns the variables on the right hand side of the assignment.
- $puses : BX \rightarrow \wp(V)$, where $puses(exp) \subseteq V$ returns the variables in the Boolean expression.

If $exp \in AX$, we need a function $apply_A : AX \times Env \rightarrow Env$ that satisfies:

- If $v \neq def(exp)$ then $apply_A(exp, \epsilon(v)) = \epsilon(v)$. This means only the assigned variable may change.
- If for all $v \in cuses(exp)$ we have $\epsilon_1(v) = \epsilon_2(v)$, then $apply_A(exp, \epsilon_1(v)) = apply_A(exp, \epsilon_2(v))$.
- If for all $\epsilon \in Env$ we have $apply_A(exp_1, \epsilon) = apply_A(exp_2, \epsilon)$, then $exp_1 = exp_2$.

If $exp \in BX$, we need a function $apply_G : BX \times Env \rightarrow \{true, false\}$ that satisfies: if for all $v \in puses(exp)$ we have $\epsilon_1(v) = \epsilon_2(v)$, then $apply_G(exp, \epsilon_1(v)) = apply_G(exp, \epsilon_2(v))$.

Each web service automaton M will be associated with certain variables and expressions. These constitute its data structure. A formal definition of a machine's data structure is given in definition 2.

Definition 2. A **Web Service Automaton** (WSA) M is a finite state machine, consisting of $WSA_M = (I_M, S_M, s_{0M}, S_{fM}, T_M, \delta_M)$. As a convention, we omit the subscript of M such that $M = (I, S, s_0, S_f, T, \delta)$, and the components of M inherit subscripts, superscripts, or dashes.

1) I is the signature of M, denoted as a three tuple $I = (E, L, O)$, where E, L, O are pair-wise disjoint and represent a set of input events, internal events, and output events, respectively. Let $Msg = (L \cup E \cup O)$ to be the set of events, we refer to the elements of $L = L_{in} \cup L_{out}$ as internal input events and internal output events, and to those of $Msg = (E \cup O)$ as external events.

2) S is a set of states, $s_0 \in S$ is the initial state, $S_f \subseteq S$ is a set of final states.

3) $T \subseteq IN \times BX \times (\wp(AX) \cup OUT)$ is a set of transitions, where $IN = (E \cup L_{in} \cup \{\Omega\})$ and $OUT = (O \cup L_{out} \cup \{\Omega\})$. For each $t = (m, g, a) \in T$ (graphically denoted as $m[g]/a$), $m \subseteq IN$ is a set of triggering events, $g \in BX$ is the guard predicate, and $a \subseteq (\wp(AX) \cup OUT)$ is the action set composed of assignments and output events. Ω indicates the omission of an event. We would represent a transition by $\Omega[g]/\Omega$ which simply determines a state change and nothing else. The elements of transition t are denoted as $t.m = m, t.g = g, t.a = a$.

   - The events of the transition input event set $t.m \subseteq IN$ are linked by logical operator conjunction, disjunction, or negation, denoted as $AND : e_1 \wedge e_2, .., \wedge e_n$, $OR : e_1 \vee e_2, .., \vee e_n$, and $NOT : \neg(e_i)$, respectively.
   - The *data structure* of machine M is the form of (V_M, AX_M, BX_M), which can be retrieved from T. Since $T \subseteq IN \times BX \times (\wp(AX) \cup OUT)$, we can retrieve $AX_M = \{exp \in AX | \exists t \in T \wedge exp \in t.a\}$, $BX_M = \{exp \in BX | \exists t \in T \wedge exp \in t.g\}$, and V_M which is the disjoin union of $\bigcup_{exp \in AX}(\{def(exp)\} \cup cuses(exp))$ and $\bigcup_{exp \in BX}\{puses(exp)\}$.
   - We define that a WSA is *T-complete* iff $t \in T$ then $s \xrightarrow{t} s'$, which means every transition will be triggered from some state.

4) $\delta \subseteq S \times T \times S$ is the transition relation.

We use symbols $!, ?, @$ as a convention in diagrams to indicate whether an event is input, output, or internal event, denoted as $!e \in E, ?e \in O, @e \in L$, respectively. Conversely, going from the formal description to a diagram, $!, ?, @$ are introduced depending on membership of E, O, L. For instance if M sends a message m to M', then in the composite automaton (definition 10), $!m$ and $?m$ will become $@!m$ and $@?m$ to indicate that output event !m and input event ?m become internal output and internal input events, respectively.

Definition 3. A WSA is **deterministic** iff $\forall s, s', s'' \in S. \forall t \in T, s \xrightarrow{t} s' \wedge s \xrightarrow{t} s'' \Rightarrow s' = s''$ holds.

Definition 4. We define a **message** x to be a pair of send and receive events $(!x, ?x)$. If machine M_1 sends message x to machine M_2, then $!x \in O_1 \wedge ?x \in E_2$.

3.2. Dynamic Semantics

We have defined the static structure of a WSA. We now explain its dynamic semantics. We assume, first of all, that a WSA is equipped with a means of storing incoming messages. Instead of limiting the buffers to a particular type such as FIFO or multi-set, we consider different buffering schemes (section 3.4). We assume that each WSA is associated with a **finite buffer**. The buffer $\beta(E \cup L_{in})$ stores the external and internal input events. In one **step** of the WSA, either: 1) a message e is received from the environment $e \in E$, and added to the buffer $\beta(E \cup L_{in})$; or 2) an enabled transition fires, possible causing a state change and change to the values of some variables $v \in V_M$. In order to precisely define a step, we need to formalise the notion of a **configuration** η which records the current state, the current values of the variables associated with M and the current content of the buffer.

A transition t is enabled in a configuration η, when 1) its triggering event set $t.m$ is either empty or belonging to $\beta(E \cup L_{in})$, 2) $t.m$ can be consumed according to the machine's buffering scheme, and 3) the transition guard $t.g$ is evaluated to be true. When a transition t is enabled, a set of actions $t.a$ is executed, and the state machine moves from the start state to the end state of t. Such transition is called **enabled transition**.

A machine is associated with a set of events, and an event may have multiple **event occurrences**. When considering a machine's dynamic behaviour, we need to distinguish event occurrences from events. Similarly, a message sent between machines may have multiple **message instances**, and message instances also need to be distinguished from messages. We define a function $\lambda : Ins \rightarrow C$ to map class instances Ins to classes C. 1) When applying λ to event occurrences and events, $\lambda(o) = e$ returns the event e for an event occurrence o. For an enabled transition t, if $?x \in t.m$ then we say t corresponds to the event occurrence o_i with $\lambda(o_i) =?x$. 2) When applying λ to message instances and messages, for a message instance $om = (!om, ?om)$, there exists message $m = (!m, ?m)$ such that $\lambda(!om) =!m$ and $lambda(?om) =?m$.

Definition 5. A **configuration** of a machine M is of the form $\eta = (s, B, \epsilon)$, where $s \in S, B \subseteq \beta(E \cup L_{in})$ is the current buffer content, and function $\epsilon : V \rightarrow D$ assigns the variable of V with a value from D. The set of configurations of M is denoted as $confs(M)$ where $\eta \in confs(M)$. Note that we assume the buffer is empty when M is either in the initial state or in a final state (hypothesis 1), so $\eta_0 = (s_0, \emptyset, \epsilon_0)$ and $\eta_n = (s_n, \emptyset, \epsilon_n)$ are the initial and a final configuration, respectively.

Hypothesis 1. The buffer is empty when machine M is in a final state. If M is in a final state, we say a lifecycle of the machine ends. If the buffer is not empty then the remaining messages will be discarded because they cannot be consumed within this lifecycle.

Definition 6. A **step** is a triple $(\eta, x, \eta') \in confs(M) \times (T \cup E \cup L_{in}) \times confs(M)$. A step $\eta \xrightarrow{x} \eta'$ changes machine M from configuration η to configuration η' in the following forms:

1) If $x \in E \cup L_{in}$, representing the case when a message x is arriving and is added to the buffer B, then $(s, B, \epsilon) \mapsto^x (s', B', \epsilon')$ iff $s = s', \epsilon = \epsilon', B' = B + \{x\}$.

2) If $x = t \in T$, representing the case when either the message is already in the buffer $t.m \in B$ or there is no triggering event associated with t, then $(s, B, \epsilon) \mapsto^t (s', B', \epsilon')$ iff t is an enabled transition, i.e. $(s, t, s') \in \delta, B' = (B \setminus \{t.m\} + (t.a \cap L_{in}), apply_G(t.g, \epsilon) = true$.

Definition 7. We define a set of **configuration sequences** of M as $configs(M, \eta_0)$. A configuration sequence of M,$\alpha = \langle \eta_0, x_0, .., \eta_n \rangle\rangle \in configs(M, \eta_0)$, is a sequence of configurations linked by input events,internal input events, or enabled transitions from the initial configuration to a final configuration, where $\eta_i = (s_i, B_i, \epsilon_i)$, $x \in E \cup L_{in} \cup T$.

1) We define a set of **transition sequences** as $trans(M)$, where a transition sequence of M is a sequence of enabled transitions $trans(\alpha) = \langle x_0, .., x_{n-1} \rangle$ such that $trans(M) = \{trans(\alpha)\}$, where $x_i \in T, \alpha = \langle \eta_0, x_0, .., \eta_n \rangle \in configs(M, \eta_0)$, and $trans(\eta_0) = \{\Omega\}$.

2) We define the set of **traces** as $trances(M)$. A trace of machine M is a non-empty sequence of external event occurrences $\langle o_1, .., o_n \rangle$, such that $trace(M) = \{\langle o_1, .., o_n \rangle\}$, where for all enabled transitions $trans(M)$, $\lambda(o_i) \in t.m \subseteq E$ or $\lambda(o_i) \in t.a \cap O \neq \emptyset$ holds.

Definition 8. Let $SReach(M) = \{s \in S | \exists \alpha \in traces(M).s_0 \xrightarrow{\alpha} s\}$ be the set of states reachable from the initial state of M, and let $TReach(M) = \{t \in T | \exists s \in SReach(M), \exists s' \in S.s \xrightarrow{t} s'\}$ be the set of reachable transitions. So a machine M is fully reachable iff $SReach(M) = S \wedge TReach(M) = T$.

3.3. Composition

In order for our notion of composition to be commutative and associative, we introduce the unordered Cartesian product (definition 9). The commutative property is easy to obtain based on the unordered Cartesian product, but the associative property requires further constraints on the individual WSAs. We define a general composite automaton with an interleaving semantics (definition 10). Such composite automaton is not guarantee to be a WSA (definition 11). We then elicit a set of constraints on the individual WSAs as the necessary, sufficient, or necessary and sufficient conditions for a composite automaton to be a WSA or to be a deterministic WSA. For strongly composable WSAs (definition 12), we prove that the buffer content of the composite automaton is a unique pair, the configurations as well as configuration sequences of individual WSAs can be projected from the composite automaton, and we also prove that the composition operator is associative. A detailed report with all the proofs is in [17].

Definition 9. If $(X_i)_{i \in I}$ is an indexed family of sets, we define $\prod_{i \in I} X_i$ to be the set of all functions $\omega : I \to \bigcup_{i \in I} X_i$ satisfying $\omega(i) \in X_i$ for each $i \in I$. We refer to $\prod_{i \in I} X_i$ as the **unordered Cartesian product** of a set of X_i. Note that $\prod$ is a commutative and associative binary operator.

Definition 10. If M_1, M_2 are WSA, then we define their Composite Automaton $\hat{M} = M_1 \parallel M_2$ to be the structure $\hat{M} = (\hat{I}, \hat{S}, \hat{s_0}, \hat{S_f}, \hat{T}, \hat{\delta})$, where

1) $\hat{I} = (\hat{E}, \hat{L}, \hat{O})$, we define com_{12} to be the common messages of M_1, M_2 by $com_{12} = (E_1 \cap O_2) \cup (E_2 \cap O_1)$. Now we can define $\hat{I}$ by $\hat{E} = (E_1 \cup E_2) \setminus com_{12}$, $\hat{L} = E_1 \cup E_2 \cup com_{12}$, and $\hat{O} = (O_1 \cup O_2) \setminus com_{12}$.

2) $\hat{S} = S_1 \prod S_2$.

3) $\hat{s_0} = \{s_0^1\} \prod \{s_0^2\}$.

4) $\hat{S_f} = \{S_f^1\} \prod \{S_f^2\}$. A state of $M_1 \parallel M_2, .., \parallel M_n$ is final if, for all machines, the local state s_i of M_i is final.

5) $\hat{T} = T_1 \cup T_2$. Note we assume that $\forall v \in V_i$ is a local variable of machine M_i, where $V_i = \{v \in V | \exists exp \in AX_i \cup BX_i\}$. This means that there are no shared variable between $t \in T_i$ and $t \in T_j$ where $i \neq j$.

6) $\hat{\delta} \subseteq \hat{S} \times \hat{T} \times \hat{S}$. If $t \in T_1$ then $(s_i^1, s_i^2) \xrightarrow{t} (s_j^1, s_j^2) \Leftrightarrow s_i^2 = s_j^2 \wedge s_i^1 \xrightarrow{t} s_j^1$, similarly if $t \in T_2$ then $(s_i^1, s_i^2) \xrightarrow{t} (s_j^1, s_j^2) \Leftrightarrow s_i^1 = s_j^1 \wedge s_i^2 \xrightarrow{t} s_j^2$. If follows from the *asynchronous interleaving semantics* that a transition of the composite automaton is either from M or M' but not from both machines.

Definition 11. M_1, M_2 are **composable** iff $Msg_1 \cap Msg_2 = com_{12} \cup (E_1 \cap E_2) \cup (L_1 \cap L_2) \cup (O_1 \cap O_2)$.

Definition 12. M_1, M_2 are **strongly composable** iff a) $Msg_1 \cap Msg_2 = com_{12}$, and b) $T_1 \cap T_2 = \emptyset$.

3.4. Buffering Schemes

Since there is more than one reasonable buffering mechanism, the buffering scheme of WSA should be flexible and configurable. Inspired by Alur [1], we identify the commonly used buffering schemes:

1) FIFO buffering scheme: A machine has one FIFO buffer. The CFSM [3] and the guarded automata [8] apply this scheme. No message overtaking is allowed. A message can be consumed only when it is at the head of the FIFO buffer.

2) Multi-set (definition 13) buffering schemes: a) Multi-set buffer with FIFO sub-buffers: A machine has one multi-set buffer that consists of FIFO sub-buffers, where each FIFO sub-buffer corresponds to a message type. A message can be consumed when it is at the head of each FIFO sub-buffer. b) Multi-set buffer without sub-buffers: A machine has one multi-set buffer. A message can be consumed as long as it is in the buffer. For those messages of the same type, the machine will randomly consume a message in the buffer. c) Multi-set buffer with multi-set sub-buffers: A machine has one multi-set buffer that consists of multi-set sub-buffers, where each multi-set sub-buffer corresponds to a message type. The machine consumes messages in the same way as of b).

Definition 13. A **finite multi-set** is formally defined as a pair (X, n), where X is some set and $n : X \to N$ is a function from X to the set N of natural numbers. A multi-set differs from a set in that each element has a multiplicity. For instance, $\{a, b, b, c, b, a\}$ is a multiset, and the multiplicities are $n(a) = 2, n(b) = 3, n(c) = 1$, respectively. The usual set operations such as union, intersection, and sum can be generalized for multisets.

Note that without loss of generality, we assume the communication channels are lossless, so that every message is always received after it is sent. Message overtaking may occur during communication, Figure 1 shows two types of message overtaking in MSCs.

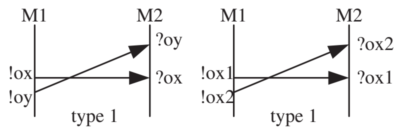

Figure 1. Message overtaking

For type 1, suppose messages $x = (!x, ?x)$ and $y = (!y, ?y)$, there exist message instances $ox = (!ox, ?ox)$ and $oy = (!oy, ?oy)$ where $\lambda(ox) = x$ and $\lambda(oy) = y$, respectively. Message overtaking happens when the order of receive event occurrences $?ox, ?oy$ is different from the order of send event occurrences $!ox, !oy$. For type 2, suppose message $x = (!x, ?x)$, there exist message instances $ox_1 = (!ox_1, ?ox_1)$ and $ox_2 = (!ox_2, ?ox_2)$ where $\lambda(ox_1) = \lambda(ox_2) = x$. Message overtaking happens when the order of receive event occurrences $?ox_1, ?ox_2$ is different from the order of send event occurrences $!ox1, !ox2$.

We assume that WSA with both FIFO and multi-set buffering schemes only allow type 1) but not type 2) of message overtaking.

4. Compatibility and Anti-patterns

In this section, we define syntax and semantic compatibility. Since syntax compatibility is easy to check from machine interfaces, we focus on checking the semantic compatibility by checking the individual machines against anti-patterns.

4.1. Compatibility

After selecting a set of candidate WSAs, we need to check whether the WSAs can interact properly as expected. Syntactic compatibility involves checking machine interfaces for the matching external events. Semantic compatibility means checking the machine behaviours for the absence of pathologies.

Definition 14. Two WSAs M_1, M_2 are **syntactically compatible** if M_1 sends a message x to M_2, then there exists $x = (!x, ?x)$ such that $!x \in O_1$ and $?x \in E_2$.

Definition 15. Two WSAs M_1, M_2 are **semantically compatible** if a) they are strongly composable, and b) $trans(M_1 \parallel M_2) \neq \emptyset$.

Now we discuss when the condition $trans(M_1 \parallel M_2) \neq \emptyset$ holds. This condition holds iff for any $\hat{\eta} = ((s_0^1, s_0^2), B, (\epsilon_0^1, \epsilon_0^2))$ there is no transition $t \in \hat{T}$ such that $\hat{\eta} \mapsto^t \hat{\eta}'$, where $(\epsilon_0^1, \epsilon_0^2)$ denotes the initial values of variables. First, suppose the initial configuration is $\hat{\eta}_0 = ((s_0^1, s_0^2), B, (\epsilon_0^1, \epsilon_0^2))$, if $B \in \beta(\hat{E})$ then there exists α such that $\hat{\eta}_0 \mapsto^\alpha \hat{\eta}_B = ((s_0^1, s_0^2), B + \alpha, (\epsilon_0^1, \epsilon_0^2))$, where α consists solely of external input events, i.e. $\alpha \in \hat{E}$. Second, from definition 10, it shows $(s_i^1, s_i^2) \xrightarrow{t} (s_j^1, s_j^2)$ iff a) if $t \in T_1$ then $s_i^2 = s_j^2 \wedge s_i^1 \xrightarrow{t} s_j^1$, or b) if $t \in T_2$ then $s_i^1 = s_j^1 \wedge s_i^2 \xrightarrow{t} s_j^2$. $\hat{\eta_B} \mapsto^{t \in T_1} \Leftrightarrow \eta_B^1 \mapsto^{t \in T_1}$ or $\eta_B^2 \mapsto^{t \in T_2}$. Here $\eta_B^i = (s_0^i, proj_i(B), \epsilon_0^i)$, and $proj$ is the projection operator. Without loss of generality suppose $\hat{\eta} \mapsto^{t \in T_2} \hat{\eta}'$, we have $s_0^1 \overset{t \in T_1}{\rightarrow} s_1^1$, $t.m \in B$, and $apply_G(t.g, \epsilon_1) = true$. Conversely, we have $\neg(\hat{\eta_B} \mapsto^{t \in T_1})$ iff for all B either $\neg(s_0^1 \overset{t \in T_1}{\rightarrow})$, $t.m \notin \hat{E}$, or $apply_G(t.g, \epsilon_1) \neq true$.

As a result, condition $trans(M_1 \parallel M_2) \neq \emptyset$ holds iff for $\forall B \in \beta(\hat{E})$, there does not exist $t \in \hat{T}$ such that $\hat{\eta_B} \mapsto^t$, which indicates that no transition is possible after receiving as many as external inputs.

4.2. Anti-patterns

According to definition 15, the condition $trans(M_1 \parallel M_2) \neq \emptyset$ can be checked only after constructing the composite WSA. This indicates that a thorough semantic compatibility checking has to be done by exploring the whole state space of the composite automaton. However we can speed up the model checking, if some obviously incompatible behaviours can be identified by only checking individual WSAs. We propose anti-patterns for such obviously incompatible behaviours. As a complementary approach to post-checking, we provide warnings so that the problematic WSA can be either re-selected or modified in the earliest stages. Furthermore, since a WSA's local ordering (definition 18) only needs to be computed once, the local ordering can be re-used for pre-checking the compatibility with other machines. After pre-checking, post-checking can be applied to thoroughly check the composite automaton for safety and liveness properties.

Referring to the event occurrences and message instances discussed in section 3.2, the anti-patterns discuss the temporal relations over event occurrences in traces of individual machines. We follow the standard definitions of **strict partial order** and **mutually exclusive** relation in set theory (e.g. [15]).

Definition 16. In a machine M, suppose $e, e' \in Msg_M$, the strict partial order over event occurrences $o_1 < o_2$ where $\lambda(o_1) = e, \lambda(o_2) = e'$ indicates that o_1 **happens before** o_2 in a configuration sequence of M.

Corollary 1. Suppose there is a message m from machine M_1 to M_2, if we have a message instance $om = (!om, ?om)$ where $\lambda(!om) = !m$ and $\lambda(?om) = ?m$, then the *strict partial order* over an event occurrence pair $!om < ?om$ enforces that a message instance must be received after it is sent.

Definition 17. In machine M, suppose $e, e' \in Msg_M$, the mutually exclusive relation on event occurrences $o_1 \sharp o_2$ where $\lambda(o_1) = e, \lambda(o_2) = e'$ indicates that o_1 is **branch-conflict** with o_2 in a configuration sequence of M. Intuitively, two branch-conflict event occurrences cannot happen in the same trace.

Definition 18. The **local ordering** on machine M_i is a structure $l_i = (C_i, <_i, \sharp_i)$, where C_i is the event occurrence set of $E_i \cup O_i$, $<_i$ and $\sharp_i$ are the strict partial order and the mutual exclusion relations on C_i, respectively.

Definition 19. For machines $\{M_1, .., M_n\}$, we define **message orderings** to be the structure $<_X = \bigcup_{\lambda(!om),\lambda(?om)\in X}(!om \quad <?om)$, where $om = (!om, ?om)$, $X \subseteq \bigcup_{1\leq i,j\leq n} com_{ij}$ is the set of messages sent between machines $\{M_1, .., M_n\}$, and $<$ is the strict partial order on event occurrence pairs.

Definition 20. The **causal ordering** for a set of machines $\{M_1, .., M_n\}$ is the structure $\prec_C = (\bigcup_{1\leq i\leq n} l_i) \cup <_X$, which describes the transitive closure of the set of local orderings and message orderings.

Definition 21. A machine M is said to be **blocking** iff there exists a state $s \notin S_f$ and a trace $\alpha \in traces(M)$ such that $s_0 \overset{\alpha}{\rightarrow} s$ and $\neg(s \overset{t}{\rightarrow})$ for $\forall t \in T$. Referring to definition 8, let S_d be the set of all blocking states,$S_d = \{s \in SReach(M) \setminus S_f : \forall t \in T.\neg(s \overset{t}{\rightarrow})\}$, so M is blocking iff $S_d \neq \emptyset$.

In state machine diagrams, an initial state is pointed by an arrow started with a filled black circle, and a final state is shaded. For the anti-patterns, we suppose for two messages $x = (!x, ?x)$ and $y = (!y, ?y)$ sent between machines M_1 and M_2, there exist message instances $ox = (!ox, ?ox)$ and $oy = (!oy, ?oy)$ of x and y, respectively, such that $\lambda(!ox) = !x$ and $\lambda(!oy) = !y$. Message Sequence Charts(MSCs) are used to show the anti-pattern scenarios. In the examples, we introduce index k to identify a message instance of a message, denoted as $oe[k] = (!oe[k], ?oe[k])$. The index k can be omitted when there is only one message instance.

Anti-Pattern 1. Suppose $!x \in O_1, ?x \in E_2$ and $!y \in O_2, ?y \in E_1$, $M_1 \parallel M_2$ has *unspecified reception* if $?oy <_1 !ox$ and $?ox <_2 !oy$ (1).

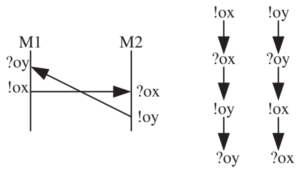

Figure 2. Unspecified reception

Figure 2 shows the corresponding MSC on the left and the causal ordering on the right. Machine M_1 sends message instance ox to machine M_2, and M_2 sends message instance oy to M_1. M_1 cannot send ox until it receives oy, while M_2 cannot send oy until it receives ox (1). Based on message ordering and (1), the causal ordering $\prec_c$ consists of $!ox <?ox <!oy <?oy$ and $!oy <?oy <!ox <?ox$. This conflict indicates that M_1, M_2 wait for message instances from each other but never get them. Hence, $M_1 \parallel M_2$ has an unspecified reception, where the blocking state sets of both machines are not empty.

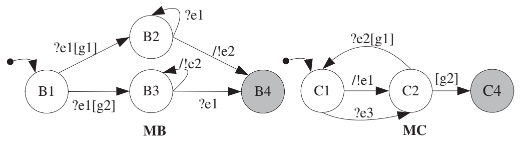

Figure 3. An example of anti-pattern1

Figure 3 illustrates an example. $traces(M_B)$ is composed of $\langle ?oe_1[1], ?oe_1[2], !oe_2\rangle$ and $\langle ?oe_1[1], !oe_2, ?oe_1[2]\rangle$. $traces(M_C)$ is composed of $\langle !oe_1[1], ?oe_2, !oe_1[2]\rangle$, $\langle !oe_1[1], ?oe_2, ?oe_3\rangle$, $\langle !oe_1[1]\rangle$, and $\langle ?oe_3\rangle$. The partial order $?oe_1[2] <_B !oe_2$ and $?oe_2 <_C !oe_1[2]$ can be obtained from one of the traces of M_B and the trace of M_C, so according to anti-patten1, $M_B \parallel M_C$ will deadlock when $?oe_1[2]$ happens before $!oe_2$ in M_B and $?oe_2$ happens before $?oe_1[2]$ in M_C. The blocking states are $S_d^B = \{B_2\}$ and $S_d^C = \{C_2\}$.

Anti-Pattern 2. Suppose $!x \in O_1, ?x \in E_2$ and $!y \in O_1, ?y \in E_2$, $M_1 \parallel M_2$ has *non-local branching choice* if $!ox\sharp_1!oy$ (2).

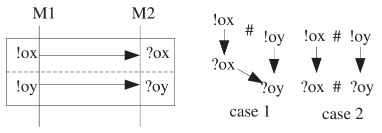

Figure 4. Non-local branching 1

Figure 4 shows the MSC and casual ordering. M_1 sends message instances ox, oy to M_2. In M_1, a send event occurrence of ox is in branch-conflict with a send occurrence of oy (2). Two cases may exist in M_2:

1) If $?ox <_2 ?oy$, M_2 waits for both message instances ox and oy from M_1, but M_1 cannot send these message instances in the same run due to (2). We have $\prec_c$:$!ox\sharp!oy \wedge !ox <?ox \wedge !oy <?oy \wedge ?ox <?oy$. By $!ox\sharp_1!oy$ and message ordering, we have $ox \Rightarrow \neg(!oy) \Rightarrow \neg(?oy)$ and $oy \Rightarrow \neg(!ox) \Rightarrow \neg(?ox)$, so $?ox <_2 ?oy$ does not hold.

2) If $?ox\sharp_2?oy$, we have casual ordering $\prec_c$: $!ox\sharp!oy \wedge !ox <?ox \wedge !oy <?oy \wedge ?ox\sharp?oy$. If $!ox$ (resp. $!oy$) happens in M_1 and $?oy$ (resp. $?ox$) happens in M_2, then M_2 will wait for message instance oy (resp. ox) forever due to (2). The blocking state set of M_2 is not empty.

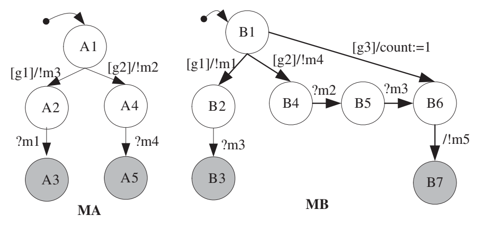

Figure 5. An example of anti-patterns2,3

Figure 5 shows an example, we have $!om_2 \sharp_A !om_3$. In case 1), we have $?om_2 <_B ?om_3$, so $M_A \parallel M_B$ has non-local branching choice with $S_d^B = \{B_4\}$ or $S_d^B = \{B_5\}$. In case 2), we have $?om_2 \sharp_B ?om_3$, so $M_A \parallel M_B$ has non-local choice with $S_d^B = \{B_2\}$ or $S_d^B = \{B_4\}$.

Anti-Pattern 3. Suppose $!x \in O_1, ?x \in E_2$ and $!y \in O_2, ?y \in E_1$, $M_1 \parallel M_2$ has *non-local branching choice* if $!ox \sharp_1 ?oy$ and $?ox \sharp_2 !oy$ (3).

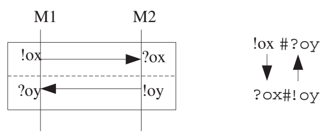

Figure 6. Non-local branching 2

Figure 6 shows the MSC and casual ordering. M_1 sends message instance ox to M_2, and M_2 sends message instance oy to M_1. The causal ordering is $\prec_c$:$!ox <?ox$, $!oy <?oy$, $!ox \sharp ?oy$, and $?ox \sharp !oy$. If $?oy$ happens in M_1 and $?ox$ happens in M_2, machine M_1 (resp. M_2) will wait for message instance oy (resp. ox) forever due to (3). The blocking state sets of both machines are not empty.

In Figure 5, $!om_3 \sharp_A ?om_4$ and $?om_3 \sharp_B !om_4$ hold. When $?om_4$ happens in M_A and $?om_3$ happens in M_B, $M_A \parallel M_B$ has non-local choice and each machine has blocking states $S_d^A = \{A_4\}$ and $S_d^B = \{B_2\}$. Similarly, $!om_2 \sharp_A ?om_1$ and $?om_2 \sharp_A !om_1$ will cause non-local choice. In this example, $om_2 \sharp_A om_3$ will also cause non-local choice with the type of anti-pattern2.

The above anti-patterns can be encoded into temporal logics, so that model checking tools (e.g. [9, 5]) can be used to automate the pre-checking.

5. Conclusion

We present Web Service Automata as a formal semantics for web services, and identify anti-patterns for web service interactions. The web service automaton is more general than the existing automata-based semantics because we include multiple-event transitions, data, and internal events. We use message-passing as the uniform mechanism for machine communications. The anti-pattern pre-checking has the advantage that it needs less computation effort to identify incompatible behaviours, and the corresponding properties can be reused. A tool is developed to implement the mapping from BPEL to WSAs, and from WSAs to SMV language [5]. Work is in hand to apply model checking tools to realize our anti-pattern pre-checking.

References

[1] R. Alur, G. J. Holzmann, and D. Peled. An analyser for mesage sequence charts. In *Proc. of TACAs*, pages 35–48. Springer-Verlag, 1996.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services version 1.1. May 2003.

[3] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. March 2001.

[5] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *Pro. of CAV*, pages 495–499. Springer-Verlag, 1999.

[6] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *Proc. of ICWS*, page 738. IEEE Computer Society, 2004.

[7] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based analysis of obligations in web service choreography. In *Proc. of AICT-ICIW*, page 149. IEEE Computer Society, 2006.

[8] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005.

[9] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[10] R. Hull and J. Su. Tools for design of composite web services. In *Proc. of SIGMOD*, pages 958–961. ACM Press, 2004.

[11] N. Kavantzas, D. Burdett, and G. Ritzinger. Web services choreography description language version 1.0. April 2004.

[12] M. Koshkina and F. van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[13] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 08(6):51–59, 2004.

[14] N. Milanovic and M. Malek. Architectural support for automatic service composition. In *Proc. of SCC*, pages 133–140. IEEE Computer Society, 2005.

[15] Wikipedia. *http://en.wikipedia.org/wiki*.

[16] A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold. Matchmaking for business processes based on choreographies. In *Proc. of EEE*, pages 359–368. IEEE Computer Society, 2004.

[17] Y. Zheng and P. Krause. Asynchronous semantics for interacting web services. Technical report, University of Surrey, 2006.