A Distributed Framework for Reliable and Efficient Service Choreographies

Young Yoon, Chunyang Ye, Hans-Arno Jacobsen
Department of Electrical and Computer Engineering, University of Toronto
10 King's College Circle, Toronto, Ontario, M5S 3G4, Canada
{yoon, chunyang}@msrg.utoronto.ca, jacobsen@eecg.toronto.edu

ABSTRACT

In service-oriented architectures (SOA), independently developed Web services can be dynamically composed. However, the composition is prone to producing semantically conflicting interactions among the services. For example, in an interdepartmental business collaboration through Web services, the decision by the marketing department to clear out the inventory might be inconsistent with the decision by the operations department to increase production. Resolving semantic conflicts is challenging especially when services are loosely coupled and their interactions are not carefully governed. To address this problem, we propose a novel distributed service choreography framework. We deploy safety constraints to prevent conflicting behavior and enforce reliable and efficient service interactions via federated publish/subscribe messaging, along with strategic placement of distributed choreography agents and coordinators to minimize runtime overhead. Experimental results show that our framework prevents semantic conflicts with negligible overhead and scales better than a centralized approach by up to 60%.

Categories and Subject Descriptors

D.2.11 [**Software Architecture**]: Service-oriented Architecture (SOA)

General Terms

Algorithms, Performance, Reliability

Keywords

Service choreography, semantic conflict prevention, service interaction, service composition, publish/subscribe, event processing

1. INTRODUCTION

The service-oriented architecture (SOA) is an emerging software engineering paradigm for developing distributed collaborative enterprise applications [19, 28, 29]. Proponents of SOA argue that autonomously and independently developed Web services can seamlessly interact through message exchange to enable large distributed composite applications [9, 32]. The resulting interactions among collaborating services are referred to as a *service choreography* [30].

A choreography of distributed Web services, deployed across different organizations, is prone to producing semantically conflicting behavior [18, 21, 38]. For example, an interactive procurement

application may involve many autonomous participants such as requester services, supplier services and evaluation services from different organizations. The interactions of these autonomous services may generate unexpected behavior that violates the application semantics. For instance, the sending of an item initiated by the supplier service is not an intended application semantics, if the item was already canceled by the requester service. The occurrence of such events is referred to as a *semantic conflict* among collaborating Web services [21].

Semantic conflicts are unacceptable for SOA applications and detrimental to business operations; often they result in loss of revenue or intolerable consumer experiences. Thus, it is critical to control the interactions among services to prevent conflicting scenarios at runtime. Existing approaches address this issue by defining a global specification (*e.g.*, WS-CDL [37]) to coordinate the behavior of all participants [36, 11, 12, 22]. The global specification defines allowable interactions to safeguard the collaborating services from entering into conflicting situations. To implement these approaches, the global specification is decomposed into local specifications that are deployed to each service involved in the composition [36, 22]. Each local specification represents the allowable behavior for the corresponding participating service. By conforming to their local specification, the participating services collaboratively guarantee conflict-free interactions when acting in concert as part of the composition.

Although the aforementioned solution can be applied to control the local behavior of each individually participating service so that their global interactions do not violate the specified application-level semantics, the solution suffers from two major limitations.

First, a global specification about the behavior of interacting services may not always be available, especially in a dynamic service composition. In such a composition, the number of participating services is not fixed, services join or leave the composition freely, and services are anonymous to each other without global knowledge about the entire interaction. The characteristics of the dynamic composition make it difficult to design a complete global specification in advance [27]. For example, in emerging social commerce applications that form instant volume purchasing promotions, the number of buyer services is unknown in advance, and the strategies, among sellers that form a network dynamically, can change frequently [33].

Second, even if a global specification is available, the global specification may be non-realizable [1, 13], in the sense that there exist no local specifications that can implement and conform to the behavior defined by the global specification. Existing approaches address the non-realizability issue by extending the global specification with extra synchronization messages and coordinators [6, 31]. As shown in Section 5.4, the additional synchronization mes-

sages processed by a centralized coordinator for the extended global specification may introduce heavy end-to-end latency in the service composition, compromising its performance.

To address these limitations, we propose a novel distributed choreography framework for developing a semantic conflict-free choreography of interacting services in loosely coupled and dynamic service compositions. In our framework, no pre-defined and carefully crafted global specification is needed. Services can join or leave a service composition dynamically by connecting and disconnecting to access points of the distributed framework that forms a *service bus*. Through the choreography framework, services can interact with partners in a flexible manner by publishing messages to other services and by subscribing to messages of interest from partners. Moreover, instead of deploying a complete global specification to define the allowable interaction behavior among collaborating services, we model only the conflicting interactions as disallowable behavior and deploy them as safety constraints into the framework.

In order to avoid semantically conflicting behavior among interacting services, our framework proactively controls the interactions among collaborating services to keep the safety constraints satisfied. To reduce the extra overhead (*e.g.*, latency) introduced by the enforcement of safety constraints, we decompose the safety constraints into sub-constraints, and deploy them to specifically selected locations to best reduce latency in the distributed runtime environment. We establish that the satisfaction of the sub-constraints can guarantee semantic conflict-free interactions among collaborating services. Our approach has been fully implemented and our experimental evaluation shows that it exhibits better performance for service compositions than existing approaches.

The main contributions of this paper are four-fold: (1) we propose a novel framework to coordinate services that can dynamically join and leave a flexibly defined and loosely-coupled service composition; (2) for the composition, we model conflicting interactions among collaborating services and propose an algorithm to decompose safety constraints that prevent semantically conflicting behavior; (3) we develop an algorithm to optimize the deployment of the safety constraints to minimize the coordination overhead; and (4) we conduct extensive experiments to evaluate the benefits and characteristics of our solution.

The rest of this paper is organized as follows: Section 2 provides motivating examples to illustrate semantic conflicts. Section 3 refines our problem statement and analyses the research challenges. Section 4 introduces the design of our framework including: (1) the algorithm to decompose the safety constraints, along with the establishment of its correctness; and (2) the algorithm to optimize the deployment of safety constraints in the distributed runtime platform. Section 5 evaluates our framework empirically. Section 6 discusses the limitations of our current solution and suggests potential directions for improvements. Section 7 compares our work to the state-of-the-art.

2. MOTIVATING EXAMPLES

Semantically conflicting behavior in dynamically and loosely coupled service compositions can be found in many real world scenarios. For example, the following scenario was reported to us about an automobile manufacturer: Due to decreasing oil prices, demand for the company's hybrid vehicles dropped and sales lagged. As a strategic business decision, the manufacturer's headquarter ordered designated dealerships to clear out their inventory through a sales promotion. The promotion resulted in a temporary increase in sales, which triggered a notification to the assembly line to increase production of vehicles. Thus, the net effect of the order to clear out excess inventory was the creation of additional inventory.
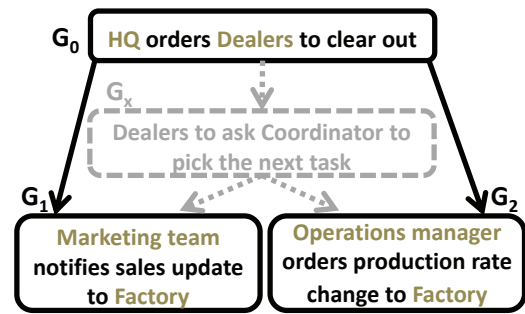

Figure 1: Hidden *pick* coordination task in choreography specification from a hybrid car manufacturing use case.
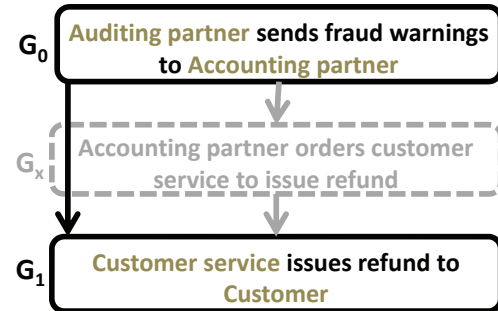

Figure 2: Implicit coordination between consecutive tasks from a credit card processing use case.

Suppose the collaboration was designed and implemented into executable processes using Web services that are independently developed for the partners as shown in Figure 1. G_0, for instance, is an interaction between the "HQ" service and the "Dealers" service. If there was no designated coordinator (as represented by the hidden task G_x) that governs and enforces a reliable *picking* of subsequent tasks given some business situation, *e.g.*, inventory clear-out, then the partners in the collaboration can behave arbitrarily, *i.e.*, marketing service notifying sales increase to the factory rather than having the operation management service order production decrease to the factory.

A further real world-inspired use case stems from a credit card processing scenario. In the example, the auditing department detects fraudulent transaction on some client's account and requests the accounting department to compensate the client by crediting the amount in question back to the client's account. However, at the same time, the client identifies the fraud on her account and demands immediate refund from the customer service department. The customer service department, not knowing the auditing department's action, processes the refund for the client as well. The net result for the client being a double credit.

Suppose this example was also implemented with Web services as shown in Figure 2. The implementation implicitly imposes the coordination between the accounting service partner and the customer service partner as shown in the task G_x such that the customer service can issue the refund only if it receives the refund order from the accounting service. The conflict in the use case was caused for the simple reason that the coordination was not actually implemented, therefore, there was nothing preventing the customer service to behave arbitrarily.

The original business logic specified in Figure 1 and Figure 2 is accurate from a business operations perspective. However, the specified interactions are agnostic to crucial details resulting from

the distributed runtime behavior that are not *explicitly* captured in the specification. The result is that the aforementioned semantic conflicts occur unintentionally.

The meticulous specification of constraints as safeguards against the conflicts in advance is infeasible, especially in a flexible environment that allows loose-coupling of Web services. The dynamic nature of loosely coupled Web service compositions, that is, anonymity and lack of a priori knowledge of the entire interaction, only exacerbates the problem. A passive and naïve solution to resolve the issue is to devise an error monitoring tool and apply an error recovery approach to restore the service composition from the conflicting scenarios. However, such a solution incurs unnecessary value loss for compensating conflicting interactions (*e.g.*, the charges incurred for re-sending the canceled products to the suppliers).

Intrigued by these examples and by the lack of a reasonable solution, we develop a framework to automatically extract the hidden safety constraints, formally articulated in Section 4. The framework enforces the constraints to prevent semantic conflicts in a service choreography.

3. PROBLEM AND CHALLENGES

In this section we discuss the need for mechanisms to enforce conflict-free interactions in service choreographies and outline the main challenges in designing and implementing these mechanisms.

As mentioned in Section 2, the choreography of autonomous services in a loosely-coupled and dynamic service composition is prone to producing conflicting interactions that violate the application semantics. Enabling a conflict-free choreography among dynamically composed services is a challenging task for three main reasons: (1) Services can participate in the choreography at any time and from any location; (2) the number of services is a priori unknown; and (3) participating services may only have a partial view of the entire collaboration and remain anonymous to each other.

In addition, preventing semantic conflicts in a choreography is challenging as well. Existing approaches that guarantee conflict-free interactions require a carefully crafted global specification in advance and can not handle the flexibility imposed by dynamic compositions, where no global specification exists a priori [27]. Instead, proactive exclusion of conflicting interactions from the choreography is a more desirable approach.

In a choreography, services tend to be distributed across organizations, which precludes the use of a centralized mechanism to monitor interactions for conflicts. Centralized coordination is inefficient, not scalable, and might be difficult to enforce administratively in distributed settings [20]. Therefore, a distributed coordination mechanism is needed. However, more importantly, the distributed mechanism of excluding the conflicting behavior may unintentionally overlook some conflicts due to the distributed nature of the service choreography.

For example, suppose a sample service choreography is given in Figure 3(a) where G_i identifies an interaction between collaborating partners, and any message interaction sequence, except m_1, m_2, m_4 or m_1, m_3, m_4, should be excluded. A distributed control mechanism is enabled in Figure 4 through a conventional decomposition method [40]. $\varphi(G_i)$ denotes a projection of G_i to a local task, L_{xy} for the collaborating partner r_x with y as the unique identifier. The control mechanism may overlook the violations of message ordering, as depicted in Figure 3(a). In fact, the control mechanism can produce disallowed message patterns, if nothing prevents r_3 from arbitrarily starting the task of sending m_2 before
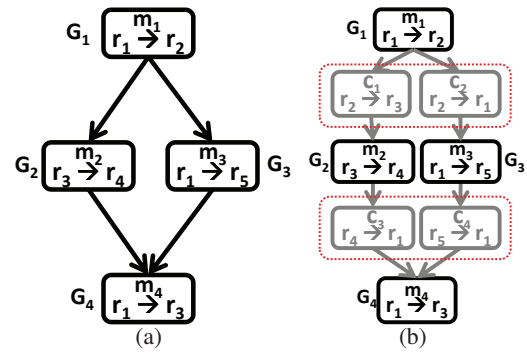

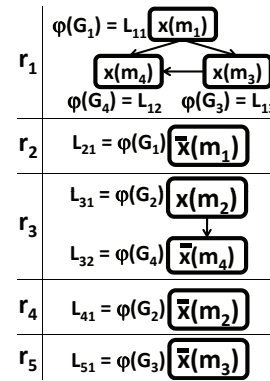**Figure 3: Implicit constraints in service choreography.**

Figure 4: Incorrect decomposition of constraints.

r_1 sends out m_1, as depicted in Figure 4. The ordering of the messages is clearly violated, rendering the safety constraint unsatisfied.

Another common type of semantic conflict that may be overlooked by a distributed control mechanism is a pick violation[1]. Given the example in Figure 3(a), once G_2 is picked, no message sequence other than m_2, m_4 should be excluded. However, r_1 in G_3 can still be triggered to send m_3, unless there is a mechanism to discover the hidden constraints in the pick activity to synchronize all the branches, as shown in Figure 3(b), *e.g.*, m_1 should strictly precede m_2 or m_3 by a confirmation message to be sent from r_2 of G_1 to r_3 of G_2. The safety constraints cannot be enforced correctly unless the hidden constraints are discovered and enforced.

Finally, reducing the overhead introduced by the enforcement of safety constraints in a service choreography constitutes a further challenge. Figure 3(b) shows that a distributed control mechanism to enforce safety constraints introduces extra synchronization messages c_1 and c_2 generated by the coordinator. This overhead is manifest, as increase in latency depends on the location of the coordinator in the distributed environment. Suppose the coordinator is placed in an organization to govern the pick between G_2 and G_3 in Figure 3(a). It is apparent that the farther the coordinator is from the senders of G_2 and G_3, the more time it will take to send the synchronization messages to the senders. Given a set of safety constraints and a number of distributed services, the deployment of the constraints into specifically selected locations to reduce overhead is a practical goal, not previously studied.

In the remainder of paper, we address all of these challenges through our novel choreography framework.

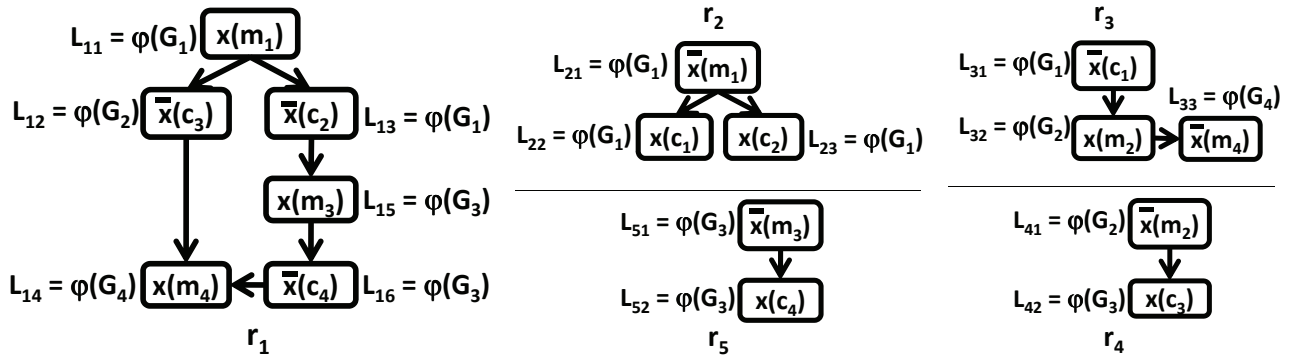[1] Pick and flow are *choice* operations specified in WS-CDL [37].

Figure 5: Example of reliable decomposition.

4. RELIABLE SERVICE CHOREOGRAPHY

In this section, we provide both abstract and practical solutions for a semantic conflict-free service choreography in a dynamic and loosely coupled service composition.

4.1 Overview

We coordinate service interactions by implementing service choreography on top of a distributed content-based publish/subscribe system [10]. The publish/subscribe system is comprised of a set of brokers interconnected to form a federated overlay to route messages from data producers to consumers [20]. As opposed to tightly-coupled, end-to-end delivery solutions, a publish/subscribe system offers efficient many-to-many communication patterns, full data source and sink decoupling, loose-coupling, message correlation capabilities, in-network content-based filtering capabilities, and location transparency [20].

Based on these capabilities, distributed services can easily interact with each other in a loosely-coupled manner by publishing messages to partner services and by subscribing to messages of interest from partner services. In this way, autonomous services can join or leave a composition dynamically by connecting or disconnecting to brokers in the publish/subscribe system at any time. However, The decomposition of safety constraints into sub-constraints, as mentioned in Section 3, may neglect to capture conflicting scenarios due to missing synchronization messages. To address this issue, our solution derives missing synchronization messages and automatically incorporates them explicitly into the decomposed sub-constraints.

To sketch out the solution, we revisit the sample safety constraint from Figure 3. The safety constraint is decomposed into sub-constraints for partner services r_1, - ,r_5. Note that the key difference between Figure 4 and Figure 5 is that coordination messages, c_i, where $i = 1, 2, 3, 4$, are derived and mapped to appropriate sub-constraints. For example, between G_1 and G_2, a coordination is necessary as the safety constraint implicitly states that r_3 should not send out message, m_2 *until* r_2 receives the message m_1 from r_2. In this particular example, every run of the service choreography, controlled by the decomposed set of sub-constraints with the introduction of explicit coordination among the sub-constraints, guarantees to exclude invalid message sequences that are implicitly specified, as in Figure 3(b). In this example, the message sequences other than m_1, c_1, m_2, c_3, m_4 or m_1, c_2, m_3, c_4, m_4 are invalid.

To minimize coordination overhead, we propose an algorithm to place coordinators, introduced by the decomposition of safety constraints, in the publish/subscribe system to best reduce latency for the service composition. The algorithm is designed based on the safety constraints and the broker overlay topology of the publish/subscribe system. It takes the aggregated weights of latency for a service composition introduced by extra coordinators into account. Then, the algorithm places coordinators at specifically selected brokers to best reduce the overall latency. In the next section, we turn our attention to the technique of safety constraints enforcement.

4.2 Enforcement of Safety Constraints

As mentioned earlier, we model conflicting interactions as disallowable behavior and enforce safety constraints to exclude them. Specifically, the safety constraints are decomposed into sub-constraints and deployed into the distributed runtime platform. In this section, we formalize the safety constraints and the decomposition algorithm to serve as a theoretical basis for our solution. We model the safety constraints as an *interaction process* among services.

Definition 1. An interaction process P is defined as follows:

$P \equiv (a_{int.} \cdot P) \,|\, (P \parallel P) \,|\, (P + P) \,|\, 0, a_{int.} \equiv r_1 \overset{m}{\Rightarrow} r_2,$

where $a_{int.}$ represents an interaction. *i.e.*, r_1 sends a message m to r_2. Operators '$\cdot$', '$\parallel$', and '+' formally represent the sequence, parallel, and choice operators in WS-CDL, respectively. The termination process is represented as 0.

Decomposing an interaction process into local processes to govern the behavior of each participating service may generate interaction that are different from what the original interaction process specifies. Formally, let P be an interaction process and LS_{r_i} represents the decomposed sub-constraints for a partner service, r_i, and there exists an interaction sequence $a_1, a_2, \cdots , a_k$ among the partner services. If the interaction sequence is the outcome of the composition of local interaction processes($LS_{r_1} \parallel \cdots \parallel LS_{r_m}$), but not the outcome of P, then the decomposition results in different behavior between the local processes and the original interaction process. Note that this difference is caused by the loss of synchronizations during the decomposition. Therefore, to correctly enforce the safety constraints, we need to explore and incorporate the missing synchronizations into the interaction processes. The following rules show how this is done:

Definition 2. Let P be an interaction process, and a transformation function, $P' \equiv \Omega(P)$, is defined recursively by the following rules:

1. $\Omega(a_{int.} \cdot b_{int.} \cdot P) = a_{int.} \cdot c_{int.} \cdot \Omega(b_{int.} \cdot P)$, where $a_{int.} \equiv r_1 \overset{m_1}{\Rightarrow} r_2, b_{int.} \equiv r_3 \overset{m_2}{\Rightarrow} r_4, r_2 \neq r_3, c_{int.} \equiv r_2 \overset{c_1}{\Rightarrow} r_3$.

2. $\Omega(a_{int.} \cdot b_{int.} \cdot P) = a_{int.} \cdot \Omega(b_{int.} \cdot P)$, where $a_{int.} \equiv r_1 \overset{m_1}{\Rightarrow} r_2, b_{int.} \equiv r_3 \overset{m_2}{\Rightarrow} r_4, r_2 = r_3$.

3. $\Omega(a_{int.} \cdot P_1 + b_{int.} \cdot P_2) = c_{int._1} \cdot \Omega(a_{int.} \cdot P_1) + c_{int._2} \cdot \Omega(b_{int.} \cdot P_2)$, where $a_{int.} \equiv r_1 \overset{m_1}{\Rightarrow} r_2, b_{int.} \equiv r_3 \overset{m_2}{\Rightarrow} r_4, r_1 \neq r_3,$

$c_{int._1} \equiv r_c \overset{c_1}{\Rightarrow} r_1, c_{int._2} \equiv r_c \overset{c_2}{\Rightarrow} r_3$.

4. $\Omega(a_{int.} \cdot P_1 + b_{int.} \cdot P_2) = \Omega(a_{int.} \cdot P_1) + \Omega(b_{int.} \cdot P_2)$, where $a_{int.} \equiv r_1 \overset{m_1}{\Rightarrow} r_2, b_{int.} \equiv r_3 \overset{m_2}{\Rightarrow} r_4, r_1 = r_3$.

5. $\Omega(P_1 \parallel P_2) = \Omega(P_1) \parallel \Omega(P_2)$.

6. $\Omega(a_{int.} \cdot 0) = a_{int.} \cdot 0$

7. $\Omega(0) = 0$.

Based on the rules in Definition 2, we can enrich a safety constraint with extra coordination messages and coordinators to make the decomposed sub-constraints consistent with the safety constraints. In particular, Rule 1 addresses the out-of-order messages by adding a synchronization message between two consecutive global tasks, (*i.e.*, interactions) where the receiver in the first one is different from the sender in the second one. Rule 3 addresses the invalid pick problem by adding a coordinator to coordinate the two branches, if the senders in the different branches belong to different partners. Rule 2 states that if the receiver in an interaction is also the sender in the immediately following interaction, then no synchronization is needed. Similarly, Rule 4 states that if the senders in the first interactions in two branches are from the same partner, then no synchronization is needed. In addition, Rule 5 states that the two interaction processes can be enriched individually in parallel. Rules 6 and 7 specify the termination conditions of enriching an interaction process. The following theorem guarantees the correctness of our approach.

Theorem 1. Given an interaction process P and $P' \equiv \Omega(P)$, then every interaction sequence of P' is also an interaction sequence of $LS_{r_1} \parallel \cdots \parallel LS_{r_m}$, and vice versa, where LS_{r_i} is the decomposed local process of P'.

The proof of Theorem 1 is based on the rules in Definition 2. The full proof is provided in our technical report [40].

In the next section, we design algorithms to implement these rules.

4.3 Reliable Decomposition Framework

Algorithm 1: Decomposition(G)

Input: Safety constraints $\mathbb{G}$,
Set of partners $\mathbb{R} = \{r_i | i = 1, 2, ...n\}$
Output: Set of local sub-constraints $\mathbb{LS}$
1   **if** *G is not decomposed* **then**
2   **if** *G is a successor of junction task* **then**
3   **foreach** $r \in \mathbb{R}$ **do**
4         $CurrentTask\_r$ = junction task;
5     $(LS\_currentSender)\_G$.AddTask(G.id(), $x(current\_message)$);
6     $(LS\_currentReceiver)\_G$.AddTask(G.id(), $\bar{x}(current\_message)$);
7   **foreach** *nextTask of G* **do**
8   **if** *(nextTask.sender $\neq$ currentReceiver) or (G is PICK)* **then**
9         $LS\_currentReceiver$.AddTask(G.id(), $x(c)$);
10        $LS\_(nextTask\_sender)$.AddTask(G.id(), $\bar{x}(c)$);
11   **foreach** *nextTask of G* **do**
12   **if** *nextTask is not decomposed* **then**
13   Decomposition(nextTask);
14   **else**
15         $LS\_(nextTask\_sender)$.AddTransition(nextTask.id());
16   **if** *currentSender = nextReceiver | currentReceiver = nextReceiver* **then**
17         $LS\_(nextTask\_receiver)$.AddTransition(nextTask.id());

In the previous section, we showed that the new coordination mechanism guarantees correct decomposition of safety constraints. In this section, we present techniques to implement the coordination mechanism.

Algorithm 1 determines the arrangement of coordination messages between partners. Suppose the two subsequent global tasks, G followed by G', are given. If the receiver of G is a different partner as the sender of G', then the algorithm must let the receiver of G

Algorithm 2: AddTask(id, L_{new})

Input: Sub-constraints $\mathbb{LS}$,
$Hash(k, List)$ where k = global task id, $Last\_k$
1   Hash.put(id, L_{new});
2   **if** L_{new} *is PICK or FLOW* **then**
3     $L_{last} = Hash.get(Last\_k).getFirstElement()$;
4   **else**
5     $L_{last} = Hash.get(Last\_k).getLastElement()$;
6   $Succ\_L_{last} \leftarrow Succ\_L_{last} \cup L_{new}.id()$;
7   $Prec\_L_{new} \leftarrow Prec\_L_{new} \cup L_{last}.id()$;
8   $Last\_k \leftarrow id$;

Algorithm 3: AddTransition(id)

Input: $Hash(k, List(L, L'))$ where k = global task id,
$Last\_k$
1   $L_{from} = Hash.get(id).getLastElement()$;
2   $L_{to} = Hash.get(Last\_k).getLastElement()$;
3   $Prec\_(L_{from}) \leftarrow Prec\_(L_{from}) \cup L_{to}.id()$;
4   $Succ\_(L_{from}) \leftarrow Succ\_(L_{from}) \cup L_{from}.id()$;
---

include a local task of sending out a coordination message toward the sender of G' into its sub-constraints as in (Algorithm 1: 6- 10). Unlike the conventional faulty decomposition methods, multiple local tasks can be produced by a single projection of a global interaction(Algorithm 1: 6- 7) for a sender and a receiver.

The mapping between the global and the local tasks is maintained in a hash table (Algorithm 2: Input). First, the local task of sending a coordination message makes a transition from the branching task (pick/flow) in the local process (Algorithm 3:2-3), if the local task is projected by a global branching task, as r_2 in Figure 5. When the decomposition framework explicitly requests (Algorithm 1:15-17) to set up a transition using AddTransition (Algorithm 3) from the task added by the previously projected global task ($Last\_k$), then there is a choice to be made to which element should the last added task make a transition to, because the mapping returns a list of local tasks. Our framework simply chooses the last element (Algorithm 3:2) and adds the transition by updating the pointers to immediately preceding ($Prec$) or succeeding ($Succ$) tasks (Algorithm 3:3-4).

The newly derived service choreography, as in Figure 5, always excludes the invalid message sequences thanks to the strict serialization through explicit arrangement of coordination among partners. Particularly for Figure 5, there is a globally unique initial send task (L_1 of r_1) while others are under the constraint to start to send a message only after receiving some message first.

4.4 Runtime Choreography Platform

We employ a platform that can support the execution of choreographies of independently developed Web services at global scale. To enable the exchange of large volumes of messages between Web services placed at geographically distributed locations, a scalable and efficient messaging substrate is essential. We adopt the federated content-based pub/sub messaging system, PADRES [10], already proven as distributed business process execution platform [20] and as management platform for SOA applications in large deployments [17, 24, 39]. We use the efficient and reliable routing mechanism of PADRES for choreography message exchange as shown in Figure 6. Each participating Web service is attached to a *choreography agent*. A choreography agent interprets and translates local processes into pub/sub constructs, *e.g.*, advertisements, subscriptions, and publications. Upon translation of the local processes, the agent advertises and subscribes to messages to be sent out and received, respectively. Once the initialization is complete, agents
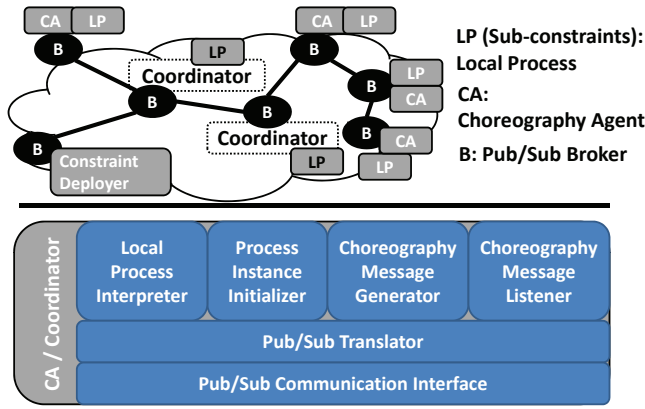
Figure 6: Choreography platform architecture.

Algorithm 4: Overhead($task$)

1   $overhead = 0$
    $List$
    if $task$ *is leaf* **then**
2   return $Overhead$ = runtime of $task$;
3   **else**
4      $Overhead$ += runtime of $task$;
      foreach $child$ *of* $task$ **do**
5   add $Overhead(child)$ to $List$;
6   return $Overhead+ = PlaceCoordinator(List)$;

Algorithm 5: PlaceCoordinator($List$)

1   $minimumOverhead = 0$
    $totalOverhead = 0$
    $bestBroker$
    foreach $broker$ *of* $Brokers$ **do**
2   **foreach** $task$ *of* $List$ **do**
3        $totalOverhead+ =$
        $Distance(task.sender, broker) * overhead$;
4   **if** $minumOverhead > totalOverhead$ **then**
5          $minimumOverhead = totalOverhead$;
          $bestBroker = broker$;
6   Assign a coordinators to $bestBroker$
7   return $minimumOverhead$;

---

step through the local processes by either publishing messages or listening (subscribing) to pending messages. Local processes are guaranteed to receive coordination messages as PADRES ensures reliable, duplicate-free, per-source ordering.

A difficulty arises for the *pick* task. Placing a single coordinator in a large messaging overlay is not desirable because it cannot efficiently handle multiple pick tasks issued concurrently at different locations in the overlay. We therefore opt to allocate multiple coordinators per pick task instead. The challenge is to determine a placement of coordinators in the messaging overlay that reduces coordination overhead. Our solution aims to minimize the messaging overhead involved with pick tasks. Given a list of services $r_1, r_2, \ldots, r_n$ that are potential recipients of the pick decision, the coordination overhead, L, is formally defined as

$$L = \sum_{i=1}^{n} w_{r_i} d_{r_i},$$

where r_i is a partner service, w_{r_i} is the weight expressed as an average time to execute the end-to-end local interaction process of r_i and d_{r_i} is a distance metric which is measured in terms of hop counts from r_i to the coordinator at the candidate location in the messaging overlay. The weight, w, is to give higher precedence to a task that contributes to the pick overhead over time, *e.g.*, a task with many subsequent tasks or with lengthy local processes. The weight can be more accurately computed by incorporating the pick probability distribution over time, *i.e.*, the task with higher probability to be picked obtains a higher weight.

A further issue is that there can be another pick task among the subsequent tasks of the process contributing to the overhead of the current pick task. Therefore, the placement of a pick coordinator should be pending subject to the placement of pick coordinators for subsequent pick tasks. To achieve this, our solution is based on recursive algorithms as presented in Algorithm 4, 5.

Revisiting the overall architecture of the platform in Figure 6, the coordinators are deployed to the overlay and follow the coordination processes just as the choreography agents follow their local processes.

5. EVALUATION

This section presents experimental evaluation of the algorithm for reliable decomposition of safety constraints and the efficiency of the runtime choreography platform. The decomposition algorithm was tested on a machine with Intel Core2 Duo 1.80GHz processors and 2GB of RAM. The runtime framework has been fully implemented on a cluster of IBM x3550 machines. The machines in the cluster communicate over a 1Gbps switched Ethernet connection, and each machine contains two Intel Xeon 5120 dual-core 1.86GHz processors and 4GB of RAM. The resulting decomposed sub-constraints are deployed to each partner that is randomly connected to one of the broker on a PADRES[2] messaging overlay.

Safety constraints for the service choreography are randomly generated as an interaction process in a directed graph. The processes vary in number of tasks, partners, and task executions. Each task specifies a pair of sender and receiver and a message to be exchanged. Every edge denotes a transition between two tasks. Given the set of partners whose initial task is to send a message, our framework randomly chooses an initial sender and triggers it at a constant rate to execute the sending task during every instance of the service choreography.

In the following subsections, we empirically assess how much the service choreography based on the *conventional* decomposition [40] is prone to producing message communication patterns that do not conform to the implicit and hidden safety constraints. Specifically, we measure the number of messages delivered out-of-order and the number of pick violations that reflect the degree of potential semantic conflicts. Then, we assess the benefits of the distributed coordination mechanism as oppose to centralized coordination in the runtime platform in terms of end-to-end latency of a choreography. Finally, we measure the overhead of the new approach to avoid potential semantic conflicts in terms of the number of coordination messages to be sent, and evaluate the runtime cost to execute the decomposition framework. Also, we measure the increase of local tasks along with the size of the rules our framework has to maintain in order to monitor invalid message patterns during runtime.

5.1 Assessment of Conflicts

Figure 7 and Figure 8 show minimum, maximum and average number of violations of the ordering and pick constraints. They indicate that the number of violations grows proportionally to the number of tasks and the number of executions. The number of violations is not affected by the number of peers, as the constraints on
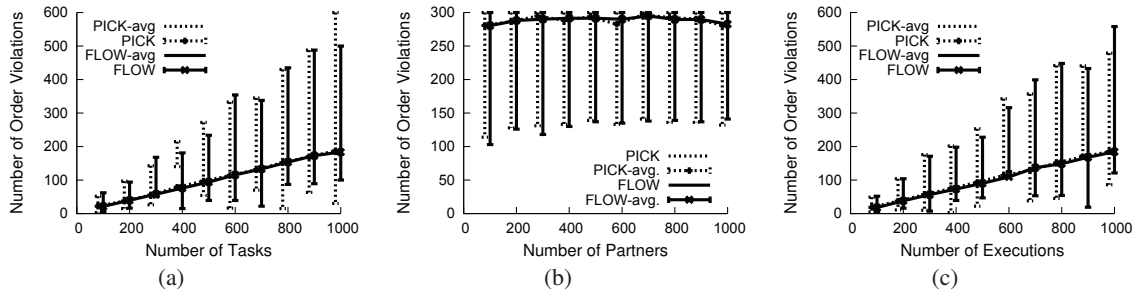
---

[2] http://msrg.org/projects/padres/
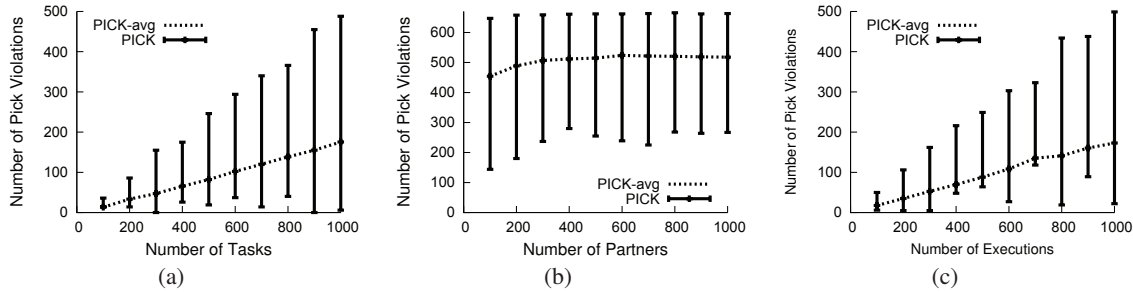
Figure 7: Number of ordering violations.

Figure 8: Number of pick violations.

the message communication sequences from the global perspective are agnostic of the participating peers, *i.e.*, no matter how many different partners are involved in the process, ordering requirements must be met. About 20% of 2,000 task executions violated the constraints. The results show high variance in the degree of the violations. Identifying the structural properties that make safety constraints prone to the violations is subject to future work. The fact that different partners contribute to the violations at different times and locations makes it a complicated and costly task to identify the root cause of the violation and to roll back the tasks already executed. The significant amount of violations reflects that detrimental damage already has been done to the collaboration through the unreliable service choreography. Figure 9(b) shows the size of the

5.2 Overhead of Decomposition

We measured the elapsed time to decompose safety constraints. Given a service choreography with 1,000 tasks and 10 partners, it required from 38% and 30% more time on average to decompose the process in a pick and a flow pattern, respectively, as shown on Figure 10. With the varying number of partners, up to 1,000, our decomposition method took 72% and 56% more time on average for a pick and a flow pattern, respectively. The general increase exhibited in our experiment is due to the additional arrangement of the coordination among partners. Yet, on average the elapsed time still remains within sub-milliseconds, thus our method does not cost significantly more than the conventional decomposition method, in terms of computation.
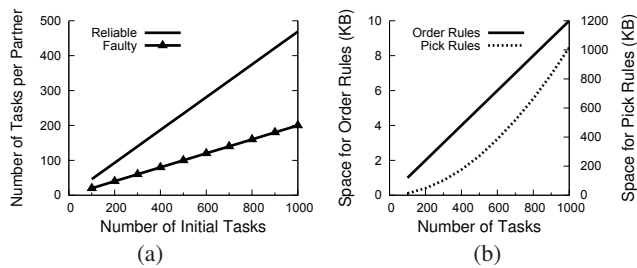

Figure 9: Coordination overhead.

Figure 10: Decomposition runtime.

rules that are used to detect conflicts by the broker in production. About 10KB and 1MB are needed for the rules to detect ordering and pick violations, respectively, given 1,000 tasks and 10 partners. Figure 9(a) shows that each partner may experience about 130% increase in the number of tasks in its sub-constraint, as the tasks of either sending or receiving coordination messages are newly added through the new decomposition method.
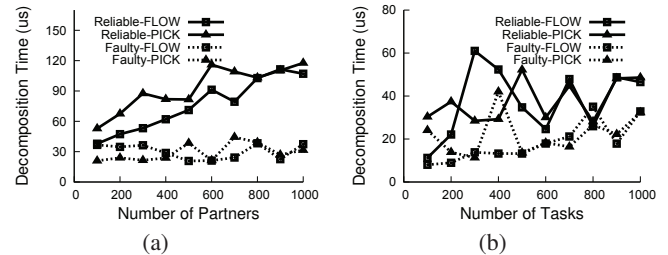
5.3 Overhead of Execution

The conflict-free service choreography is realized at the cost of strictly coordinating the messages exchanged among the partners when necessary as implied in the safety constraints. Figure 11 shows that the number of coordination messages increases with the number of tasks and the number of executions, both for the pick and
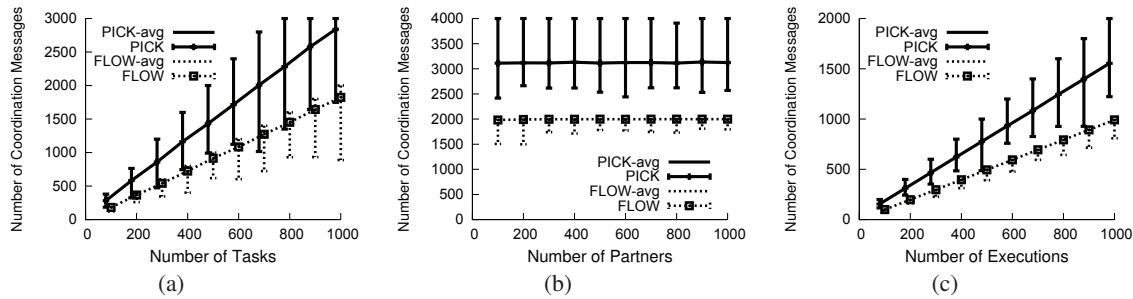
Figure 11: Number of coordination messages exchanged among partners.

flow patterns. Pick patterns yielded more coordination messages than the flow patterns, since not only a coordination message has to be exchanged between two consecutive tasks if necessary, but also a coordination message is required between the partner services and pick coordinators. In our experiment each control message simply consists of a 1 byte character to distinguish the type of message and a 4 byte integer to identify the message. Thus, for 2,000 executions, a total of only 14KB on average are used by the system. In practice, the control message does not have to be significantly larger than ours.
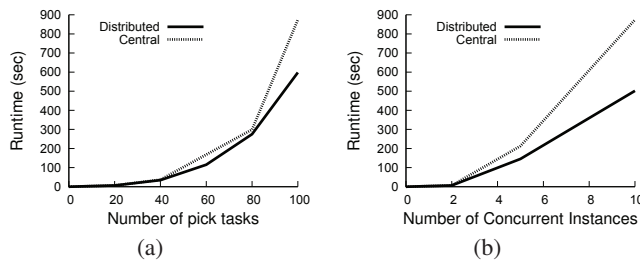
5.4 Runtime Latency

Figure 12: Latency in service choreography.

We measured the end-to-end latency of completing a set of choreographies with the varying number of pick coordination tasks and the varying degree of concurrency. An acyclic PADRES broker overlay with fanout of up to 2 is randomly generated on 15 nodes in our cluster. 20 unique services are randomly assigned to the nodes in the overlay. For benchmarking purpose, we tuned Algorithm 4, 5 to create coordination mechanism. Instead of assigning the broker at Algorithm 5:6, Algorithm 4 collects every intermediate overhead statement into a list which is iterated over to pick one coordination location in the overlay after the collection is completed.

The distributed coordination mechanism outperforms the centralized one from 2% to 14% with a single process instance with from 20 to 100 unique pick coordination tasks, respectively. The benefit of the distributed coordination mechanism becomes bigger when multiple process instances run concurrently (Figure 12(a)). With 10 independent choreography instances (each with 20 services and 20 picks) starting at the same time and running concurrently, the distributed coordination performed 60% better than the centralized in terms of the latency (Figure 12(b)). Therefore, the distributed coordination promises more scalable approach to host multiple choreographed processes.

6. DISCUSSION

Services that participate in the collaboration may re-locate frequently to cause coordination overhead to change. This may require adaptive re-computation of the the locations for the pick coordinators.

Note that our coordination mechanism is enabled per instance. As a future work, our framework can be extended to support coordination for interaction among multiple instances. An alternative approach to our coordination mechanism is to allow individual services to govern the pick operation. But this can lead to wrong pick decision making by a malicious service. Moreover, services may have affinity to particular locations, therefore, controlling the overhead of coordination such as latency can be less flexible.

In our solution, we enrich the safety constraints with extra coordination messages to avoid potential overlooked semantic conflicts. Such coordination messages may not exactly reflect the design intention of the choreography and thus restrict the behavior of services, but our solution discovers only the hidden and missing synchronizations among the service conversations. Choreography designers can check whether the hidden and missing synchronizations identified by our framework are design faults and choose to fix the problems using our recommended solution or other solutions.

In a more flexible environment, safety constraints can evolve and multiple versions of decomposed constraints can co-exit during runtime, which inevitably causes semantic conflicts. In order to avoid such a problem, management tools that can monitor the status of choreography instances and stop/resume them, if necessary, is required. We have already started investigating the adaptation of relevant workflow management standards [15].

7. RELATED WORK

In this section, our work is put in context of existing techniques in service composition, verification of service choreography and decomposition algorithms.

Coordination Framework for Dynamic Service Compositions. Dynamic composition of services is challenging but of benefit to changing business requirements [29]. Several approaches have been proposed to provide infrastructure support for dynamic composition of services [16]. For example, eFlow [5] is a framework supporting adaptive and dynamic composition of services. eFlow allows the business logic of a service composition to be modified dynamically. Bultan *et al.* proposed a theoretical framework to model service conversation specifications and studied realizability of service compositions [4]. Nezhad *et al.* designed a business conversation manager to support business conversations among people around best practices [26]. Brambilla *et al.* proposed a CASE tool together with a web engineering methodology to model and revise

processes in web applications. A global specification is assumed as a template in these approaches to guide the adaptation or coordination of service interactions. Our framework coordinates service interactions in a more flexible and loosely-coupled way, especially when a global specification is not available.

There also exist approaches based on message-oriented middleware to support service orchestration and choreography. For example, Li *et al.* proposed a framework to deploy a centralized BPEL process into a set of distributed execution engines on top of a pub/sub middleware [20]. Overbeek *et al.* combined SOA and event-driven architectures to deliver services in a pub/sub network [27]. Tai *et al.* proposed to use message-oriented middleware for service interactions [34]. In addition, some industrial Web service coordination frameworks and standards are also proposed to develop service-oriented applications. For example, WS-Coordination (WS-C)[3] is a framework that provides protocols to coordinate the activities of involved services in a service composition. Web Services Composite Application Framework (WS-CAF)[4] coordinates the activities and propagates the contexts among the involved services. The difference is that we applied a pub/sub middleware not only to route messages, but also to coordinate loosely coupled, flexible and dynamic service compositions. Moreover, we also deployed safety constraints into the distributed framework to prevent semantically conflicting interactions with acceptable processing overhead.

Resolving Mismatches for Service Choreography. Many research efforts have been devoted to constructing a choreography for independently developed services. To resolve mismatches between interacting services, Nezhad *et al.* proposed a semi-automated approach to adapt service interactions based on a constructed mismatch tree [23]. Benatallah *et al.* proposed a framework to develop adapters for service interoperability based on mismatch patterns [2]. Dumas *et al.* resolved mismatches between services by adapting their behavioral interfaces with algebra and visual notations [8]. These approaches can be applied to resolve semantic conflicts for interacting services by introducing adapters to bridge mismatches. However, these solutions are incapable of handling loosely coupled and dynamic service compositions, as the adapters designed to model mismatches may not be able to anticipate new mismatches introduced by dynamically joining services.

Doganata *et al.* suggested to detect compliance failures in business processes using automatic auditing tools [7]. Our work provides a more flexible coordination framework to resolve semantic conflicts among interacting services in dynamic service compositions. Instead of adapting the involved services, our framework enforces safety constraints to control interactions among services to prevent service compositions from entering into semantically conflicting situations.

Conformance Checking for Service Choreography. Another solution to prevent conflicts in service choreography is to select services that correctly implement a global specification. This requires to check the conformance between selected services and the global specification of the service composition. Aalst *et al.* [36] evaluated the conformance relationships between service behavior at runtime and its global specification in terms of two properties, that is, *fitness* and *appropriateness*. Montali *et al.* [22] described the global specification with a declarative language and verified the services' behavior based on logic. Fu *et al.* [12] used automata to verify service compositions with asynchronous communication. Foster *et al.* [11] verified the safety and liveness properties of service compositions based on process algebra. All these approaches are restricted to a service composition with a fully defined global specification. In a loosely coupled and dynamic service composition where the number of services is unknown and services can join or leave dynamically, a well planned global specification is not available in advance [27]. Therefore, our framework enforces safety constraints instead of a full global specification to prevent conflicts in loosely coupled and dynamic service interactions.

Realizability Issue. In service choreography, the local specification orchestrates the behavior of each service so that their interaction conforms to the global specification. An assumption is that the global specification should be realizable, in the sense that the global specification can be implemented by a set of distributed services. Verifying the realizability of a global specification is undecidable in general [1, 13]. Existing approaches enrich a global specification with additional synchronization messages and coordinators to render global specifications realizable [6, 31]. Our approach also adds synchronization messages and coordinators to guarantee that the distributed sub-constraints conform to the safety constraints that are defined to exclude conflicting scenarios. The difference between our work and the aforementioned work is that our work is to exclude disallowable behavior as opposed to specifying allowable behavior. Moreover, we also developed an algorithm to deploy sub-constraints to specifically selected locations in the distributed framework to reduce the enforcement overhead associated with safety constraints, which is not addressed in existing work.

Decomposition Algorithms. In a top-down service composition approach, service developers develop a global specification (*e.g.*, in WS-CDL [37]) and decompose it into local specifications. Services are then selected to implement the local specifications. Aalst *et al.* applied a Petri-net based approach to decompose global specifications into local public views with four kinds of consistency relationships [35]. Broy *et al.* proposed a formal model to split a service into sub-services by projection [3]. Giese *et al.* developed a modular approach to design, decompose, refine and verify complex systems [14]. Nanda *et al.* introduced an approach to partition a composite Web service written as a single BPEL program into an equivalent set of decentralized processes [25]. In our work, we also provided an algorithms to decompose safety constraints into a set of distributed sub-constraints. The difference is that our approach is applied to exclude semantically conflicting interactions in the framework whereas existing approaches are used to specify the allowable behavior of each involved service.

8. CONCLUSIONS

This paper introduces a novel distributed framework to coordinate semantic conflict-free interactions among autonomous services in a dynamic service composition. The framework allows services to interact with each other in a flexible and loosely-coupled way by publish/subscribe messaging. To prevent arbitrary interactions among services that violate application semantics, we provide algorithms to enforce safety constraints for dynamic service compositions. The algorithms decompose the safety constraints into sub-constraints which are deployed into the distributed runtime framework. We prove that these sub-constraints can collaboratively prevent semantic conflicting interactions among the participating services. We also develop an algorithm to place the sub-constraints to well selected locations in the distributed framework to reduce the enforcement overhead for safety constraints. According to our evaluation, the framework exhibits negligible overhead in enforcing the safety constraints and higher scalability than a centralized approach.

---

[3] http://docs.oasis-open.org/ws-tx/wscoor/

[4] http://www.oasis-open.org/committees/ws-caf/

9. REFERENCES

[1] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of msc graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005.

[2] B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani. Developing adapters for web services integration. In *CAiSE*, pages 415–429, 2005.

[3] M. Broy, I. H. Krüger, and M. Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1):5, 2007.

[4] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW '03*, pages 403–410, New York, NY, USA, 2003. ACM.

[5] F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Inf. Syst.*, 26(3):143–163, 2001.

[6] G. Decker, A. Barros, F. M. Kraft, and N. Lohmann. Non-desynchronizable service choreographies. In *ICSOC '08*, pages 331–346, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Y. N. Doganata and F. Curbera. Effect of using automated auditing tools on detecting compliance failures in unmanaged processes. In *BPM*, pages 310–326, 2009.

[8] M. Dumas, K. W. Wang, and M. L. Spork. Adapt or perish: Algebra and visual notation for service interface adaptation. 2006.

[9] S. Dustdar and W. Schreiner. A survey on web services composition. *Int. J. Web Grid Serv.*, 1(1):1–30, 2005.

[10] E. Fidler et al. Distributed publish/subscribe for workflow management. In *ICFI*, 2005.

[11] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *ASE '03*, pages 152–161, Los Alamitos, CA, USA, 2003.

[12] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *WWW '04*, pages 621–630, New York, NY, USA, 2004. ACM.

[13] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. Softw. Eng.*, 31(12):1042–1055, 2005.

[14] H. Giese, S. Burmester, W. Schäfer, and O. Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *SIGSOFT '04/FSE-12*, pages 179–188, New York, NY, USA, 2004. ACM.

[15] D. Hollingsworth. Workflow management coalition - the workflow reference model. Technical report, Workflow Management Coalition, Jan. 1995.

[16] R. Hull and J. Su. Tools for composite web services: a short overview. *SIGMOD Rec.*, 34(2):86–95, 2005.

[17] H.-A. Jacobsen and Muthusamy. BPM in cloud architectures: Business process management with SLAs and events. In *BPM '10*, Hoboken, NJ, USA, 2010.

[18] R. Kazhamiakin, A. Metzger, and M. Pistore. Towards correctness assurance in adaptive service-based applications. In *ServiceWave '08*, pages 25–37, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] D. Kuropka. What does service-oriented computing really mean? In *Service Oriented Computing*, 2005.

[20] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Trans. Web*, 4(1):1–33, 2010.

[21] L. T. Ly, S. Rinderle, and P. Dadam. Integration and verification of semantic constraints in adaptive process management systems. *Data Knowl. Eng.*, 64(1):3–23, 2008.

[22] M. Montali, M. Pesic, W. M. P. v. d. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative specification and verification of service choreographiess. *ACM Trans. Web*, 4(1):1–62, 2010.

[23] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *WWW '07*, pages 993–1002, New York, NY, USA, 2007. ACM.

[24] V. Muthusamy et al. SLA-driven business process management in SOA. In *CASCON '09*, pages 86–100, New York, NY, USA, 2009. ACM.

[25] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *OOPSLA '04*, pages 170–187, New York, NY, USA, 2004. ACM.

[26] H. R. M. Nezhad, S. Graupner, and S. Singhal. Business conversation manager: Facilitating people interactions in outsourcing service engagements. In *ICWE*, pages 468–481, 2010.

[27] S. Overbeek, B. Klievink, and M. Janssen. A flexible, event-driven, service-oriented architecture for orchestrating service delivery. *IEEE Intelligent Systems*, 24(5):31–41, 2009.

[28] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer. Service-oriented computing: A research roadmap. In *Service Oriented Computing*, 2005.

[29] M. P. Papazoglou, P. Traverso, I. Ricerca, and S. Tecnologica. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40:2007, 2007.

[30] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[31] G. Salaün and T. Bultan. Realizability of choreographies using process algebra encodings. In *IFM '09*, pages 167–182, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] B. Srivastava. Web service composition - current solutions and open problems. In *ICAPS '03*, pages 28–35, 2003.

[33] A. T. Stephen and O. Toubia. Deriving Value from Social Commerce Networks. *SSRN eLibrary*, 2009.

[34] S. Tai, T. A. Mikalsen, and I. Rouvellou. Using message-oriented middleware for reliable web services messaging. In *WES*, pages 89–104, 2003.

[35] W. M. P. van der Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theor. Comput. Sci.*, 270:125–203, January 2002.

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

[37] W3C. Web services choreography description language version 1.0, 2005. http://www.w3.org/TR/ws-cdl-10.

[38] I. Weber, J. Hoffmann, and J. Mendling. Beyond soundness: on the verification of semantic business process models. *Distrib. Parallel Databases*, 27(3):271–343, 2010.

[39] W. Yan, S. Hu, V. Muthusamy, H.-A. Jacobsen, and L. Zha. Efficient event-based resource discovery. In *DEBS '09*, pages 1–12, New York, NY, USA, 2009. ACM.

[40] Y. Yoon, C. Ye, and H.-A. Jacobsen. On semantics conflict in service choreography. In *Middleware Systems Research Group Technical Report*, Toronto, Canada, March 2010.