3-2010

Model-based methods for linking web service choreography and orchestration

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Jin Song DONG

Geguang PU

Tian Huat TAN

Citation

SUN, Jun; LIU, Yang; DONG, Jin Song; PU, Geguang; and TAN, Tian Huat. Model-based methods for linking web service choreography and orchestration. (2010). *Proceedings of the 17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3*. 166-175.
Available at: https://ink.library.smu.edu.sg/sis_research/5033

2010 Asia Pacific Software Engineering Conference

Model-based Methods for Linking Web Service Choreography and Orchestration

Jun Sun*, Yang Liu†, Jin Song Dong†, Geguang Pu‡ and Tian Huat Tan†
Singapore University of Technology and Design
sunjun@sutd.edu.sg
National University of Singapore
{*liuyang,dongjs,tianhuat*}*@comp.nus.edu.sg*
East China Normal University
ggpu@sei.ecnu.edu.cn

***Abstract*—In recent years, many Web service composition languages have been proposed. Web service *choreography* describes collaboration protocols of cooperating Web service participants from a global view. Web service *orchestration* describes collaboration of the Web services in predefined patterns based on local decision about their interactions with one another at the message/execution level. In this work, we present model-based methods to close the gap between the two views. Building on the strength of model checking techniques, Web service choreography and orchestration are verified against temporal properties or against each other (to show that they are consistent). Specialized optimization techniques are developed to handle large Web service models. Furthermore, we propose a method to mechanically synthesize a prototype Web service orchestration from choreography, by repairing the choreography if necessary and projecting relevant behaviors to each service provider.**

I. Overview

The Web services paradigm promises to enable rich, dynamic, and flexible interoperability of highly heterogeneous and distributed Web-based platforms. In recent years, many Web service composition languages have been proposed. There are two different viewpoints, and correspondingly two terms, in the area of Web service composition. Web service *choreography* is usually referred to Web service specification which describes collaboration protocols of cooperating Web service participants from a global point of view. An example is WS-CDL (short for Web Service Choreography Description Language [8]). Web service *orchestration* refers to Web service descriptions which take a local point of view. That is, an orchestration describes collaborations of the Web services in predefined patterns based on local decision about their interactions with one another at the message/execution level. A representative is WS-BPEL (short for Web Service Business Process Execution Language [19]), which models business processes by specifying the work flows of carrying out business transactions.

Informally, a choreography may be viewed as a contract among multiple corporations, i.e., a specification of requirements (which may not be executable). An orchestration is the composition of concrete services provided by each corporation who realizes the contract. The distinction between choreography and orchestration resembles the well studied distinction between sequence diagrams (which describes inter-object system interactions, taking a global view) and state machines (which may be used to describe intra-object state transitions, taking a local view). Likewise, there are two important problems to be addressed. One is the *verification* problem, i.e., to verify whether a choreography or an orchestration is correct with respect to critical system properties or whether they are consistent with each other. The latter means that the orchestration faithfully implements all and only what the contract states. The other one is the *synthesis* problem, i.e., to decide whether a choreography can be realized by any orchestration (refereed as implementable) and synthesize a prototype orchestration if possible.

The solutions to both problems are important in the development of Web services. Solving either problem is however highly non-trivial. Firstly and most importantly, choreography and orchestration are generally modeled in different languages/formalisms, and choreography models are even not executable, Hence, there is natural gap between the two views. To perform effective analysis on the two views, we need to bridge the gap. Secondly, ideally it is sufficient to verify a single Web service invocation which is independent of other service invocations. In reality, this is often not true because of physical constraints (see [11], like the number of Web service instances are bounded by the thread pool size of the underlying operating system). As a result, multiple service invocations must be verified as a whole. Because Web services are designed for potentially large number of users (who may invoke the services simultaneously), verifying Web services based on model checking techniques must cope with state space explosion due to concurrent service invocations. Lastly, synthesizing orchestration from choreography resembles the distributed synthesis problem (e.g., in the setting of sequence diagrams), which has been shown to be undecidable in general and in many restrictive settings [22]. Worse, synthesizing a distributed object system with the exact behaviors is impossible if there are implied scenarios [2]. Both results apply to Web

This research was partially supported by a grant "SRG ISTD 2010 001" from Singapore University of Technology and Design.

1530-1362/10 $26.00 © 2010 IEEE
DOI 10.1109/APSEC.2010.28

service choreography (see examples later).

In this work, we offer practical solutions to both problems using a model based approach. First of all, we propose formal languages for modeling choreography and orchestration respectively with formal operational semantics. This creates a unified semantics model for the two views, which allows communications between choreography and orchestration models. To make them practical, these languages cover many language constructs for Web service compositions (i.e., behavioral aspects of WS-CDL and WS-BPEL).

In order to verify Web services under physical constraints, on-the-fly model checking techniques are adopted and extended specially to handle multiple concurrent service invocations. Consistency between choreography and orchestration is verified by showing conformance relationship (i.e., trace inclusion) between the choreography and the orchestration. Based on the refinement checking [25], we develop a verification algorithm to support data communications between choreography and orchestration, which allows orchestration to drive the execution of (non-executable) choreography. It is further optimized for Web services.

In order to deal with undecidability of the synthesis problem, we adopt a scalable lightweight approach. We do not claim to solve the problem completely, instead, we present a practical way to avoid undecidability. That is, instead of semantically checking whether a choreography is distributively implementable or not, we apply static analysis (based on the syntax) to check whether the choreography satisfies certain sufficient condition for being implementable. If positive, a synthesis procedure is invoked to automatically generate an orchestration prototype. Otherwise, we go further by using a repairing process to generate an implementable choreography by inserting communications between service providers. The repaired choreography may provide hints on how to correct the original one. Lastly, our engineering efforts have realized the methods in a toolkit named WS@PAT (available at http://pat.comp.nus.edu.sg), which is a self-contained framework for Web service modeling, simulation, verification and synthesis.

The work is related to research on verifying or synthesizing Web services [14], [5], [4], particularly, the line of work by Foster *et al* presented in [12], [13], [11]. They proposed to apply model-based verification for Web services. Their approach is to build Finite State Processes (FSP) model from Web services and then apply verification techniques based on FSP to verify Web services. For instance, conformance between choreography and orchestration is verified by showing a bi-simulation relationship between the respective FSP models. In particular, they identified the model of resource constraint in Web service verification [11] and proposed to perform verification under resource constraints. In addition, they developed a tool named LTSA-WS [13] (and later WS-Engineer). Our work can also be categorized as model-based verification, and is similar to theirs. Our approach complements their works in a number of aspects. Firstly, our model is based on a modeling language which is specially designed for Web service composition with features like channel passing, shared variables/arrays, service invocation with service replication, etc. Secondly, our verification algorithms employ specialized optimizations for Web services verification, e.g., model reduction based on algebraic properties of the models, partial order reduction for orchestration with multiple local computational steps, etc. These optimizations allow us to handle large state space and potentially large Web services. Lastly, we study the synthesis problem and offer a lightweight and practical solution, which is related to the work presented in [7]. The synthesis approach in [4] generates high level behavior patterns from WSDL description, while our approach synthesizes implementation from WS-CDL design.

The conformance checking is also discussed in [21], [3], [1]. In [21], formalizations are provided for the two views and symbolic model checking is used for the conformance checking. In [3] the notion of conformance is defined by means of automata and is restricted only to compositions of two services. The work of [1] concentrates on checking that the choreography specification is respected by the implementing services at run time. The formalization is given in terms of Petri Nets. Compared with these approaches, we provide a unified semantic model for Web service composition with efficient verification algorithm.

This work is related to works on verifying WS-BPEL [9], [20] and WS-CDL [23] by translating to other formalisms and verifying using existing model checkers like Uppaal [9], Java Path Finder [23] or NuSMV [20]. Compared to them, we provide direct verification and dedicated optimizations for the Web services specification languages. Our approach follows the formalization of Web service and the discussion on Web service generation in [8] and [24]. Our modeling languages are inspired from the simple Web service languages used in [8], [24] (which are sufficient for theoretical discussion). However, in order to develop a useful tool, we extend the we extend them to cover larger subset of the language constructs for Web service compositions. For example, variables and channel messages are supported in our languages but abstracted out in [24], which makes the modeling of real-world systems much easier. One could argue that it is possible to model Web services using other process algebra like modeling language, like Promela, or MSC for choreography and FSM for orchestration. These proposals suffer from the disadvantages of the translation approach. For example, the translation from Web service model to target process algebra may not be optimal, and the reflection of the counterexample is also non-trivial. Additionally, specific verification may not supported is the existing tool. For instance, the SPIN model checker for Promela does not support refinement checking, hence it is not possible to check the Web service conformance. WS@PAT as a verifier

is related to tools on equivalence/refinement checking (or language containment checking), e.g., FDR. Motivated by the features of Web services, we extend the algorithms to check conformance relations with specialized optimizations.

II. Modeling

In this section, we present modeling languages which are expressive enough to capture all core features of Web service choreography and orchestration. There are two reasons for introducing intermediate modeling languages for Web services. First, heavy languages like WS-CDL or WS-BPEL are designed for machine consumption and therefore are lengthy and complicated in structure. Moreover, there are mismatches between WS-CDL and WS-BPEL. For instance, WS-CDL allows channel passing whereas WS-BPEL does not. The intermediate languages focus on the interactive behavioral aspect. The languages are developed based on previous works of formal models for WS-CDL and WS-BPEL [8], [24], [23]. Second, based on the intermediate languages and their semantic models (namely, labeled transition systems), our verification and synthesis approaches is not bound to one particular Web service language. For instance, newly proposed orchestration languages like Orc [18] is also supported in our tool. This is important because Web service languages evolve rapidly. Being based on intermediate languages allows us to quickly cope with new syntaxes or features (e.g., by tuning the preprocessing component).

A. Choreography: Syntax and Semantics

The following is the core syntax for modeling interactive behaviors of Web service choreography, e.g., in WS-CDL.

$$
\begin{array}{lll}
\mathcal{I} ::= \mathit{Stop} \mid \mathit{Skip} & & \text{– inaction and termination} \\
\quad \mid \mathit{svr}(A,B,\tilde{ch}) \rightarrow \mathcal{I} & & \text{– service invocation} \\
\quad \mid \mathit{ch}(A,B,\mathit{exp}) \rightarrow \mathcal{I} & & \text{– channel commmunication} \\
\quad \mid x := \mathit{exp};\ \mathcal{I} & & \text{– assignment} \\
\quad \mid \mathit{if}\ b\ \mathcal{I}\ \mathit{else}\ \mathcal{J} & & \text{– conditional} \\
\quad \mid \mathcal{I} \,\Box\, \mathcal{J} & & \text{– choice} \\
\quad \mid \mathcal{I} \,|||\, \mathcal{J} & & \text{– service interleaving} \\
\quad \mid \mathcal{I};\ \mathcal{J} & & \text{– sequential}
\end{array}
$$

In WS@PAT, we support user-defined data types and dynamic invocation of C# library and hence modeling data components of Web services are feasible. For simplicity, we skip details on data variables in this paper. Let $\mathcal{I}$ (short of *interaction*), $\mathcal{J}$ be terms of choreography. Let A, B range over Web service roles; *ch* range over communication channels; *svr* range over a set of pre-setup service invocation channels (refer to discussion later); $\tilde{ch}$ denote a sequence of channels; x range over variables; *exp* be an expression and b be a predicate over only the variables.

We assume that each role is associated with a set of local variables and there are no globally shared variables among roles. This is a reasonable assumption as each role (which is a service) may be realized in a remote computing device. Informally, $\mathit{svr}(A,B,\tilde{ch})$, where *svr* is pre-defined service invocation channel, states that role A invokes a service provided by role B through channel *svr*. A service invocation channel is the one that is registered with a service repository so that the service is subject for invocation. $\tilde{ch}$ is a sequence of session channels which are created for this service invocation only. Notice that because the same service shall be available all the time, service channel *svr* is reserved for service invocation only. $\mathit{ch}(A,B,\mathit{exp})$ where *ch* is a session channel states that role A sends the message *exp* to role B through channel *ch*.

$x := \mathit{exp}$ assigns the value of *exp* to variable x. Without loss of generality, we always require that the variables constituting *exp* and x must be associated with the same role. If b evaluates to true, $\mathit{if}\ b\ \mathcal{I}\ \mathit{else}\ \mathcal{J}$ behaves as $\mathcal{I}$, otherwise $\mathcal{J}$. Given a variable x (a condition b), we write $\mathit{role}(x)$ ($\mathit{role}(b)$) to denote the associated role. $\mathcal{I} \,\Box\, \mathcal{J}$ is an unconditional choice (i.e., choice of two unguarded working units in WS-CDL) between $\mathcal{I}$ and $\mathcal{J}$, depending on whichever executes first. $\mathcal{I} \,|||\, \mathcal{J}$ denotes two interactions running in parallel. Notice that there are no message communications between $\mathcal{I}$ and $\mathcal{J}$. Two choreographies executing in a sequential order is written as $\mathcal{I};\ \mathcal{J}$. We remark that recursion is supported by referencing a choreography name.

The syntax above is expressive enough to capture the core Web service choreography features. For instance, channel passing is supported as we are allowed to transfer a sequence of channels on service invocation. Fig. 1 presents a choreography of an online store. The choreography coordinates three roles (i.e., *Buyer*, *Seller* and *Shipper*) to complete a business transaction among two pre-defined services channel *B2S* and *S2H*. At line 1, the *Buyer* communicates with the *Seller* through service channel *B2S* to invoke its service. Channel *Bch* which is sent along the service invocation is to be used as a session channel for the session only. In the *Session*, the *Buyer* firstly sends a message *QuoteRequest* to the *Seller* through channel *Bch*. At line 2, the *Seller* responds with some quotation value x, which is a variable. Notice that in choreography, the value of x may be left unspecified at this point. At line 5, the *Seller* sends a message through the service channel *S2H* to invoke a shipping service. Notice that the channel *Bch* is passed onto the *Shipper* so that the shipper may contact the *Buyer* directly. At line 6, the *Shipper* sends delivery details to the *Buyer* and *Seller* through the respective channels. The rest is self-explanatory.

In this work, we focus on the operational semantics. Given a choreography model, a system configuration is a 2-tuple $(\mathcal{I}, V)$, where $\mathcal{I}$ is a choreography and V is a mapping from the variables to their values, i.e., from data variables to their valuations or from channel variables to channel instances. A transition is expressed in the form of $(\mathcal{I}, V) \xrightarrow{e} (\mathcal{I}', V')$. The transition rules are presented in Fig. 2. Rule *inv*1 captures service invocation, where event $\mathit{svr}!\tilde{ch}$ occurs. Afterwards, rule *inv*2 becomes applicable so that the service invoking

1. $BuySell() = B2S(Buyer, Seller, \{Bch\}) \rightarrow Session();$
2. $Session() = Bch(Buyer, Seller, QuoteRequest) \rightarrow Bch(Seller, Buyer, QuoteResponse.x) \rightarrow$
3. $\quad if\ (x <= 1000)\ \{$
4. $\quad\quad Bch(Buyer, Seller, QuoteAccept) \rightarrow Bch(Seller, Buyer, OrderConfirmation) \rightarrow$
5. $\quad\quad S2H(Seller, Shipper, \{Bch, Sch\}) \rightarrow$
6. $\quad\quad (Sch(Shipper, Seller, DeliveryDetails.y) \rightarrow Stop \ ||| \ Bch(Shipper, Buyer, DeliveryDetails.y) \rightarrow Stop)$
7. $\quad \}\ else\ \{\ Bch(Buyer, Seller, QuoteReject) \rightarrow Session() \ \square \ Bch(Buyer, Seller, Terminate) \rightarrow Stop\ \};$

Figure 1. A sample choreography

$$\frac{}{(svr(A,B,\tilde{ch}) \rightarrow \mathcal{I}, V) \overset{svr!\tilde{ch}}{\rightarrow} (svr?(B,\tilde{ch}) \rightarrow \mathcal{I} \ ||| \ svr(A,B,\tilde{ch}) \rightarrow \mathcal{I}, V)}\ [\ inv1\ ]$$

$$\frac{}{(svr?(B,\tilde{ch}) \rightarrow \mathcal{I}, V) \overset{svr?\tilde{ch}}{\rightarrow} (\mathcal{I}, V)}\ [\ inv2\ ] \qquad \frac{}{(ch(A,B,exp) \rightarrow \mathcal{I}, V) \overset{ch!v}{\rightarrow} (ch?(B,v) \rightarrow \mathcal{I}, V)}\ [\ ch1\ ]$$

$$\frac{}{(ch?(B,v) \rightarrow \mathcal{I}, V) \overset{ch?v}{\rightarrow} (\mathcal{I}, V)}\ [\ ch2\ ] \qquad \frac{eval(exp, V) = v}{(x := exp;\ \mathcal{I}, V) \overset{\tau}{\rightarrow} (\mathcal{I}, V' \oplus x \mapsto v)}\ [\ assign\ ]$$

$$\frac{(\mathcal{I}, V) \overset{e}{\rightarrow} (\mathcal{I}', V'), eval(b, V) = true}{(if\ b\ \mathcal{I}\ else\ \mathcal{J}, V) \overset{e}{\rightarrow} (\mathcal{I}, V')}\ [\ b1\ ] \qquad \frac{(\mathcal{J}, V) \overset{e}{\rightarrow} (\mathcal{J}', V'), eval(b, V) = false}{(if\ b\ \mathcal{I}\ else\ \mathcal{J}, V) \overset{e}{\rightarrow} (\mathcal{J}, V')}\ [\ b2\ ]$$

$$\frac{(\mathcal{I}, V) \overset{e}{\rightarrow} (\mathcal{I}', V')}{(\mathcal{I} \ \square \ \mathcal{J}, V) \overset{e}{\rightarrow} (\mathcal{I}', V')}\ [\ choice1\ ] \qquad \frac{(\mathcal{J}, V) \overset{e}{\rightarrow} (\mathcal{J}', V')}{(\mathcal{I} \ \square \ \mathcal{J}, V) \overset{e}{\rightarrow} (\mathcal{J}', V')}\ [\ choice2\ ] \qquad \frac{(\mathcal{I}, V) \overset{e}{\rightarrow} (\mathcal{I}', V')}{(\mathcal{I} \ ||| \ \mathcal{J}, V) \overset{e}{\rightarrow} (\mathcal{I}' \ ||| \ \mathcal{J}, V')}\ [\ inter1\ ]$$

$$\frac{(\mathcal{I}, V) \overset{e}{\rightarrow} (\mathcal{I}', V')}{(\mathcal{I} \ ||| \ \mathcal{J}, V) \overset{e}{\rightarrow} (\mathcal{I}' \ ||| \ \mathcal{J}, V')}\ [\ inter2\ ] \qquad \frac{(\mathcal{I}, V) \overset{e}{\rightarrow} (\mathcal{I}', V'), e \neq \checkmark}{(\mathcal{I};\ \mathcal{J}, V) \overset{e}{\rightarrow} (\mathcal{I}';\ \mathcal{J}, V')}\ [\ seq1\ ] \qquad \frac{(\mathcal{J}, V) \overset{\checkmark}{\rightarrow} (\mathcal{J}', V')}{(\mathcal{I};\ \mathcal{J}, V) \overset{\tau}{\rightarrow} (\mathcal{J}, V')}\ [\ seq2\ ]$$

Figure 2. Choreography structural operational semantics: where ✓ is the special event of termination

request is ready to be received. At the same time, a copy of the choreography is forked. This is because a service may be invoked multiple times, possibly simultaneously, by different service users and all service invocations must conform to the choreography. In fact, in the standard practice of Web services, a service is embodied by a shared channel in the form of URLs or URIs through which many users can throw their requests at any time. For instance, different processes acting as *Buyer*s may invoke the service provided by the *Seller*. All *Buyer*s must follow the communication sequence. Furthermore, in order to match the reality, we assume that both service invocation and channel communication are asynchronous in this work. As a result, service invocation (or channel communication) is divided into two events, i.e, the event of issuing a service invocation (or channel output) and the event of receiving a service invocation (or channel input). This is captured by rules *inv*1, *inv*2, *ch*1 and *ch*2. For simplicity, we assume that a function *eval* returns the value of an expression *exp* given the valuation of variables V. Rule *assign* updates variable valuations. The rest of the rules resembles those for the classic CSP [15]. Notice that an assignment results in a invisible transition (written as τ). Only communication are visible.

Given a choreography $\mathcal{I}$, we build a Labeled Transition System (LTS) $(S, init, T)$ where S is the set of reachable configurations, *init* is the initial state (i.e., the initial choreography and the initial valuation of the variables) and T is a labeled transition relation defined by the semantics rules. A run of the LTS is a finite sequence of alternating configurations/events $\langle s_0, e_0, s_1, e_1, \cdots, e_{n-1}, s_n \rangle$ such that s_0 is *init* and $(s_i, e_i, s_{i+1}) \in T$ for all $i : 0..n$. A trace of $\mathcal{I}$ is a finite sequence of events $\langle e_0, e_1, \cdots, e_k \rangle$ if and only if there is a run of the LTS $\langle s_0, x_0, s_1, x_1, \cdots, x_{n-1}, s_n \rangle$ such that $\langle x_0, \cdots, x_{n-1} \rangle \upharpoonright \{\tau\} = \langle e_0, \cdots, e_k \rangle$ where $\upharpoonright$ is the filtering operation to remove all τ transitions (i.e., invisible events). The set of all finite traces of $\mathcal{I}$ is denoted as $traces(\mathcal{I})$.

In order to verify properties about the choreography, we use model checking techniques to explore all traces of the transition system. One complication is that the choreography's behavior may depend on environmental input which

is only known during runtime with the execution of an orchestration. For instance, the price quote provided by the *Seller* is unknown given only the choreography in Fig. 1. We discuss this issue in Section III.

B. Orchestration: Syntax and Semantics

$P ::= Stop \mid Skip$	– primitives
$\mid inv!\tilde{ch} \rightarrow P$	– service invoking
$\mid inv?\tilde{x} \rightarrow P$	– service being invoked
$\mid ch!exp \rightarrow P$	– channel output
$\mid ch?x \rightarrow P$	– channel input
$\mid x := exp;\ P$	– assignment
$\mid$ *if b P else Q*	– conditional branching
$\mid P \,\Box\, Q$	– orchestration choice
$\mid P \,\triangle\, Q$	– interrupt
$\mid P \,\|\|\|\, Q$	– interleaving
$\mid P;\ Q$	– sequential

A Web service orchestration $\mathcal{O}$ is composed of multiple roles, each of which is specified as an individual process defined using the syntax above. A slightly different syntax is used to build orchestration models. The reason is that orchestration takes a local view and therefore all primitive actions are associated with a single role. Let P and Q be the processes, which describe behaviors of a role.

Process $inv!\tilde{ch} \rightarrow P$ invokes a service (e.g., ¡invoke¿ in BPEL) through service channel *inv* and then behaves as specified by P. Or a service can be invoked by $inv?\tilde{x} \rightarrow P$ where $\tilde{x}$ is a sequence of channel variables which store the received channels. A process may send (receive) a message through a channel *ch* by $ch!exp \rightarrow P$ ($ch?x \rightarrow P$). Further, choice $\Box$ and interrupt $\triangle$ can be used to model event/exception handler in languages like BPEL, e.g. *LoginProgram* $\triangle$ *LoginExceptionHandler*. To match the reality, we always assume that the communication channels between different processes are asynchronous (and with a fixed buffer size) in this work. The rest are similar to those of choreography.

Similarly, we define the operational semantics. Let V_A be the valuation of the variables associated with the role A. Let C be a valuation function of the channels, which maps a channel to the sequence of items in the buffer. C is a set of tuples of the form $c \mapsto \tilde{msg}$. A configuration of the process is a 3-tuple (P, V_A, C). The firing rules are skipped for the sake of space. We remark that as in choreography, service invocation in orchestration forks a new copy of the service and thus allows potentially many concurrent service invocations. In reality, however, the number of overlapping service invocations is bounded by the maximum number of threads the underlying operating system allows [11]. In next section, we discuss how to capture this constraint and at the same time perform efficient verification.

Because an orchestration is the cooperation of multiple roles or processes, behaviors of the processes must be composed in order to obtain the global behavior. Assume that P plays the role A in the orchestration is written as $P@A$. Given two processes, e.g., P and Q, playing different roles, e.g., A and B, the composition is $P@A \parallel Q@B$. The semantic rules for process composition is straightforward, i.e., a global step is constituted of a local step by either P or Q. Following the rules, given an orchestration with multiple roles, each of which is specified as a process defined above, we may build a LTS. The executions of the orchestration equal to the executions the LTS. Similarly, we define traces of an orchestration as τ-filtered traces of the LTS. Given an orchestration $\mathcal{O}$, let $traces(\mathcal{O})$ be the set of finite executions.

Fig. 3 presents an orchestration which implements the choreography in Fig. 1. Each role is implemented as a separate component. Each component contains variable declarations (optional) and process definitions. We assume that the process *Main* defines the computational logic of the role after initialization. We remark that the orchestration generally contains more details than the choreography, e.g., the variable *counter* in *Buyer* constraints the number of attempts the buyer would try before giving up.

III. Verification

An orchestration can be verified against critical system properties like temporal properties or a choreography. We remark that service verification is performed under the physical constraints (e.g., a service may be blocked after the thread pool is full) in this work. In WS@PAT, we support full LTL formulae composed of propositions on data variables or events (e.g., a channel input/output, a local action, etc.). We adapt the automata-based on-the-fly approach to verify LTL formulae, i.e., by firstly translating a formula to a Büchi automaton and then check emptiness of the product of the system and the automaton. The details can be found in [26].

In the following, we define conformance between a choreography and an orchestration based on trace refinement and present an approach to verify it by showing refinement relationships. An orchestration $\mathcal{O}$ is valid w.r.t. a choreography $\mathcal{I}$ if and only if $\mathcal{O}$ refines $\mathcal{I}$i.e., $traces(\mathcal{O}) \subseteq traces(\mathcal{I})$. As discussed above, both choreography and orchestration can be translated into LTSs. By the assumption that both the ranges of the variables and sizes of channels are finite and the number of concurrent service invocations are bounded, the LTSs have finite number of states. As a result, we can extend the refinement checking algorithm proposed in [25] to do the conformance checking.

A main challenge for verifying practical Web services by model checking is state space explosion. There are multiple causes of state space explosion. Two of them are 1) the numerous different interleaving of processes executing concurrently in service orchestration and 2) the large number of concurrent service invocations. In the following, we discuss two optimization techniques which have been adopted to cope with the above issues.

$$
\begin{array}{ll}
\textit{Role Buyer } \{\textit{var counter} & = 0; \\
\quad \textit{Main}() & = \textit{B2S}!\{\textit{bch}\} \rightarrow \textit{Session}(); \\
\quad \textit{Session}() & = \textit{bch}!\textit{QuoteRequest} \rightarrow \textit{counter}{+}{+};\ \textit{bch}?\textit{QuoteResonse.x} \rightarrow \\
 & \quad \textit{if } (x <= 1000)\{\ \textit{bch}!\textit{QuoteAccept} \rightarrow \textit{bch}?\textit{OrderConfirmation} \rightarrow \textit{bch}?\textit{DeliveryDetails.y} \rightarrow \textit{Stop}\ \} \\
 & \quad \textit{elseif } (\textit{counter} > 3)\ \{\textit{bch}!\textit{QuoteReject} \rightarrow \textit{Session}()\}\ \textit{else}\ \{\textit{Stop}\};\ \ \} \\
\textit{Role Seller } \{\textit{var x} & = 1200; \\
\quad \textit{Main}() & = \textit{B2S}?\{\textit{ch}\} \rightarrow \textit{Session}(); \\
\quad \textit{Session}() & = \textit{ch}?\textit{QuoteRequest} \rightarrow \textit{ch}!\textit{QuoteResonse.x} \rightarrow (\textit{ch}?\textit{QuoteAccept} \rightarrow \textit{ch}!\textit{OrderConfirmation} \rightarrow \\
 & \quad \textit{S2H}!\{\textit{ch}, \textit{Sch}\} \rightarrow \textit{Sch}?\textit{DeliveryDetails.y} \rightarrow \textit{Stop} \,\Box\, \textit{ch}?\textit{QuoteReject} \rightarrow \textit{Session}());\ \ \} \\
\textit{Role Shipper } \{\textit{var detail} & = \text{“}20/10/2009\text{”}; \\
\quad \textit{Main}() & = \textit{S2H}?\{\textit{ch}1, \textit{ch}2\} \rightarrow (\textit{ch}1!\textit{DelieryDetails.detail} \rightarrow \textit{Stop} \,|||\, \textit{ch}2!\textit{DelieryDetails.detail} \rightarrow \textit{Stop});\ \ \}
\end{array}
$$

Figure 3. A simple orchestration

Firstly, the algorithm is improved with partial order reduction, to reduce the number of possible interleaving (particularly for orchestration). Events performed by single service role (e.g., local variable updates in service choreography or orchestration) are often independent with the rest of the system and hence are subject to reduction. During model checking, if a local action which results in a τ-transition is enabled (together with actions performed by other roles), we only expand the system graph using this action and postpone the rest. By this way, we build a smaller LTS and therefore checks deadlock-freeness or LTL more efficiently. For refinement checking, we apply this reduction in two ways. One is to apply partial order reduction separately to invisible events of either the choreography or the orchestration. Notice that this reduction is trace preserving and therefore is sound for refinement checking.

Secondly, by a simple argument, it can be shown that $|||$ is symmetric and associative. Naturally, different invocations of the same Web service are similar or even identical. By the above laws, the interleaving of multiple choreographes can be sorted (in certain fixed ordering) without changing the system behaviors. Therefore, if the choreography is in the form of $\mathcal{I} ||| \cdots ||| \mathcal{I} ||| \cdots$, it is equivalent whether the first $\mathcal{I}$ makes a transition or the second does. For verification of deadlock-freeness, safety or liveness properties, it is thus sound to pick one of the transitions and ignore the others. In general, this reduction could reduce the number of states up to the factor of $N!$ where N is the number of identical components. This reduction is inspired by research on model checking parameterized systems [17] and [10].

There are a number of other algebraic laws which may help to reduce the number of states (e.g., $\mathcal{I} \,\Box\, \mathcal{J} = \mathcal{J} \,\Box\, \mathcal{I}$). Nonetheless, it is a balance between the computational overhead (for the additional checking) and gain in state reduction. In our implementation (refer to Section V), a set of specially chosen algebraic laws are used to detect equivalence of system configurations.

IV. Prototype Synthesis

Given a choreography as a contract among multiple organizations, it is vital to guarantee that not only the contract is implementable but also it can be implemented in a non-ambiguous way. The synthesis problem of the classic sequence diagrams has been studied extensively [2], [6]. The negative results apply to the synthesis problem of Web service choreography. For instance, the following demonstrates the problem of implied scenarios in the setting of choreography. Assume that *ch* is an asynchronous session channel (with buffer size more than 2), A and B are two participating roles and M_1, M_2 are two messages.

$$
\begin{aligned}
\mathcal{I}_{exa} = &\ (\textit{ch}(A, B, M_1) \rightarrow \textit{ch}(B, A, M_2) \rightarrow \textit{Stop}) \\
&\ \Box\ (\textit{ch}(B, A, M_1) \rightarrow \textit{ch}(A, B, M_2) \rightarrow \textit{Stop})
\end{aligned}
$$

The specification states that either A sends M_1 to B first and then B responds by M_2, or the system works the other way around. Exactly as in the setting of sequence diagram [2], the above choreography $\mathcal{I}$ is not implementable because any distributed implementation would allow the following trace,

$$\langle \textit{ch}!M_1, \textit{ch}!M_2, \textit{ch}?M_2, \textit{ch}?M_1 \rangle$$

where $\textit{ch}!M_1$ is the event of (A) sending message M_1.

Exactly telling whether a choreography is implementable or synthesizing a minimally restrictive prototype is expensive. Therefore, we follow and extend the work presented in [8], to check whether the choreography satisfies a sufficient condition for implementability, by syntactic analysis. We check whether the choreography is *strongly connected* (or *well threaded* [8]), which intuitively means whether there is sufficient communication between the service roles so that the choreography is implementable. In general, there are choreographies which are not strongly connected but implementable. Nevertheless, strongly-connectedness remains a desirable property. If a choreography is strongly connected, a sound prototype orchestration may be generated by projecting the relevant behaviors to the respective role. If the choreography is not strongly connected, we then offer to repair the choreography to make it strongly connected.

$$
\begin{array}{lll}
\mathit{Stop} \triangleright X \;=\; \mathit{Skip} \triangleright X & = \mathit{Stop} & \\
(\mathit{svr}(A,B,\tilde{ch}) \rightarrow \mathcal{I}) \triangleright A & = \mathit{svr}!(A,\tilde{ch}) \rightarrow (\mathcal{I} \triangleright A) & \\
(\mathit{svr}(A,B,\tilde{ch}) \rightarrow \mathcal{I}) \triangleright B & = \mathit{svr}?(B,\tilde{ch}) \rightarrow (\mathcal{I} \triangleright B) & \\
(\mathit{svr}(A,B,\tilde{ch}) \rightarrow \mathcal{I}) \triangleright X & = \mathcal{I} \triangleright X & -\text{ if } X \notin \{A,B\} \\
(\mathit{ch}(A,B,\mathit{exp}) \rightarrow \mathcal{I}) \triangleright A & = \mathit{ch}!(A,\mathit{exp}) \rightarrow (\mathcal{I} \triangleright A) & \\
(\mathit{ch}(A,B,\mathit{exp}) \rightarrow \mathcal{I}) \triangleright B & = \mathit{ch}?(B,\mathit{exp}) \rightarrow (\mathcal{I} \triangleright B) & \\
(\mathit{ch}(A,B,\mathit{exp}) \rightarrow \mathcal{I}) \triangleright X & = \mathcal{I} \triangleright X & -\text{ if } X \notin \{A,B\} \\
(x := \mathit{exp};\ \mathcal{I}) \triangleright X & = x := \mathit{exp};\ (\mathcal{I} \triangleright X) & -\text{ if } X = \mathit{role}(x) \\
(x := \mathit{exp};\ \mathcal{I}) \triangleright X & = \mathcal{I} \triangleright X & -\text{ if } X \neq \mathit{role}(x) \\
(\mathit{if}\ b\ \mathcal{I}\ \mathit{else}\ \mathcal{J}) \triangleright X & = \mathit{if}\ b\ (\mathcal{I} \triangleright X)\ \mathit{else}\ (\mathcal{J} \triangleright X) & \\
 & & -\text{ if } X = \mathit{role}(b) \\
(\mathit{if}\ b\ \mathcal{I}\ \mathit{else}\ \mathcal{J}) \triangleright X & = (\mathcal{I} \triangleright X) \,\square\, (\mathcal{J} \triangleright X) & -\text{ if } X \neq \mathit{role}(b) \\
(\mathcal{I} \,\square\, \mathcal{J}) \triangleright X & = (\mathcal{I} \triangleright X) \,\square\, (\mathcal{J} \triangleright X) & \\
(\mathcal{I} \,|||\, \mathcal{J}) \triangleright X & = (\mathcal{I} \triangleright X) \,|||\, (\mathcal{J} \triangleright X) & \\
(\mathcal{I};\ \mathcal{J}) \triangleright X & = (\mathcal{I} \triangleright X);\ (\mathcal{J} \triangleright X) &
\end{array}
$$

Figure 4. Choreography to orchestration projection function

In the following, we present our approach in details. Firstly, we define a projection function which extracts relevant behaviors of a role from a choreography. Let $\mathcal{I}$ be a choreography. The projection of $\mathcal{I}$ onto role X is written as $\mathcal{I} \triangleright X$, which is defined by the rules presented in Fig. 4. We highlight that a conditional choice is projected to an unconditional choice if the condition is independent of variables associated with the role.

Ideally, given $A, B, \cdots, X$ as the roles $\mathcal{I}$, $\mathcal{I} \triangleright A \parallel \mathcal{I} \triangleright B \parallel \cdots \parallel \mathcal{I} \triangleright X$ shall be trace-equivalent to $\mathcal{I}$. This is not true for many reasons. For instance, assume that $\mathcal{I}$ is as follows,

$$\mathit{svr}(A,B,\tilde{ch}_1) \rightarrow \mathit{svr}(C,D,\tilde{ch}_2) \rightarrow \mathit{Stop}$$

It is easy to show that $\mathcal{I} \triangleright A \parallel \mathcal{I} \triangleright B \parallel \mathcal{I} \triangleright C \parallel \mathcal{I} \triangleright D$ allows more behaviors that $\mathcal{I}$ does. The reason is that the two interactions involve different roles and therefore it is impossible to ensure the global ordering without introducing extra communication. Another example is $\mathcal{I}_{exa}$, as shown above. In order to handle all choreographes and keep the synthesis algorithm simple, we take an alternative approach. We firstly define the sufficient conditions which guarantee the soundness of the projection and then discuss how to solve the problem if the conditions are not met.

Initiating roles Let $\mathcal{I}$ be a choreography. The set of initiating roles of $\mathcal{I}$, written as $\mathit{init}(\mathcal{I})$, is defined as follows.

$$
\begin{array}{ll}
\mathit{init}(\mathit{Stop}) & = \mathit{init}(\mathit{Skip}) = \varnothing \\
\mathit{init}(\mathit{svr}(A,B,\tilde{ch}) \rightarrow \mathcal{I}) & = \{A\} \\
\mathit{init}(\mathit{ch}(A,B,\mathit{exp}) \rightarrow \mathcal{I}) & = \{A\} \\
\mathit{init}(x := \mathit{exp};\ \mathcal{I}) & = \{\mathit{role}(x)\} \\
\mathit{init}(\mathit{if}\ b\ \mathcal{I}\ \mathit{else}\ \mathcal{J}) & = \{\mathit{role}(b)\} \\
\mathit{init}(\mathcal{I} \,\square\, \mathcal{J}) & = \mathit{init}(\mathcal{I}) \cup \mathit{init}(\mathcal{J}) \\
\mathit{init}(\mathcal{I} \,|||\, \mathcal{J}) & = \mathit{init}(\mathcal{I}) \cup \mathit{init}(\mathcal{J}) \\
\mathit{init}(\mathcal{I};\ \mathcal{J}) & = \mathit{init}(\mathcal{I})
\end{array}
$$

Similarly, we define the terminating roles of $\mathcal{I}$, written as $\mathit{term}(\mathcal{I})$, i.e., the roles participating in the last event of $\mathcal{I}$.

Strongly-connectedness Let $\mathcal{I}$ be a choreography. $\mathcal{I}$ is strongly connected if and only if it can be inductively deduced from the following rules,

- *Stop* and *Skip* are strongly connected.
- $\mathit{svr}(A,B,\tilde{ch}) \rightarrow \mathcal{I}$ is strongly connected if and only if $\mathit{init}(\mathcal{I}) = \{B\}$, and $\mathcal{I}$ is strongly connected.
- $\mathit{ch}(A,B,\mathit{exp}) \rightarrow \mathcal{I}$ is strongly connected if and only if $\mathit{init}(\mathcal{I}) = \{B\}$, and $\mathcal{I}$ is strongly connected.
- $x := \mathit{exp};\ \mathcal{I}$ is strongly connected if and only if $\{\mathit{role}(x)\} = \mathit{init}(\mathcal{I})$, and $\mathcal{I}$ is strongly connected.
- $\mathit{if}\ b\ \mathcal{I}\ \mathit{else}\ \mathcal{J}$ is strongly connected if and only if both $\mathcal{I}$ and $\mathcal{J}$ are strongly connected, and $\{\mathit{role}(b)\} = \mathit{init}(\mathcal{I}) = \mathit{init}(\mathcal{J})$.
- $\mathcal{I} \,\square\, \mathcal{J}$ is strongly connected if and only if $\mathit{init}(\mathcal{I}) = \mathit{init}(\mathcal{J})$, and both $\mathcal{I}$ and $\mathcal{J}$ are strongly connected.
- $\mathcal{I} \,|||\, \mathcal{J}$ is strongly connected if and only if both $\mathcal{I}$ and $\mathcal{J}$ are strongly connected.
- $\mathcal{I};\ \mathcal{J}$ is strongly connected if and only if both $\mathcal{I}$ and $\mathcal{J}$ are strongly connected and there exists role A such that $\{A\} = \mathit{term}(\mathcal{I}) = \mathit{init}(\mathcal{J})$.

Intuitively, a choreography is strongly connected if there is no "gap" between two consecutive statements. By definition, strongly connectedness can be checked syntactically and the complexity is linear in the size of the choreography. For instance, it is straightforward to verify that the choreography $\mathcal{I}_{exa}$ (presented above) is not strongly connected because the two choices have different initiating roles. The choreography presented in Fig. 1 is not strongly connected because of the "gap" between the first two messages.

$$\mathit{B2S}(\mathit{Buyer}, \mathit{Seller}, \{\mathit{Bch}\})$$
$$\mathit{Bch}(\mathit{Buyer}, \mathit{Seller}, \mathit{QuoteRequest})$$

The last role participated in the first message is *Seller*, whereas the initiating role of the second message is *Buyer*. We remark that if message sending/receiving is synchronous, then this choreography becomes "strongly connected". This can be repaired by adding an acknowledge message from *Seller* to *Buyer* in between.

Theorem 4.1: Let $\mathcal{I}$ be a choreography. Let $A, B, \cdots, X$ be the roles participating in $\mathcal{I}$. Let $\mathcal{O}$ be an orchestration such that $\mathcal{O} = \mathcal{I} \triangleright A \parallel \mathcal{I} \triangleright B \parallel \cdots \parallel \mathcal{I} \triangleright X$. If $\mathcal{I}$ is strongly connected, then $\mathit{traces}(\mathcal{O}) = \mathit{traces}(\mathcal{I})$. □

The theorem states that strongly-connectedness serves as a sufficient condition for the correctness of the projection function presented in Fig. 4. It can be proved by structural induction. We skip the proof in this paper.

Strongly-connectedness allows to apply fully automated synthesis in a straightforward way. Nonetheless, because it requires all messages must be *connected*, e.g., a message output must be followed by an acknowledgement. Choreography drafts may not often be strongly connected. It is

$$
\begin{array}{lll}
\mathcal{R}(Stop,S,E) & = Stop & \text{– if } I \text{ is } Stop;\\
\mathcal{R}(Skip,S,E) & = Skip & \text{– if } S = E;\\
\mathcal{R}(Skip,S,E) & = ch1(S,E,*) \rightarrow Skip & \text{– if } S \neq E;\\
\mathcal{R}(svr(A,B,\tilde{ch}) \rightarrow \mathcal{I},S,E) & = svr(A,B,\tilde{ch}) \rightarrow \mathcal{R}(\mathcal{I},B,E) & \text{– if } S = A;\\
\mathcal{R}(svr(A,B,\tilde{ch}) \rightarrow \mathcal{I},S,E) & = ch1(S,A,*) \rightarrow svr(A,B,\tilde{ch}) \rightarrow \mathcal{R}(\mathcal{I},B,E) & \text{– if } S \neq A;\\
\mathcal{R}(ch(A,B,exp) \rightarrow \mathcal{I},S,E) & = ch(A,B,exp) \rightarrow \mathcal{R}(\mathcal{I},B,E) & \text{– if } S = A;\\
\mathcal{R}(ch(A,B,exp) \rightarrow \mathcal{I},S,E) & = ch1(S,A,*) \rightarrow ch(A,B,exp) \rightarrow \mathcal{R}(\mathcal{I},B,E) & \text{– if } S \neq A;\\
\mathcal{R}(x := exp;\ \mathcal{I},S,E) & = x := exp;\ \mathcal{R}(\mathcal{I},S,E) & \text{– if } S = role(x);\\
\mathcal{R}(x := exp;\ \mathcal{I},S,E) & = ch1(S,role(x),*) \rightarrow x := exp;\ \mathcal{R}(\mathcal{I},S,E) & \text{– if } S \neq role(x);\\
\mathcal{R}(if\ b\ \mathcal{I}\ else\ \mathcal{J},S,E) & = if\ b\ \mathcal{R}(\mathcal{I},S,E)\ else\ \mathcal{R}(\mathcal{J},S,E) & \text{– if } S = role(b);\\
\mathcal{R}(if\ b\ \mathcal{I}\ else\ \mathcal{J},S,E) & = ch1(S,role(b),*) \rightarrow\ if\ b\ \mathcal{R}(\mathcal{I},S,E)\ else\ \mathcal{R}(\mathcal{J},S,E) & \text{– if } S \neq role(b);\\
\mathcal{R}(\mathcal{I} \,\square\, \mathcal{J},S,E) & = \mathcal{R}(\mathcal{I},S,E) \,\square\, \mathcal{R}(\mathcal{J},S,E) & \text{– if } S \in init(\mathcal{I} \,\square\, \mathcal{J})\\
\mathcal{R}(\mathcal{I} \,\square\, \mathcal{J},S,E) & = ch1(S,X,*) \rightarrow (\mathcal{R}(\mathcal{I},S,E) \,\square\, \mathcal{R}(\mathcal{J},S,E)) & \text{– if } S \notin init(\mathcal{I} \,\square\, \mathcal{J}) \text{ and } X \in init(\mathcal{I} \,\square\, \mathcal{J})\\
\mathcal{R}(\mathcal{I} \,|||\, \mathcal{J},S,E) & = \mathcal{R}(\mathcal{I},S,E) \,|||\, \mathcal{R}(\mathcal{J},S,E) & \text{– if } S \in init(\mathcal{I} \,|||\, \mathcal{J})\\
\mathcal{R}(\mathcal{I} \,|||\, \mathcal{J},S,E) & = ch1(S,X,*) \rightarrow (\mathcal{R}(\mathcal{I},S,E) \,|||\, \mathcal{R}(\mathcal{J},S,E)) & \text{– if } S \notin init(\mathcal{I} \,|||\, \mathcal{J}) \text{ and } X \in init(\mathcal{I} \,|||\, \mathcal{J})\\
\mathcal{R}(\mathcal{I};\ \mathcal{J},S,E) & = \mathcal{R}(\mathcal{I},S,X);\ \mathcal{R}(\mathcal{J},X,E) & \text{– } X \in init(\mathcal{J}).
\end{array}
$$

Figure 5. Choreography repair function

not very helpful if we simply claim a choreography is bad. Hence, we provide a method to automatically repair choreographies which are not strongly connected. The idea is to insert extra communications in order to fill the "gaps".

Let S (for starting role), E (for ending role) be two roles. Let $\mathcal{R}$ be the repairing function. Fig. 5 shows how $\mathcal{R}$ calculates a refined choreography in a compositional way. Notice that we assume $ch1$ is a channel between the sending role and receiving role and * is any message.

Theorem 4.2: Let $\mathcal{I}$ be an arbitrary choreography. Let *start* be a role in $init(\mathcal{I})$. Let *end* be a role in $term(\mathcal{I})$. $\mathcal{R}(\mathcal{I}, start, end)$ is strongly connected. □

The proof of the theorem is straightforward. By theorem 4.2, we may then apply the projection and generate a prototype orchestration. For instance, choreography $\mathcal{I}_{exa}$ will be modified as follows,

$$
\begin{array}{l}
(ch(B,A,*) \rightarrow ch(A,B,M_1) \rightarrow ch(B,A,M_2) \rightarrow Stop)\\
\square\ (ch(B,A,M_1) \rightarrow ch(A,B,M_2) \rightarrow Stop)
\end{array}
$$

The following orchestration can then be generated.

$$
\begin{array}{l}
((ch!* \rightarrow ch?M_1 \rightarrow ch!M_2 \rightarrow Stop)\\
\qquad \square\ (ch!M_1 \rightarrow ch?M_2 \rightarrow Stop))@B\\
\|\ ((ch?* \rightarrow ch!M1 \rightarrow ch?M_2 \rightarrow Stop)\\
\qquad \square\ (ch?M_1 \rightarrow ch!M_2 \rightarrow Stop))@A
\end{array}
$$

It can be shown that the generated orchestration above is equivalent to the repaired choreography. We remark it is wasteful to simply tell that a choreography is not implementable without telling how to correct it. *Our method gives the best effort to help users.* By comparing the repaired choreography and the original one, users are essentially presented why the original one is not implementable, and better, an easy way to correct it. In our toolkit, we offer other syntactic analysis as well, e.g., all kinds of well-formness. For instance, depending whether a channel is to be used once or multiple times (which can be specified in WS-CDL document), we can check whether a violation is possible during runtime.

In our formalism and Web-service composition languages such as WS-CDL, we may request the *service channel principle*. That is, service channels are intended to be repeatedly invocable and be always available to those who know the port names. Syntactically, this requires that $inv?\tilde{x}$ shall not be preceded in every processes. The synthesized service, however, may not satisfy this principle. We perform a simple checking and give a warming message if the generated orchestration violates the principle. It is our future work to identify the ways of generating orchestration which does satisfy the principle from a maximum set of choreographes.

V. Making It Mechanical

The methods discussed in previous sections have been realized in a toolkit named WS@PAT. WS@PAT is developed as a self-contained module in the PAT (Process Analysis Toolkit) framework, which is designed for supporting multiple domain specific modeling languages. WS@PAT has four main components, i.e., an editor with advanced editing features, a simulator which can be used to simulate the Web service models in different ways (e.g., interactive simulation, automated random simulation, generation of state graph, etc.), a verifier which integrates different model checking algorithms for different properties and a synthesizer which performs choreography repairing and orchestration generation.

WS@PAT has been applied to multiple case studies, including ones from *http://www.oracle.com/* and from [8], [24]. We are currently applying WS@PAT to several large WS-CDL and WS-BPEL models. Notice that our approach for synthesis is based on syntactic analysis and therefore

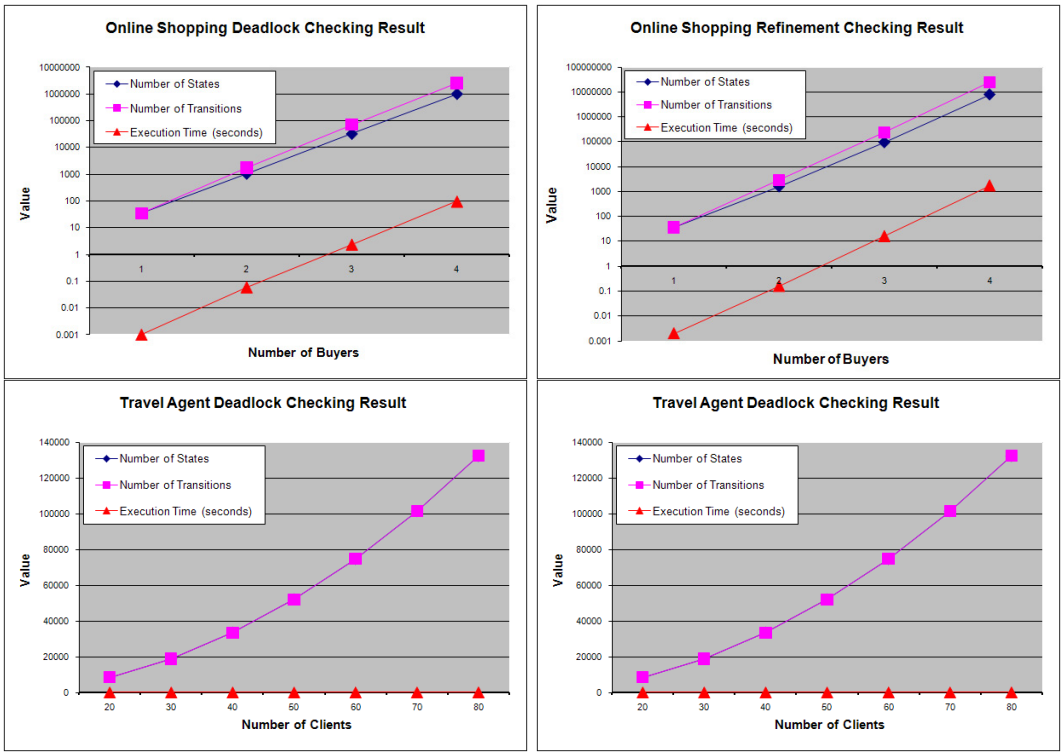

Figure 6. WS@PAT verification performance

scales up for large Web service models. We thus demonstrate the scalability of our verification approach, using two models. One is the online store example presented in Fig. 1 and Fig. 3. Instead of one buyer and one service invocation, we amend the model so that multiple users are allowed to use the services multiple times. The other is the service for travel arrangement. Its WS-BPEL model is available at *http://www.comp.nus.edu.sg/~pat/cdl/*. A WS-CDL specification is created manually. A number of clients invoke the business process, specifying the name of the employee, the destination, the departure date, and the return date. The BPEL process checks the employee travel status (through a Web service). Then it checks the prices for the flight ticket with multiple airlines (through Web services). Finally, the BPEL process selects the lowest price and returns the travel plan to each client.

Fig. 6 shows WS@PAT's efficiency using the two examples, obtained on a PC with Intel Q9500 CPU at 2.83GHz and 4GB RAM. Notice that in the experiments, we model the physical constraints as in [11] and verify the whole system instead of one service invocation. For both examples, we verify whether the orchestration is deadlock-free or not, by a reachability analysis searching for a deadlock state. In the online store example, we allow buyers to invoke the service repeatedly. As a result, the orchestration is deadlock-free. In the travel arrangement example, one client invokes the service only once. Because the number of concurrent service invocations is bound by the maximum number of threads allowed, the system reaches a deadlock state after exhausting all threads. This is consistent with the finding in [11]. In such case, WS@PAT is able to find a counterexample execution reasonably quickly with 80 clients using the service at the same time. We also verify that the orchestration conforms to the choreography using the refinement checking algorithm, as shown in Fig. 6. In both cases, the number of states and the time increase rapidly. Yet, WS@PAT is able to confirm that the orchestration conforms to the choreography with a few buyers/clients using the service concurrently.

In a nutshell, WS@PAT explores 10^8 states in a few hours, which suggests that WS@PAT is comparable to FDR [25] and SPIN [16] in terms of efficiency. WS@PAT shares many idea with WS-Engineer. To compare with WS-Engineer, we use Police Enquiry Obligations case study inside WS-Engineer. In the original example, the police officer will send request to a officer device, and following that, the officer device will enquire some items in sequence, e.g., nominal record, vehicle record, insurance record, ANPR record, DNA record, and then reply to the officer for the information. In order to make the example more challenging, we let the officer device enquire items in parallel. The size in the table above denotes the number of items to be retrieved in parallel. For each size, we provide two orchestrations, one conforms to the choreography (*Correct Police*), while the other one replies to the police officer before retrieving the items (*Wrong Police*).

The experiments data in the table below show that

WS@PAT is faster than WS-Engineer for *Correct Police* cases. When the conformance does not hold, WS@PAT stops immediately after the counterexample is detected, but not for WS-Engineer. We conclude that WS@PAT complements WS-Engineer for the orchestration synthesis, and still has competitive performance. The result, however, should be taken with a grain of salt, since both the input languages and verification algorithms are different.

Example		WS@PAT		WS-Engineer	
Name	**Size**	**Time**	**#States**	**Time**	**#States**
Correct Police	6	0.25	734	0.3	731
Wrong Police	6	0.02	3	0.4	731
Correct Police	7	0.93	2192	1.1	2189
Wrong Police	7	0.02	3	1	2189
Correct Police	8	3.96	6566	6.2	6563
Wrong Police	8	0.02	3	6.2	6563
Correct Police	9	15.57	19688	51.3	19685
Wrong Police	9	0.02	3	50.7	19685

VI. Conclusion

In this work, we presented model-based methods for automatic analysis of Web service compositions, in particular, linking two different views of Web services. Our methods built on the strength of advanced model checking techniques. We verified whether designs of Web services from two different views are consistent or not, by refinement checking with specialized optimizations. Furthermore, we offered a lightweight approach to tackle the synthesis problem and a toolkit to make the methods available.

References

[1] W. M. P. v. d. Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance checking of service behavior. *ACM Trans. Internet Technol.*, 8(3):1–30, 2008.

[2] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, 2003.

[3] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In *WS-FM'05*, pages 257–271, 2005.

[4] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic Synthesis of Behavior Protocols for Composable Web-Services. In *FSE'09*, pages 141–150, 2009.

[5] D. Bianculli, C. Ghezzi, and P. Spoletini. A Model Checking Approach to Verify BPEL4WS Workflows. In *SOCA'07*, pages 13–20, 2007.

[6] Y. Bontemps and P. Schobbens. The Complexity of Live Sequence Charts. In *FOSSACS'05*, pages 364–378, 2005.

[7] T. Bultan and X. Fu. Specification of Realizable Service Conversations Using Collaboration Diagrams. In *SOCA'07*, pages 122–132, 2007.

[8] M. Carbone, K. Honda, N. Yoshiba, R. Milner, G. Brown, and S. Ross-Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming. Technical report.

[9] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of Computation Orchestration Via Timed Automata. In *ICFEM'06*, pages 226–245. Springer, 2006.

[10] E. A. Emerson and V. Kahlon. Reducing Model Checking of the Many to the Few. In *CADE'00*, pages 236–254, London, UK, 2000. Springer-Verlag.

[11] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. S. Rosenblum, and S. Uchitel. Model Checking Service Compositions under Resource Constraints. In *FSE'07*, pages 225–234, 2007.

[12] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *ASE'03*, pages 152–163, 2003.

[13] H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: a Tool for Model-Based Verification of Web Service Compositions and Choreography. In *ICSE'06*, pages 771–774, 2006.

[14] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *WWW'04*, pages 621–630, 2004.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. International Series on Computer Science. Prentice-Hall, 1985.

[16] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engeering*, 23(5):279–295, 1997.

[17] C. N. Ip and D. L. Dill. Verifying Systems with Replicated Components in Murphi. In *CAV'96*, pages 147–158, 1996.

[18] M. Jayadev and C. William. Computation Orchestration: A Basis for Wide-area Computing. *Software and Systems Modeling (SoSyM)*, 6(1):83–110, March 2007.

[19] D. Jordan and J. Evdemon. Web Services Business Process Execution Language Version 2.0. http://www.oasis-open.org/specs/#wsbpelv2.0, Apr 2007.

[20] R. Kazhamiakin, P. K. Pandya, and M. Pistore. Representation, Verification, and Computation of Timed Properties in Web. In *ICWS'06*, pages 497–504, 2006.

[21] R. Kazhamiakin and M. Pistore. Choreography conformance analysis: Asynchronous communications and information alignment. In *WS-FM'06*, pages 227–241, 2006.

[22] A. Pnueli and R. Rosner. Distributed Reactive Systems are Hard to Synthesis. In *FOCS'90*, pages 746–757, 1990.

[23] G. Pu, J. Shi, Z. Wang, L. Jin, J. Liu, and J. He. The Validation and Verification of WSCDL. In *APSEC'07*, pages 81–88. IEEE Computer Society, 2007.

[24] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the Theoretical Foundation of Choreography. In *WWW'07*, pages 973–982, 2007.

[25] A. W. Roscoe. Model-checking CSP. *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378, 1994.

[26] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV'09*, pages 709–714. Springer, 2009.