An Application of Web-Service Interfaces *

Dirk Beyer
Simon Fraser University
Canada

Arindam Chakrabarti
UC Berkeley
USA

Thomas A. Henzinger
EPFL Lausanne
Switzerland

Sanjit A. Seshia
UC Berkeley
USA

Abstract

We present a case study to illustrate our formalism for the specification and verification of the method-invocation behavior of web-service applications constructed from asynchronously interacting multi-threaded distributed components. Our model is expressive enough to allow the representation of recursion and dynamic thread creation, and yet permits the algorithmic analysis of the following two questions: (1) Does a given service satisfy a safety specification? (2) Can a given service be substituted by a another service in an arbitrary context? Our case study is based on the Amazon.com E-Commerce Services (ECS) platform.

1 Introduction

In this era of global-scale networked computing, disparate software components developed, maintained, used, and managed by different organizations, communicate with each other and collaborate over the internet to manage global supply chains, e-commerce platforms, financial systems, etc. In the past, a number of networked computing standards like COM, DCOM, RPC, RMI, CORBA, or .NET Remoting have been implemented by the major players in the software-engineering industry. However, the platform-independent, language-neutral, XML-based web-service framework seems poised to become the standard going forward.

Previously, the vast majority of business software applications used to be either developed completely in-house in the end-user organization, put together inside the end-user organization with standard components bought off-the-shelf, or developed by software firms contracted by the end-user organization for the job, and then maintained and run completely inside the end-user organization. The advent of web-service standards has completely changed the scenario. These standards allow the development of cross-platform, cross-organization, global-scale computing applications. However, for this very reason, they introduce new software specification, verification, and testing challenges. It is no longer possible for the end-user organization or the contractually hired software-developer organization to verify or test the entire application. No one organization has the entire source code for, or administrative control over the entirety of, an application such as an e-commerce engine built on the widely-used networked computing infrastructure known as the Amazon.com E-Commerce Services (ECS) platform.

Standards such as the Business Process Execution Language for Web Services (BPEL4WS) and the Web Service Choreography Interface (WSCI) have been proposed to allow distributed system components to be described behaviorally. However, these standards were not developed focusing on software-verification applications, and have hence turned out to be ill-suited for such purposes, primarily because they do not have a clearly defined formal semantics.

In formal modeling, much recent work has focused on the behavioral representation and verification of web services. Finite-state models for asynchronously communicating components [6, 7, 12, 14] allow checking safety properties for web services. Several communication models have been analyzed [18] and conditions proposed allowing the analysis of components communicating asynchronously through unbounded queues [4, 13, 16]. Hierarchical state machines [3] have been proposed to achieve succinctness compared to flat finite-state models. Temporal logics for specifying and verifying temporal properties of web services [21, 22], as well as logics to model dynamic contracts between services [9] have been proposed. Several other interesting models for e-business conversations have been put forward, e.g. based on the π-calculus [19, 20], communicat-

*This research was supported in part by the NSF grant CCR-0225610 and by the Swiss National Science Foundation.

ing sequential processes [8], abstract state machines [11], and Petri nets [15, 23]. Other work has looked at handling data [2, 17] and modeling data-driven web services [10].

Our web-service interface formalism [5] focuses exclusively on the web-method invocation behavior of web-service applications and provides the constructs of recursion, sequential composition, and two kinds of parallel composition that are useful in this context. We find this explicit representation of concurrency and recursion useful for representing web services. The formalism is restricted in such a manner that the questions of reachability and simulation checking are decidable.

This allows us to address the following two questions of interest. First, does a given service satisfy a safety specification given as a formula of linear temporal logic? This is the *specification-checking* problem. Second, can a service $\mathcal{P}$ be replaced with an alternative service $\mathcal{P}'$ in an arbitrary context ? For instance, if $\mathcal{P}$ represents the FedEx web service, and $\mathcal{P}'$ represents the UPS web service, then if such a substitutivity relation should hold, it would imply that an organization such as Amazon.com currently partnering with FedEx could simply add UPS as a new partner without having to make any changes to the rest of its networked computing infrastructure. We call this the *substitutivity* problem.

In this paper we illustrate the usefulness of this approach for the verification of distributed asynchronous multi-threaded component-based systems. We choose a case study based on the Amazon.com E-Commerce Services (ECS) platform. The specification language presented in [5] is enriched to allow greater expressivity.

The rest of the paper is organized as follows. In Section 2 we present the preliminaries of our formalism with examples illustrating the definitions. In Section 3 we revisit the questions of checking safety specifications and substitutivity. The main section, Section 4, presents the case study based on the ECS distributed computing platform.

2 Preliminaries

In this section we review the web-service interface formalism presented in [5]. Let $\mathcal{M}$ and $\mathcal{O}$ be finite sets of *web methods* and *outcomes*, respectively. Outcomes are associated with calls to web methods, and encode parameters passed to the web method, return values from the web method, and other behavioral differences between various calls to the web method; for instance, whether the invocation was synchronous or asynchronous, or in the latter case, if it will lead to a callback. For instance, the action $\langle \texttt{ShipItems}, \texttt{OK} \rangle$ represents the invocation of the web method `ShipItems` that eventually completes successfully, whereas the action $\langle \texttt{ShipItems}, \texttt{FAIL} \rangle$ represents the invocation of the same web method that eventually returns a failure code. Let $\mathcal{A} \subseteq \mathcal{M} \times \mathcal{O}$ be the set of *actions*. We present the following formalism to formally model the web method invocation behavior exhibited by web services.

2.1 Syntax

The set of *terms* over a set of actions $\mathcal{A}$ is defined by the following grammar for elements *term* $\in$ *terms* ($a \in A$ and $A \subseteq \mathcal{A}$, $|A| \geq 2$):

$$\textit{term} \quad ::= \quad a \mid \sqcap A \mid \boxplus A$$

A *protocol automaton* $\mathcal{F}$ is a tuple $(Q, \bot, \delta)$, where Q is a finite set of control *locations*, $\bot \in Q$ is the *return location*, and $\delta : (Q \setminus \{\bot\}) \rightarrow (\textit{terms} \times Q)$ is the *switch function* of the protocol automaton, which assigns to each location different from $\bot$ a *term* and a successor location. A *web-service protocol interface* (web-service interface for short) $\mathcal{P}$ is a tuple $\mathcal{P} = (\mathcal{D}, \mathcal{F})$, where $\mathcal{D} : \mathcal{A} \rightarrow 2^Q$ is a partial function that assigns to an action a set of control locations, and $\mathcal{F}$ is a protocol automaton.

2.2 Semantics

The execution of an action a begins at one of the locations from $\mathcal{D}(a)$. A switch $\delta(q) = (\textit{term}, q')$ means that, if the automaton is currently in location q, it recursively invokes *term*, and remembers the successor location q' as the return location, where control returns when the recursive invocation of *term* terminates. If the automaton reaches the location $\bot$, it returns control to the caller; a return for the very first invocation of action a leads to termination of the execution. The term $a = \langle m, o \rangle$ represents a call to web method m with expected outcome o. The term $\sqcap A$ represents the invocation of actions $a \in A$ in parallel; the execution of the term is not complete until all parallel invocations are complete and return. The term $\boxplus A$ represents the invocation of actions $a \in A$ in parallel; the execution of the term is not complete until at least one of the parallel invocations is complete and returns. We require that starting from any location of the automaton as the current location, the execution of the interface can be completed.

At each stage of the execution, the set of actions invoked in parallel is generated as output. A fresh symbol *ret* is generated as output whenever any invoked action completes execution and returns control (or overall execution completes). The result of the execution is the sequence of the generated output sets. Thus, action invocations and returns are *visible* [1] in the generated execution trace. Intuitively, this is equivalent to giving our analysis algorithm the ability to monitor the local stack of the system, in order to be able to answer questions such as web-service substitutivity defined as simulation in terms of the web method invocation behavior.

Formally, a web-service interface $\mathcal{P}$ denotes an *underlying transition relation* defined over an infinite set of trees. Given a finite set L of *symbols*, a *tree* t over L is a partial function $t : \mathbb{N}^* \to L$, where $\mathbb{N}^*$ denotes the set of finite words over the set $\mathbb{N}$ of natural numbers, and the domain $dom(t) = \{p \in \mathbb{N}^* \mid \exists (p, l) \in t\}$ is prefix-closed. Each element from $dom(t)$ represents a *node* of the tree t, and each node p is named with the symbol $t(p)$. The root is represented by the empty word ρ. The set of child nodes of node p in tree t is denoted by $ch(t, p) = \{p' \mid \exists b \in \mathbb{N} : p' = p \cdot b \ \wedge \ p' \in dom(t)\}$, where $\cdot$ is the concatenation operator. The set of leaf nodes of a tree t is $leaf(t) = \{p \in dom(t) \mid ch(t, p) = \emptyset\}$. The set of all trees over a finite set L is denoted by $\mathcal{T}(L)$.

Given a protocol interface $\mathcal{P} = (\mathcal{D}, \mathcal{F})$, the *underlying transition relation* of $\mathcal{P}$ is a labeled transition relation $\rightarrow_{\mathcal{P}} \subseteq \mathcal{T}(Q^{\boxplus}) \times 2^{\mathcal{A} \cup \{ret\}} \times \mathcal{T}(Q^{\boxplus})$, where the states are trees over the tree symbols $Q^{\boxplus} = Q \times \{\boxplus, \circ\}$, where Q is the set of locations of protocol automaton $\mathcal{F}$, and the transitions between states are labeled with sets of elements from $\mathcal{A} \cup \{ret\}$. We write $t \xrightarrow{A} t'$ for $(t, A, t') \in \rightarrow_{\mathcal{P}}$. In the rules below, if action c is supported by $\mathcal{P}$, then q_c is an element of $\mathcal{D}(c)$, otherwise q_c is $\perp$. The relation $\rightarrow_{\mathcal{P}}$ is defined as follows:

- Call: $t \xrightarrow{\{a\}} t'$ if there exists a node p such that $p \in leaf(t)$, $t(p) = q\circ$, and $\delta(q) = (a, q')$ is a switch of $\mathcal{F}$, and $t' = (t \setminus \{(p, q\circ)\}) \cup \{(p, q'\circ), (p \cdot 0, q_a\circ)\}$.

- Fork: $t \xrightarrow{A} t'$ if there exists a node p such that $p \in leaf(t)$, $t(p) = q\circ$, and $\delta(q) = (\sqcap A, q')$ is a switch of $\mathcal{F}$, and $t' = (t \setminus \{(p, q\circ)\}) \cup \{(p, q'\circ), (p \cdot 0, q_{a_0}\circ), (p \cdot 1, q_{a_1}\circ), \ldots, (p \cdot k, q_{a_k}\circ)\}$, where $A = \{a_0, a_1, \ldots, a_k\}$.

- Fork-Choice: $t \xrightarrow{A} t'$ if there exists a node p such that $p \in leaf(t)$, $t(p) = q\circ$, and $\delta(q) = (\boxplus A, q')$ is a switch of $\mathcal{F}$, and $t' = (t \setminus \{(p, q\circ)\}) \cup \{(p, q'\boxplus), (p \cdot 0, q_{a_0}\circ), (p \cdot 1, q_{a_1}\circ), \ldots, (p \cdot k, q_{a_k}\circ)\}$, where $A = \{a_0, a_1, \ldots, a_k\}$.

- Return: $t \xrightarrow{\{ret\}} t'$ if there exists a node $p \cdot b$, where $b \in \mathbb{N}$, such that $p \cdot b \in leaf(t)$, $t(p \cdot b) = \perp\circ$, $t(p) = q\circ$, and $t' = t \setminus \{(p \cdot b, \perp\circ)\}$.

- Return & Remove Sibling Tree: $t \xrightarrow{\{ret\}} t'$ if there exists a node $p \cdot b$, where $b \in \mathbb{N}$, such that $p \cdot b \in leaf(t)$, $t(p \cdot b) = \perp\circ$, $t(p) = q\boxplus$, and $t' = (t \setminus \{(p \cdot p', q'?) \mid p' \in \mathbb{N}^* \wedge q'? \in Q^{\boxplus}\}) \cup \{(p, q\circ)\}$.

A *run* of a transition relation is an alternating sequence of trees and sets of actions $t_0, A_1, t_1, A_2, t_2, \ldots$ with $\forall i \in \{1, \ldots n\} : t_{i-1} \xrightarrow{A_i} t_i$. A *trace* is the projection of a run to its action sets, e.g., for the run $t_0, A_1, t_1, A_2, t_2, \ldots$, the corresponding trace is $A_1, A_2, \ldots$; for a location q, a *q-run* is a run $t_0, A_1, t_1, \ldots$ with $t_0 = \{(\rho, q\circ)\}$, i.e., a run starting from location q; and a *q-trace* is the trace corresponding to a q-run.

Example 1 Let us consider a simple calendar management web-service interface $\mathcal{P} = (\mathcal{D}, \mathcal{F})$ that supports the actions $\langle \texttt{OrganizeMeeting}, \texttt{OK} \rangle$ and $\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle$, and requires the actions $\langle \texttt{Busy}, \texttt{OK} \rangle$ and $\langle \texttt{SendConfirmationEmail}, \texttt{OK} \rangle$. The function $\mathcal{D}$ together with the protocol automaton $\mathcal{F}$ are given as follows, where an arrow $a \mapsto q$ represents a pair of function $\mathcal{D}$, a triple $q : (term, q')$ represents a switch of $\mathcal{F}$:

$\langle \texttt{OrganizeMeeting}, \texttt{OK} \rangle \mapsto$
  $q_0 : (\langle \texttt{SyncCalendars}, \texttt{OK} \rangle, q_1)$
  $q_1 : (\langle \texttt{Confirm}, \texttt{OK} \rangle, \perp)$
$\langle \texttt{SyncCalendars}, \texttt{OK} \rangle \mapsto$
  $q_2 : (\sqcap \{\langle \texttt{AddEventToCalendar}_1, \texttt{OK} \rangle,$
    $\langle \texttt{AddEventToCalendar}_2, \texttt{OK} \rangle, \ldots,$
    $\langle \texttt{AddEventToCalendar}_\texttt{k}, \texttt{OK} \rangle\}, \perp)$
$\langle \texttt{Confirm}, \texttt{OK} \rangle \mapsto$
  $q_3 : (\langle \texttt{SendConfirmationEmail}, \texttt{OK} \rangle, \perp)$
$\langle \texttt{Confirm}, \texttt{OK} \rangle \mapsto \perp$
$\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle \mapsto \perp$
$\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle \mapsto$
  $q_4 : (\langle \texttt{Busy}, \texttt{OK} \rangle, \perp)$

This web-service interface has the following semantics. The execution begins with the invocation of action $\langle \texttt{OrganizeMeeting}, \texttt{OK} \rangle$, which leads to a number of parallel invocations of $\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle$ (one for each desired participant in the meeting being scheduled), sequentially followed by $\langle \texttt{SendConfirmationEmail}, \texttt{OK} \rangle$, which can only be invoked after all parallel invocations of $\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle$ have completed their execution. In each invocation of the later action, the system may find out that the participant concerned is busy (leading to the invocation of $\langle \texttt{Busy}, \texttt{OK} \rangle$) or not. After all parallel invocations have finished, the system nondeterministically chooses to send a confirmation email to all participants noting that the meeting has been scheduled, or not: the actual implementation could decide to send the confirmation only if none of the $\langle \texttt{AddEventToCalendar}_\texttt{i}, \texttt{OK} \rangle$ actions led to the invocation of $\langle \texttt{Busy}, \texttt{OK} \rangle$, i.e., all desired participants were indeed found to be free.

Now let us consider the following, slightly modified version $\mathcal{P}'$ of the calendar management service:

$\langle \texttt{OrganizeMeeting}, \texttt{OK} \rangle \mapsto$
  $q_0 : (\langle \texttt{SyncCalendars}, \texttt{OK} \rangle, q_1)$
  $q_1 : (\langle \texttt{Confirm}, \texttt{OK} \rangle, \perp)$

$\langle \texttt{SyncCalendars}, \texttt{OK}\rangle \mapsto$
$\quad q_2 : (\boxplus\{\langle \texttt{AddEventToCalendar}_1, \texttt{OK}\rangle,$
$\qquad \langle \texttt{AddEventToCalendar}_2, \texttt{OK}\rangle, \ldots,$
$\qquad \langle \texttt{AddEventToCalendar}_k, \texttt{OK}\rangle\}, \bot)$
$\langle \texttt{Confirm}, \texttt{OK}\rangle \mapsto$
$\quad q_3 : (\langle \texttt{SendConfirmationEmail}, \texttt{OK}\rangle, \bot)$
$\langle \texttt{Confirm}, \texttt{OK}\rangle \mapsto \bot$
$\langle \texttt{AddEventToCalendar}_i, \texttt{OK}\rangle \mapsto \bot$
$\langle \texttt{AddEventToCalendar}_i, \texttt{OK}\rangle \mapsto$
$\quad q_4 : (\langle \texttt{Busy}, \texttt{OK}\rangle, \bot)$

This modified web service does not wait for the execution to complete on all parallel invocations of $\langle \texttt{AddEventToCalendar}_i, \texttt{OK}\rangle$. As soon as the first participant's status (busy or not) can be obtained, the system is free to move forward and decide whether to schedule the meeting or not. ∎

3 Specifications and Substitutivity

3.1 Specifications

We now present an extended version of the specification language introduced in [5]. There, we allowed specifications of the form $a \not\rightsquigarrow \hat{\varphi}$, where $a \in \mathcal{A}$ and $\hat{\varphi}$ is a temporal-logic formula of the form $(\neg C)\ \mathcal{U}\ B$ ("not C until B"), with $C, B \subseteq \mathcal{A}$. In this paper, we use a richer language and illustrate its use on the examples.

A *specification* ψ for a web-service interface $\mathcal{P}$ is a temporal safety property of the form $\psi = a \not\rightsquigarrow \varphi$, where φ is a temporal-logic formula of the following form:

$$\varphi ::= \phi \wedge (\phi\, \mathcal{U}\, \varphi) \mid \phi$$
$$\phi ::= \textsc{t} \mid \textsc{f} \mid b \mid \neg b \mid \phi \wedge \phi \mid \phi \vee \phi$$

where $a, b \in \mathcal{A}$.

The temporal-logic formula $\phi\ \mathcal{U}\ \varphi$ (read "ϕ until φ") represents a temporal property of traces. Intuitively, a trace satisfies a formula $\phi\ \mathcal{U}\ \varphi$ if it satisfies φ eventually, and satisfies ϕ at each step until then.

Formally, a web-service interface $\mathcal{P}$ *satisfies* a specification $\psi = a \not\rightsquigarrow \varphi$ (denoted $\mathcal{P} \models \psi$) if there exists no q-trace $\sigma = S_1 \cdot S_2 \cdot \ldots \cdot S_k$ of $\mathcal{P}$ such that $\sigma \models_t \varphi$ with $q \in \mathcal{D}(a)$. The satisfaction relation $\models_t$ between traces and temporal-logic formulae is defined as follows. For the trace $\sigma = S_1 \cdot S_2 \cdot \ldots \cdot S_k$, we have $\sigma \models_t a$ if $a \in S_1$ and $\sigma \models_t \neg a$ otherwise; and $\sigma \models_t \phi_1 \wedge \phi_2$ if $\sigma \models_t \phi_1$ and $\sigma \models_t \phi_2$; and $\sigma \models_t \phi_1 \vee \phi_2$ if $\sigma \models_t \phi_1$ or $\sigma \models_t \phi_2$; and $\sigma \models_t \textsc{t}$ for all σ; and $\sigma \models_t \textsc{f}$ for no σ. For a trace $\sigma = S_1 \cdot S_2 \cdot \ldots \cdot S_k$, we have $\sigma \models_t \phi_1\, \mathcal{U}\, (\phi_2 \wedge (\phi_3\, \mathcal{U}\, \varphi))$ if either (1) $\bigwedge S_1 \wedge (\neg \bigvee(\mathcal{A} \setminus S_1)) \Rightarrow \phi_1$ and $\sigma' \models_t \phi_2$ and $\sigma' \models_t \phi_3\, \mathcal{U}\, \varphi$ where $\sigma' = S_2 \cdot S_3 \cdot \ldots \cdot S_k$, or (2) $\sigma \models_t \phi_2$ and $\sigma \models_t \phi_3\, \mathcal{U}\, \varphi$.

We are able to check specifications in this enriched language using an algorithm that is similar to the existing algorithm [5], by using the modified set of production rules as presented in Figures 1 and 2. The modified rules have been augmented to consider the extended temporal logic, the interleaving semantics of executions by parallel subtrees of the overall system configuration at each stage, and the non-completing semantics of the operator $\boxplus$, as appropriate. For ease of presentation, we use the following abbreviations in Figures 1 and 2. Let $\mathbb{N}_{2j+1,2k+1}$ with $j \leq k$ be the set of naturals $\{2j+1, 2j+1, \ldots, 2k, 2k+1\}$. Given a set of naturals $\mathbb{N}_{2j+1,2k+2} \setminus \{2i \mid i \in N\}$ where N is some subset of $\mathbb{N}$ such that $2k+2 \notin N$, then $\varphi^0_{2j+1,2k+2}$ is an abbreviation for the formula $\phi_{2j+1}\, \mathcal{U}\, (\phi_{2j+2} \wedge (\phi_{2j+3}\, \mathcal{U}\, (\phi_{2j+4} \wedge \ldots\, \mathcal{U}\, (\phi_{2k} \wedge (\phi_{2k+1}\ \mathcal{U}\ \phi_{2k+1})))))$ where $\phi_{2i} = \textsc{t}$ for every $i \in N$. Given a set of naturals $\mathbb{N}_{2j+1,2k+1} \setminus \{2i \mid i \in N\}$ where N is some subset of $\mathbb{N}$, then $\varphi_{2j+1,2k+1}$ is an abbreviation for the formula $\phi_{2j+1}\ \mathcal{U}\ (\phi_{2j+2} \wedge (\phi_{2j+3}\ \mathcal{U}\ (\phi_{2j+4} \wedge \ldots\, \mathcal{U}\, (\phi_{2k} \wedge \Box\phi_{2k+1}))))$ where $\phi_{2i} = \textsc{t}$ for every $i \in N$. Given a set of naturals $\mathbb{N}_{2j+1,2k+1} \setminus \{2i \mid i \in N\}$ where N is some subset of $\mathbb{N}$, then $\varphi'_{2j+1,2k+1}$ is an abbreviation for the formula $\phi_{2j+1}\ \mathcal{U}\ (\phi_{2j+2} \wedge (\phi_{2j+3}\ \mathcal{U}\ (\phi_{2j+4} \wedge \ldots\, \mathcal{U}\, (\phi_{2k} \wedge (\phi_{2k+1}\ \mathcal{U}\ \textsc{t})))))$ where $\phi_{2i} = \textsc{t}$ for every $i \in N$.

Example 2 Let us consider the calendar management web-service interface $\mathcal{P}$ as defined in Example 1, and the specification $\psi = \langle \texttt{OrganizeMeeting}, \texttt{OK}\rangle \not\rightsquigarrow \textsc{t}\ \mathcal{U}\ (\langle \texttt{SendConfirmationEmail}, \texttt{OK}\rangle \wedge (\textsc{t}\ \mathcal{U}\ \langle \texttt{Busy}, \texttt{OK}\rangle))$. Intuitively, the specification ψ represents the question "Is there an execution trace resulting from the invocation of action $\langle \texttt{OrganizeMeeting}, \texttt{OK}\rangle$ on which at least one desired participant is found to be busy after the email confirmation for the meeting had already been sent out?" As expected, we find that $\mathcal{P} \models \psi$, and $\mathcal{P}' \not\models \psi$. ∎

3.2 Substitutivity

Substitutivity between web-service interfaces is defined in terms of simulation of execution traces: if for every web method a that is supported by $\mathcal{P}$, the underlying transition system representing the invocation of a on $\mathcal{P}'$ can be simulated by that representing the invocation of a on $\mathcal{P}$, then $\mathcal{P}$ can be safely substituted with $\mathcal{P}'$ in any arbitrary context (denoted $\mathcal{P}' \preccurlyeq \mathcal{P}$) [5].

4 Case Study

We present the following case study on using our formalism to formally model a web-based sales system $\mathcal{P}$ that is built using the Amazon.com E-Commerce Services (ECS) platform.

$$\frac{}{q \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge (\phi_3 \,\mathcal{U}\, (\phi_4 \wedge \ldots \,\mathcal{U}\, \phi_{2k})))} \qquad \begin{array}{l}(\delta(q) = (c, q') \ \wedge \ (c \ \wedge \ \neg \bigvee(\mathcal{A} \setminus \{c\})) \Rightarrow \bigwedge_{1 \le i \le k} \phi_{2i}) \ \vee \\ (\ \delta(q) = (\circ A, q') \ \wedge \ (\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow \bigwedge_{1 \le i \le k} \phi_{2i}) \\ \circ \in \{\sqcap, \boxplus\}\end{array}$$

(Reached $\mathcal{U}^0$)

$$\frac{q_c \models (\phi_{2j_0+1} \,\mathcal{U}\, (\phi_{2j_0+2} \wedge \ldots \wedge \mathcal{U}\, (\phi_{2k})))}{q \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge (\phi_3 \,\mathcal{U}\, (\phi_4 \wedge \ldots \,\mathcal{U}\, \phi_{2k})))} \qquad \begin{array}{l}\delta(q) = (c, q'), \\ c \ \wedge \ \neg \bigvee(\mathcal{A} \setminus \{c\}) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ q_c \in \mathcal{D}(c)\end{array}$$

(Reached $\mathcal{U}_1^+$)

$$\frac{\begin{array}{l}q_{a_1} \models \varphi'_{2j_{11}+1,2j_{12}+1} \\ q_{a_2} \models \varphi'_{2j_{21}+1,2j_{22}+1} \\ \ldots \\ q_{a_l} \models \varphi^0_{2j_{l1}+1,2j_{l2}+2} \\ \ldots \\ q_{a_k} \models \varphi'_{2j_{k1}+1,2j_{k2}+1}\end{array}}{q \models \varphi^0_{1,2j_1+2}} \qquad \begin{array}{l}\delta(q) = (\circ A, q'), A = \{a_1, a_2, \ldots, a_l, \ldots, a_k\}, \\ \bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ q_{a_i} \in \mathcal{D}(a_i), 1 \le i \le k, \circ \in \{\sqcap, \boxplus\}, j_0 \le j_{i1} \le j_{i2} \le j_1 \\ \text{for all } 1 \le i \le k, \text{ such that } \mathbb{N}_{2j_{i1}+1,2j_{i2}+1} \cap \mathbb{N}_{2j_{m1}+1,2j_{m2}+1} \\ \text{contains no even natural for all } 1 \le i \le k, 1 \le m \le k, i \ne m, \text{ and} \\ \bigcup_{1 \le i \le k} \mathbb{N}_{2j_{i1}+1,2j_{i2}+1} = \mathbb{N}_{2j_0+1,2j_1+1}, \text{ and } j_{l2} = j_1\end{array}$$

(Reached $\mathcal{U}_2^+$)

$$\frac{\begin{array}{l}q_a \models (\phi_{2j_0+1} \,\mathcal{U}\, (\phi_{2j_0+2} \wedge \ldots \wedge \mathcal{U}\, (\phi_{2j_3} \wedge \Box \phi_{2j_3+1}))) \\ q' \models \phi_{2j_3+1} \,\mathcal{U}\, \ldots \,\mathcal{U}\, \phi_{2k}\end{array}}{q \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge (\phi_3 \,\mathcal{U}\, (\phi_4 \wedge \ldots \,\mathcal{U}\, \phi_{2k})))} \qquad \begin{array}{l}a \ \wedge \ \neg \bigvee(\mathcal{A} \setminus a) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1} \\ \delta(q) = (a, q'), q_a \in \mathcal{D}(a)\end{array}$$

(Call $\mathcal{U}$)

$$\frac{\begin{array}{l}q_{a_1} \models \varphi_{2j_{11}+1,2j_{12}+1} \\ q_{a_2} \models \varphi_{2j_{21}+1,2j_{22}+1} \\ \ldots \\ q_{a_k} \models \varphi_{2j_{k1}+1,2j_{k2}+1} \\ q' \models \varphi^0_{2j_1+1,2j_2+2}\end{array}}{q \models \varphi^0_{1,2j_2+2}} \qquad \begin{array}{l}(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ \delta(q) = (\sqcap A, q'), q_{a_i} \in \mathcal{D}(a_i) \text{ for } 1 \le i \le k, \\ A = \{a_1, a_2, \ldots, a_k\}, j_0 \le j_{i1} \le j_{i2} \le j_1 \text{ for all } 1 \le i \le k, \\ \text{such that } \mathbb{N}_{2j_{i1}+1,2j_{i2}+1} \cap \mathbb{N}_{2j_{m1}+1,2j_{m2}+1} \text{ contains no} \\ \text{even natural for all } 1 \le i \le k, 1 \le m \le k, i \ne m, \text{ and} \\ \bigcup_{1 \le i \le k} \mathbb{N}_{2j_{i1}+1,2j_{i2}+1} = \mathbb{N}_{2j_0+1,2j_1+1}\end{array}$$

(Fork $\mathcal{U}$)

$$\frac{\begin{array}{l}q_{a_1} \models \varphi'_{2j_{11}+1,2j_{12}+1} \\ q_{a_2} \models \varphi'_{2j_{21}+1,2j_{22}+1} \\ \ldots \\ q_{a_l} \models \varphi_{2j_{l1}+1,2j_{l2}+1} \\ \ldots \\ q_{a_k} \models \varphi'_{2j_{k1}+1,2j_{k2}+1} \\ q' \models \varphi^0_{2j_1+1,2j_2+2}\end{array}}{q \models \varphi^0_{1,2j_2+2}} \qquad \begin{array}{l}(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \le i \le j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \\ \delta(q) = (\boxplus A, q'), q_{a_i} \in \mathcal{D}(a_i), 1 \le i \le k, \\ A = \{a_1, a_2, \ldots, a_l, \ldots, a_k\}, j_0 \le j_{i1} \le j_{i2} \le j_1 \text{ for all } 1 \le i \le k, \\ \text{such that } \mathbb{N}_{2j_{i1}+1,2j_{i2}+1} \cap \mathbb{N}_{2j_{m1}+1,2j_{m2}+1} \text{ contains no} \\ \text{even natural for all } 1 \le i \le k, 1 \le m \le k, i \ne m, \text{ and} \\ \bigcup_{1 \le i \le k} \mathbb{N}_{2j_{i1}+1,2j_{i2}+1} = \mathbb{N}_{2j_0+1,2j_1+1}\end{array}$$

(Fork-Choice $\mathcal{U}$)

Figure 1. Main set of proof rules for specification checking

$$\frac{}{q \models \mathrm{T}} \quad (\mathrm{T})$$

$$\frac{}{q \models a} \quad \begin{array}{l}\delta(q) = (a, q') \ \vee \ (\delta(q) = \circ A \ \wedge \ a \in A),\\ \circ \in \{\sqcap, \boxplus\}\end{array} \quad (a)$$

$$\frac{}{q \models \neg a} \quad \begin{array}{l}\delta(q) \neq (a, q') \ \wedge \ \neg(\delta(q) = \circ A \ \wedge \ a \in A),\\ \circ \in \{\sqcap, \boxplus\}\end{array} \quad (\neg a)$$

$$\frac{q \models \phi_1 \quad q \models \phi_2 \,\mathcal{U}\, \varphi}{q \models \phi_1 \wedge (\phi_2 \,\mathcal{U}\, \varphi)} \quad (\wedge^1)$$

$$\frac{q \models \phi_1 \quad q \models \phi_2}{q \models \phi_1 \wedge \phi_2} \quad (\wedge^2)$$

$$\frac{q \models \phi_1}{q \models \phi_1 \vee \phi_2} \quad (\vee)$$

$$\frac{}{\bot \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge \ldots \,\mathcal{U}\, (\phi_{2j} \wedge \Box\phi_{2j+1}))} \quad \begin{array}{l}\phi_{2i} = \bigwedge_{a \in A_i}(\neg a) \text{ where } A_i \subseteq \mathcal{A}, 1 \leq i \leq j,\\ \phi_{2j+1} = \bigwedge_{a \in A_{2j+1}}(\neg a) \text{ where } A_{2j+1} \subseteq \mathcal{A}\end{array} \quad (\text{Return } \Box)$$

$$\frac{\begin{array}{l}q_a \models \phi_{2j_0+1} \,\mathcal{U}\, (\phi_{2j_0+2} \wedge \ldots \,\mathcal{U}\, (\phi_{2j_1} \wedge \Box\phi_{2j_1+1}))\\ q' \models \phi_{2j_1+1} \,\mathcal{U}\, (\phi_{2j_1+2} \wedge \ldots \,\mathcal{U}\, (\phi_{2j_2} \wedge \Box\phi_{2j_2+1}))\end{array}}{q \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge \ldots \,\mathcal{U}\, (\phi_{2j_2} \wedge \Box\phi_{2j_2+1}))} \quad \begin{array}{l}(a \ \wedge \neg \bigvee(\mathcal{A} \setminus a)) \Rightarrow (\bigwedge_{0 \leq i \leq j_0} \phi_{2i}) \wedge \phi_{2j_0+1},\\ \delta(q) = (a, q'), q_a \in \mathcal{D}(a)\end{array} \quad (\text{Call } \Box)$$

$$\frac{\begin{array}{l}q_{a_1} \models \varphi_{2j_{11}+1,2j_{12}+1}\\ q_{a_2} \models \varphi_{2j_{21}+1,2j_{22}+1}\\ \ldots\\ q_{a_k} \models \varphi_{2j_{k1}+1,2j_{k2}+1}\\ q' \models \varphi_{2j_1+1,2j_2+1}\end{array}}{q \models \varphi_{1,2j_2+1}} \quad \begin{array}{l}(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \leq i \leq j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \delta(q) = (\sqcap A, q'),\\ q_{a_i} \in \mathcal{D}(a_i), 1 \leq i \leq k, A = \{a_i \mid 1 \leq i \leq k\}, j_0 \leq j_{i1} \leq j_{i2} \leq j_1\\ \text{for all } 1 \leq i \leq k, \text{ such that } \mathbb{N}_{2j_{i1}+1,2j_{i2}+1} \cap \mathbb{N}_{2j_{m1}+1,2j_{m2}+1}\\ \text{contains no even natural for all } 1 \leq i \leq k, 1 \leq m \leq k, i \neq m, \text{ and}\\ \bigcup_{1 \leq i \leq k} \mathbb{N}_{2j_{i1}+1,2j_{i2}+1} = \mathbb{N}_{2j_0+1,2j_1+1}\end{array} \quad (\text{Fork } \Box)$$

$$\frac{\begin{array}{l}q_{a_1} \models \varphi'_{2j_{11}+1,2j_{12}+1}\\ q_{a_2} \models \varphi'_{2j_{21}+1,2j_{22}+1}\\ \ldots\\ q_{a_l} \models \varphi_{2j_{l1}+1,2j_{l2}+1}\\ \ldots\\ q_{a_k} \models \varphi'_{2j_{k1}+1,2j_{k2}+1}\\ q' \models \varphi_{2j_1+1,2j_2+1}\end{array}}{q \models \varphi_{1,2j_2+1}} \quad \begin{array}{l}(\bigwedge A \ \wedge \ \neg \bigvee(\mathcal{A} \setminus A)) \Rightarrow (\bigwedge_{0 \leq i \leq j_0} \phi_{2i}) \wedge \phi_{2j_0+1}, \delta(q) = (\boxplus A, q'),\\ q_{a_i} \in \mathcal{D}(a_i), 1 \leq i \leq k, A = \{a_1, a_2, \ldots, a_l, \ldots, a_k\}, j_0 \leq j_{i1} \leq j_{i2} \leq j_1\\ \text{for all } 1 \leq i \leq k, \text{ such that } \mathbb{N}_{2j_{i1}+1,2j_{i2}+1} \cap \mathbb{N}_{2j_{m1}+1,2j_{m2}+1}\\ \text{contains no even natural for all } 1 \leq i \leq k, 1 \leq m \leq k, i \neq m, \text{ and}\\ \bigcup_{1 \leq i \leq k} \mathbb{N}_{2j_{i1}+1,2j_{i2}+1} = \mathbb{N}_{2j_0+1,2j_1+1}\end{array} \quad (\text{Fork-Choice } \Box)$$

Figure 2. Auxiliary set of proof rules for specification checking

The system has the set of web methods $\mathcal{M}_{\texttt{Local}} = \{\texttt{BeginTransaction}, \texttt{ContinueTransaction}, \texttt{FindItems}, \texttt{BrowseNewItems}, \texttt{ProcessPayment}, \texttt{ShipItems}\}$. It uses the web methods provided by the Amazon.com ECS platform represented by the set $\mathcal{M}_{\texttt{Amazon}} = \{\texttt{ItemSearch}, \texttt{CartCreate}, \texttt{CartAdd}, \texttt{CartModify}, \texttt{CheckOut}\}$. The set of outcomes $\mathcal{O} = \{s\} \cup \texttt{ASIN} \cup \texttt{CARTID} \cup \texttt{CATID}$ is used to characterize web method invocations, where s denotes successful completion of a web method invocation, ASIN is the set of Amazon Standard Identification Numbers used to represent items for sale, CARTID is the set of Cart Identifiers used by the Amazon.com ECS platform to distinguish between the virtual shopping carts assigned to various online shopping customers, and CATID is the set of Category Identifiers used by the ECS platform to represent various categories of items, such as Books, Music, Movies, Garments, etc. The set of actions $\mathcal{A}$ is given by $\mathcal{A} = (\mathcal{M}_{\texttt{Local}} \cup \mathcal{M}_{\texttt{Amazon}} \cup F) \times \mathcal{O}$, where F is a set of fresh symbols.

The web-based sales system $\mathcal{P}$ is now defined formally as a web-service interface $\mathcal{P} = (\mathcal{D}, \mathcal{F})$, where $\mathcal{D} : \mathcal{A} \to 2^Q$ maps an action to a set of locations, which we denote below by writing an action in front of every location that the action is mapped to, and the protocol automaton $\mathcal{F} = (Q, \bot, \delta)$ is represented by giving the switch function δ of the automaton as a sequence of triples $q : (\mathit{term}, q')$.

$$
\begin{array}{l}
\mathcal{P} = \{ \\
\quad \langle \texttt{BeginTransaction}, s\rangle \mapsto \\
\qquad q_0 : (\langle \texttt{CartCreate}, c\rangle, q_1) \\
\qquad q_1 : (\langle \texttt{ContinueTransaction}, c\rangle, \bot) \\
\quad \langle \texttt{ContinueTransaction}, c\rangle \mapsto \\
\qquad q_2 : (\langle a, s\rangle \boxplus \langle b, s\rangle \boxplus \langle c, s\rangle \boxplus \langle d, s\rangle, \bot) \\
\quad \langle a, s\rangle \mapsto \\
\qquad q_3 : (\langle \texttt{BrowseNewItems}, c\rangle, \bot) \\
\quad \langle b, s\rangle \mapsto \\
\qquad q_4 : (\langle \texttt{CartModify}, c\rangle, \bot) \\
\quad \langle c, s\rangle \mapsto \\
\qquad q_5 : (\langle \texttt{CheckOut}, c\rangle, \bot) \\
\quad \langle d, s\rangle \mapsto \\
\qquad q_6 : (\langle \texttt{ContinueTransaction}, c\rangle, \bot) \\
\quad \langle \texttt{BrowseNewItems}, c\rangle \mapsto \\
\qquad q_7 : (\langle \texttt{FindItems}, s\rangle, \bot) \\
\quad \langle \texttt{BrowseNewItems}, c\rangle \mapsto \\
\qquad q_8 : (\langle \texttt{FindItems}, s\rangle, q_9) \\
\qquad q_9 : (\langle \texttt{CartAdd}, c\rangle, \bot) \\
\quad \langle \texttt{FindItems}, s\rangle \mapsto \\
\qquad q_{10} : (\langle \texttt{ItemSearch}, c1\rangle \boxplus \langle \texttt{ItemSearch}, c2\rangle, \bot) \\
\}
\end{array}
$$

The execution begins with the invocation of the web method BeginTransaction, which is implemented by the service using the web method CartCreate provided by the Amazon ECS, which creates a new shopping cart and returns the cart identifier c to the service $\mathcal{P}$. The service $\mathcal{P}$ then allows the user (the customer, i.e., the person using the service $\mathcal{P}$ to buy items online) to search for new items and optionally add them to the shopping cart, modify the contents of the shopping cart, and check out. The former three actions can be done in parallel: the user can open multiple browser windows, through which multiple search, cart-add, and cart-modify transactions can be run in parallel, all of which would be dealing with the multi-threaded shopping cart object provided by the ECS platform. When the web method CheckOut is invoked, Amazon.com atomically inspects the contents of the user's cart, charges the customer for the total price, and ships the items, completing the sale. Thus, note that in particular, the user pays for exactly those items that get shipped.

Now, let us consider a modification of the system above to account for a change in the business model of our web-based shop. Previously, the shop depended upon Amazon.com to take care of billing the customer and shipping products. Now, to establish a closer relationship with the customer, our web-based shop decides to implement billing and shipping themselves in their own local check-out method LocalCheckOut and helpers ComputePrice and ShipItems, and continue using the Amazon.com ECS platform for only its product search services and shopping cart implementation.

The modified system is $\mathcal{P}' = (\mathcal{D}', \mathcal{F}')$ with the following additional actions $\langle \texttt{LocalCheckOut}, c\rangle$, $\langle \texttt{ComputePrice}, \texttt{OK}\rangle$, and $\langle \texttt{ShipItems}, \texttt{OK}\rangle$, and the modified partial function $\mathcal{D}'$ is obtained from $\mathcal{D}$ by adding the mappings indicated below, and the modified protocol automaton $\mathcal{F}'$ obtained from $\mathcal{F}$ by replacing the invocation of $\langle \texttt{CheckOut}, c\rangle$ in $\mathcal{F}$ with that of $\langle \texttt{LocalCheckOut}, c\rangle$, and adding to $\mathcal{F}$ the locations and transitions represented as follows:

$$
\begin{array}{l}
\langle \texttt{LocalCheckOut}, c\rangle \mapsto \\
\quad q_{11} : (\langle \texttt{ComputePrice}, \texttt{OK}\rangle, q_{12}) \\
\quad q_{12} : (\langle \texttt{ShipItems}, \texttt{OK}\rangle, \bot) \\
\langle \texttt{ComputePrice}, \texttt{OK}\rangle \mapsto \bot \\
\langle \texttt{ShipItems}, \texttt{OK}\rangle \mapsto \bot
\end{array}
$$

This simple modification has indeed introduced a severe error in our application. To appreciate this, let us consider the specification $\psi = \langle \texttt{BeginTransaction}, s\rangle \not\rightsquigarrow (\texttt{T}\ \mathcal{U}\ (\langle \texttt{ComputePrice}, \texttt{OK}\rangle \wedge (\texttt{T}\ \mathcal{U}\ (\langle \texttt{CartAdd}, c\rangle \vee \langle \texttt{CartModify}, c\rangle) \wedge (\texttt{T}\ \mathcal{U}\ \langle \texttt{ShipItems}, \texttt{OK}\rangle))))$. Intuitively, this specification represents the question "Is an execution starting with the invocation of $\langle \texttt{BeginTransaction}, s\rangle$ possible on which a cart-add or cart-modify transaction is successfully completed after price computation has been successfully invoked on the cart

but before the items in the cart have been shipped?" Note that $\mathcal{P}' \not\models \psi$. Thus, it is possible for an execution trace to occur (involving a race condition), in which the customer is able to make one or more cart-add or cart-modify transaction(s) after the price has been computed, but before the items in the shopping cart have been shipped. On such an execution trace, the customer could pay for items that never get shipped, or vice versa; both cases involving incorrect behavior.

We note that the problem can be resolved by getting rid of some of the parallelism afforded by the original design. We modify the service above accordingly, and obtain the following web-service interface $\mathcal{P}'' = (\mathcal{D}'', \mathcal{F}'')$, where the modified partial function $\mathcal{D}''$ is obtained from $\mathcal{D}'$ by adding the mappings indicated below, and the modified protocol automaton $\mathcal{F}''$ obtained from $\mathcal{F}'$ by replacing the locations and transitions for $\langle a, s\rangle$, $\langle b, s\rangle$, $\langle c, s\rangle$, and $\langle d, s\rangle$ in $\mathcal{F}'$ with the following:

$\langle a, s\rangle \mapsto$
  $q_{13} : (\langle \texttt{BrowseNewItems}, c\rangle, q_{14})$
  $q_{14} : (\langle \texttt{ContinueTransaction}, c\rangle, \bot)$
$\langle a, s\rangle \mapsto$
  $q_{15} : (\langle \texttt{CartModify}, c\rangle, q_{14})$
$\langle a, s\rangle \mapsto$
  $q_{16} : (\langle \texttt{LocalCheckOut}, c\rangle, \bot)$
$\langle b, s\rangle \mapsto \bot$
$\langle c, s\rangle \mapsto \bot$
$\langle d, s\rangle \mapsto \bot$

We note that $\mathcal{P}'' \models \psi$. However, how would we know that the modified service $\mathcal{P}''$ can be safely substituted in place of the service $\mathcal{P}'$? To this end we can simply check substitutivity by asking the question if $\mathcal{P}'' \preccurlyeq \mathcal{P}'$. The answer turns out to be Yes, and we safely conclude that the final service is the desired result.

References

[1] R. Alur and P. Madhusudan. Visibly Pushdown Languages. In *Proc. STOC*, pages 202–211. ACM Press, 2004.

[2] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, M. Lenzerini, and M. Mecella. Modeling Data & Processes for Service Specifications in Colombo. In *Proc. EMOI-INTEROP*. CEUR-WS.org, 2005.

[3] A. Betin-Can and T. Bultan. Verifiable Web Services with Hierarchical Interfaces. In *Proc. ICWS*, pages 85–94. IEEE Computer Society Press, 2005.

[4] A. Betin-Can, T. Bultan, and X. Fu. Design for Verification for Asynchronously Communicating Web Services. In *Proc. WWW*, pages 750–759. ACM Press, 2005.

[5] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web Service Interfaces. In *Proc. WWW*, pages 148–159. ACM Press, 2005.

[6] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proc. WWW*, pages 403–410. ACM Press, 2003.

[7] T. Bultan, J. Su, and X. Fu. Analyzing Conversations of Web Services. *IEEE Internet Computing*, 10(1):18–25, 2006.

[8] M. J. Butler, C. A. R. Hoare, and C. Ferreira. A Trace Semantics for Long-Running Transactions. In *25 Years Communicating Sequential Processes*, pages 133–150. Springer, 2004.

[9] H. Davulcu, M. Kifer, and I. V. Ramakrishnan. CTR-S: A Logic for Specifying Contracts in Semantic Web Services. In *Proc. WWW*, pages 144–153. ACM Press, 2004.

[10] A. Deutsch, L. Sui, and V. Vianu. Specification and Verification of Data-Driven Web Services. In *Proc. PODS*, pages 71–82. ACM Press, 2004.

[11] R. Farahbod, U. Glässer, and M. Vajihollahi. A Formal Semantics for the Business Process Execution Language for Web Services. In *Proc. WSMDEIS*, pages 122–133. INSTICC Press, 2005.

[12] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility Verification for Web Service Choreography. In *Proc. ICWS*, pages 738–741. IEEE Computer Society Press, 2004.

[13] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. WWW*, pages 621–630. ACM Press, 2004.

[14] X. Fu, T. Bultan, and J. Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. *Theor. Comput. Sci.*, 328(1-2):19–37, 2004.

[15] R. Hamadi and B. Benatallah. A Petri Net-Based Model for Web Service Composition. In *Proc. ADC*, pages 191–200. Australian Computer Society, 2003.

[16] R. Kazhamiakin and M. Pistore. Analysis of Realizability Conditions for Web Service Choreographies. In *Proc. FORTE*, volume 4229 of *LNCS*, pages 61–76. Springer, 2006.

[17] R. Kazhamiakin and M. Pistore. Static Verification of Control and Data in Web Service Compositions. In *Proc. ICWS*, pages 83–90. IEEE Computer Society Press, 2006.

[18] R. Kazhamiakin, M. Pistore, and L. Santuari. Analysis of Communication Models in Web Service Compositions. In *Proc. WWW*, pages 267–276. ACM Press, 2006.

[19] C. Laneve and G. Zavattaro. Foundations of Web Transactions. In *Proc. FoSSaCS*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.

[20] M. Mazzara and I. Lanese. Towards a Unifying Theory for Web Services Composition. In *Proc. WS-FM*, volume 4184 of *LNCS*, pages 257–272. Springer, 2006.

[21] M. Solanki, A. Cau, and H. Zedan. Augmenting Semantic Web Service Descriptions with Compositional Specification. In *Proc. WWW*, pages 544–552. ACM Press, 2004.

[22] M. Solanki, A. Cau, and H. Zedan. ASDL: A Wide Spectrum Language for Designing Web Services. In *Proc. WWW*, pages 687–696. ACM Press, 2006.

[23] J. Zhang, J.-Y. Chung, C. K. Chang, and S. Kim. WS-Net: A Petri Net-Based Specification Model for Web Services. In *Proc. ICWS*, pages 420–427. IEEE Computer Society Press, 2004.