Service Adaptation with Probabilistic Partial Models*

Manman Chen[1], Tian Huat Tan[1] Jun Sun[1], Jingyi Wang[1], Yang Liu[2], Jing Sun[3], and Jin Song Dong[4]

[1] Singapore University of Technology and Design
[2] Nanyang Technological University
[3] The University of Auckland
[4] National University of Singapore

Abstract. Web service composition makes use of existing Web services to build complex business processes. Non-functional requirements are crucial for the Web service composition. In order to satisfy non-functional requirements when composing a Web service, one needs to rely on the estimated quality of the component services. However, estimation is seldom accurate especially in the dynamic environment. Hence, we propose a framework, ADFLOW, to monitor and adapt the workflow of the Web service composition when necessary to maximize its ability to satisfy the non-functional requirements automatically. To reduce the monitoring overhead, ADFLOW relies on asynchronous monitoring. ADFLOW has been implemented and the evaluation has shown the effectiveness and efficiency of our approach. Given a composite service, ADFLOW achieves 25% – 32% of average improvement in the conformance of non-functional requirements, and only incurs 1% – 3% of overhead with respect to the execution time.

1 Introduction

Service Oriented Architecture (SOA) is emerging as a methodology for building Web applications by using of existing Web services from different enterprises as components. Web services provide an affordable and adaptable framework that can produce a significantly lower cost of ownership for the enterprises over time. Web services make use of open standards, such as WSDL [8] and SOAP [14], which enable the interaction among heterogeneous applications.

The Web service composed by Web service composition is called a *composite service* (e.g., Travel Agency service) and the Web services that constitute the composite service are called *component services* (e.g., American Airline booking service). Non-functional requirements are an important class of requirements for Web services. They are concerned with quality of service (QoS) (e.g., response time, availability, cost) of Web services. The non-functional requirements are often an important clause in service-level agreements (SLAs), which is the contractual basis between service consumers and service providers on the expected QoS level. For example, nowadays, many big players in the market (e.g., Netflix, Amazon, and Microsoft Azure) have adopted microservice architecture [2]. It works by decomposing their existing monolithic applications into smaller, and highly decoupled services (also known as microservices). These microservices are then composed to fulfill their business requirements. For example, Netflix decomposed their monolithic DVD rental application into microservices that work together, and then stream digital entertainment to millions of Netflix customers every day.

* This work is supported by research project T2MOE1303.

In this work, the requirements of QoS for the composite service can be specified as *global constraints*. For example, an example of the global constraint is that the response time of the Web service composition must be less than 8 ms. To guarantee the SLAs between the Web service composition and its users, the design of Web service composition involves the estimation of QoS of component services. The QoS of component services could be solicited from the providers of component services either in the form of SLAs or based on past history of executions by making use of existing approaches (e.g., KAMI [9]).

However, due to the highly evolving and dynamic environment that the Web service composition is running, the design time assumptions for Web service composition, even if they are initially accurate, may later change during runtime. For example, the execution time of a component service may increase unexpectedly due to reasons such as network congestion, which could affect the response time of the composite service. Furthermore, at runtime, the non-functional properties of a composite service rely on the behaviors of component services offered by third-party partners. The distributed ownership makes the non-functional properties of Web service composition subject to changes. For instance, component service providers could modify existing component Web services, and usage profiles of the component Web services may change over time. These behaviors may result in potential violations of SLA of the composite Web service. Since estimations are seldom accurate, it is desirable that Web service compositions could *dynamically adapt* themselves to their environment with little or no human intervention in order to meet the guaranteed QoS levels. The loose coupling and binding features of SOA systems make them particularly suitable for runtime adaptation.

Existing works [5,17,18,15] address this problem by replacing component services or invoke component services adaptively, which we denote it as *point adaptation strategy*. *Point adaptation strategy* suffers several disadvantages. First, there are cases where such a strategy does not work. For example, there is no alternate service that can satisfy the non-functional requirements. In addition, there might not exist an alternating service that could be switched directly. Secondly, there maybe incur much cost as they may invoke another service to compensate it.

In this work, we propose the usage of *workflow adaptation strategy* to address this issue. A workflow adaptation strategy involves modifying the workflow to find a path for execution that can maximize the ability to satisfy the non-functional requirements. Therefore, we present *runtime ADaptation framework based on workFlow* (ADFLOW), a framework to alleviate the management problem of complex Web compositions that operate in rapidly changing environments. We propose the notion of *probabilistic partial model*, which is extended from the previous notion of partial model [11], to capture the *uncertainties* of system execution with probabilistic. The global constraints of the composite service are decomposed into local requirements for each state of a probabilistic partial model. When a possible violation of the global constraints is detected, adaptive actions are taken preemptively based on the probabilistic partial model, to avoid unsatisfactory behaviors or failures. In particular, the adaptive action chooses the execution that could maximize the likelihood of conformance of the global constraints.

Our contributions are summarized as follows.

1. We propose the probabilistic partial model to capture the runtime uncertainties of Web service composition.
2. We propose a runtime adaptation framework, ADFLOW for Web service composition. ADFLOW monitors the execution of Web service composition based on local requirements of the probabilistic partial model. If a possible violation of the global constraints of the composite service is detected, adaptive actions would be taken preemptively to prevent the violation.
3. To reduce the monitoring overhead, we propose to use *asynchronous monitoring* where the execution status is monitored asynchronously whenever possible. We show that this approach reduces the overhead significantly.
4. We have evaluated our method on real-world case studies, and we show that it significantly improves the chance of the composite service to conform to the global constraints.

Outline The rest of paper is structured as follows. Section 2 describes a motivating example. Section 3 introduces the probabilistic partial model used for Web service compositions. Section 4 presents our ADFLOW adaptation framework for runtime adaptation. Section 5 evaluates the performance of our approach in several scenarios with the increasing complexity. Section 6 discusses related work. Section 7 concludes the paper and describes future work.

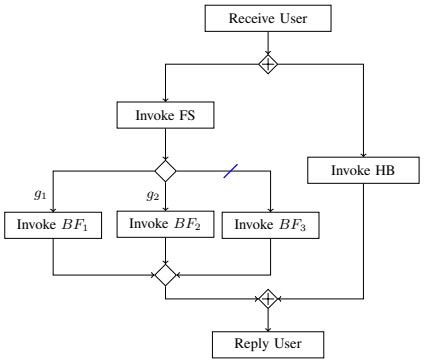
2 Motivating Example

In this work, we introduce four elementary compositional structures for composing the component services, i.e., the sequential (⟨`sequence`⟩), parallel (⟨`flow`⟩), loop (⟨`while`⟩) and conditional (⟨`if`⟩) compositions, which are all the essential structures of many programming languages; therefore, our work can be applied to other languages potentially. In addition, there are three basic activities to communicate with component services, i.e., receive (⟨`receive`⟩), reply (⟨`reply`⟩), and invocation (⟨`invoke`⟩) activities. The ⟨`receive`⟩ and ⟨`reply`⟩ activities are used to receive requests from and reply results to the users of the composite service respectively. The ⟨`invoke`⟩ activity is used to invoke component services for their functionalities. There are two kinds of ⟨`invoke`⟩ activities, i.e., synchronous and asynchronous ⟨`invoke`⟩ activities. The *synchronous* ⟨`invoke`⟩ activity invokes the component service and wait for the reply, while the *asynchronous* ⟨`invoke`⟩ activity moves on after the invocation without waiting for the reply.

2.1 Running Example – Travel Booking Service

In this section, we introduce the Travel Booking Service (TBS) as a running example in this work. TBS is designed for providing a combined budget flight and hotel booking composite service by incorporating with several existing component services. The workflow of TBS is sketched in Figure 1a.

TBS has five component Web services, namely a flight searching service (*FS*), three budget flight booking services (BF_1, BF_2 and BF_3), and a hotel booking service (*HB*). Upon receiving the request from the customer (*Receive User*), a ⟨`flow`⟩ activity (denoted as ⟐) is triggered, and *Invoke FS* and *Invoke HB* are executed concurrently; *Invoke HB* invokes the *HB* service to book the hotel (All invocation activities in this work are assumed to be synchronous, unless otherwise stated). *Invoke FS* invokes the *FS* service to search for budget flights. Upon receiving the reply from the *FS* service,



(a) Travel Booking Service (TBS)

QoS Attribute	*FS*	*HB*	BF_1	BF_2	BF_3
Response Time (ms)	300	200	300	200	100
Availability	1	1	0.95	0.9	0.95
Cost ($)	2	1	2	2	1

(b) QoS for component services of TBS

a conditional activity (denoted as $\diamondsuit$) is followed. If the ticket price of BF_1 is the lowest (represented by the guard condition g_1), BF_1 is invoked (*Invoke* BF_1) to book the flight ticket. If the ticket price of BF_2 is the lowest (represented by the guard condition g_2), then BF_2 is invoked (*Invoke* BF_2) to book the ticket. Otherwise, BF_3 is invoked to book the ticket (*Invoke* BF_3). Upon completion of the concurrent activities, TBS replies the user with a booking confirmation message (*Reply User*).

TBS provides an SLA for their service consumers such that it must respond within 600 ms upon any request with at least 95% availability. The cost per invocation of TBS is 8 dollars – therefore TBS service provider needs to ensure it does not spend more than 8 dollars for its component services.

Now, let us consider a scenario where the flight searching service takes 500 ms. Classic point adaptation strategy may switch some service to an alternating service [5,17,18,15], which has been mentioned in the introduction, as it involves retrying or switching of a particular service. There are cases where such a strategy does not work. For example, there is no alternate service that can satisfy the non-functional requirements. In addition, there might not exist an alternating service that could be switched directly. In such a case, our workflow adaptation strategy, could be used.

2.2 Service Composition Notations

We use the syntax below to specify the workflow of a service composition succinctly.

- $P_1; P_2$ and $P_1 ||| P_2$ are used to denote sequential and concurrent executions of the activities P_1 and P_2 respectively.
- $C([g_1]P_1, [g_2]P_2, \cdots, [g_n]P_n, P_0)$ is used to denote the conditional activity, where g_i is a guard with $i \in \{1, 2, \cdots, n\}$. The guards are evaluated sequentially from g_0 to g_n, and activity P_i is executed for the first g_i that is evaluated to true. If all the guards are evaluated to false, the activity P_0 is executed.
- $sInv(P)$ and $aInv(P)$ are used to denote the synchronous and asynchronous invocations respectively of the activity P.
- $pick(S_1 \Rightarrow P_1, S_2 \Rightarrow P_2)$ is used to denote the pick activity, which contains two branches of *onMessage* activities where exactly one branch would be executed. Activity P_1 is activated when the message from the component service S_1 is received, while activity P_2 is activated if the message from the component service S_2 is received.

QoS Attribute	Sequential	Parallel	Loop	Conditional
Response Time	$\sum_{i=1}^{n} q(s_i)$	$\max_{i=1}^{n} q(s_i)$	$k * (q(s_1))$	$\sum_{i=1}^{n} p_i * q(s_i)$
Availability	$\prod_{i=1}^{n} q(s_i)$	$\prod_{i=1}^{n} q(s_i)$	$(q(s_1))^k$	$\sum_{i=1}^{n} p_i * q(s_i)$
Cost	$\sum_{i=1}^{n} q(s_i)$	$\sum_{i=1}^{n} q(s_i)$	$k * (q(s_1))$	$\sum_{i=1}^{n} p_i * q(s_i)$

Table 1: Aggregation function

The process description of TBS, P_{TBS} is shown in Figure 2a. The numbers annotated to each activity will be introduced in our technical report [3].

3 Preliminaries

In this section, we introduce various notions used in this work. A composite service *CS* is constructed using a finite number of component services. We use S_{CS}=$\langle s_1, s_2, ..., s_n \rangle$ to denote the set of all component services used in CS.

3.1 QoS Attributes

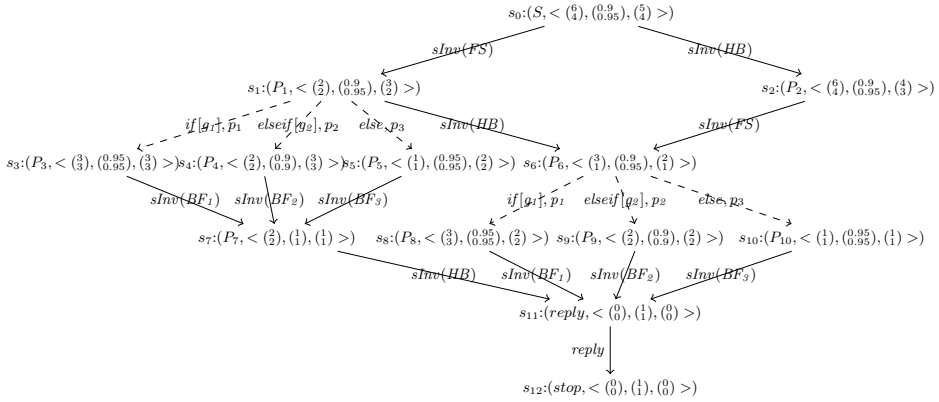
In this work, we use three QoS attributes, i.e., response time, availability and cost as examples to demonstrate our approach. The response time $r \in \mathbb{R}_{\geq 0}$ of a service is defined as the delay between sending the request to the service and receiving the response from it. The availability $a \in \mathbb{R} \cap [0, 1]$ of a service is the probability of the service being available. The cost of a service is the price that incurs by invoking the service. We use $R(a)$, $A(a)$ and $C(a)$ to denote the response time, availability and cost of the activity a respectively. Table 1b lists QoS values for component services of TBS, that will be used in the subsequent sections. There are two kinds of QoS attributes, positive and negative ones. Positive attributes, e.g., availability, provide good effect on the QoS; therefore, they need to be maximized. While negative attributes, e.g., response time and cost, need to be minimized. Our QoS attributes could be addressed similarly as these three QoS attributes. For example, reliability could be handled in the same way as availability.

3.2 QoS for Composite Services

The values of QoS attributes for composite service CS are aggregated from each component service based on internal compositional structures. There are four types of compositional structures: sequential, parallel, loop and conditional compositional structures. Table 1 shows the aggregation function for each compositional structure. In the parallel composition, the response time is the maximum one among response times of all participating component services since all participating component services execute concurrently. In the loop composition, it is aggregated by summing up the response time of the involved component service for k times where k is the number of maximum iteration of the loop and it could be inferred by using loop bound analysis tools (e.g., [10]). In the conditional composition, we use the expected value as the evaluation of guards is not known at the design time, where q_i is the probability for executing the service s_i.

$$P_{TBS} = \{(\{[sInv(FS)]_4^6; C([g_1][sInv(BF_1)]_3^3, [g_2][sInv(BF_2)]_2^2, [sInv(BF_3)]_1^1)_1^3\}_4^6 ||| [sInv(HB)]_2^2)_2^6; [reply]_0^0\}_4^6$$

(a) Process Description of TBS



where S=$(sInv(FS); A)|||(sInv(HB)); reply$, P_1=$A|||sInv(HB); reply$, P_2=$sInv(FS); A; reply$, P_3=$P_7|||sInv(BF_1); reply$, P_4=$P_7|||sInv(BF_2); reply$, P_5=$P_7|||sInv(BF_3); reply$, P_6=$A; reply$, P_8=$sInv(BF_1); reply$, P_9=$sInv(BF_2); reply$, P_{10}=$sInv(BF_3); reply$, A=$C([g_1]sInv(BF_1), [g_2]sInv(BF_2), sInv(BF_3)))$

(b) Probabilistic Partial Model of TBS

Fig. 2: TBS Example

3.3 Probabilistic Partial Models

Our approach is grounded on *probabilistic partial models*, which extend partial models introduced in [11]. In the following, we define various related notions before introducing the probabilistic partial model.

Definition 1 (State). *A* state *s is a tuple (P, V, Q), where P is a service process, V is a (partial) variable valuation that maps variables to their values, and Q is a vector which represents the* local estimation *of the state s, which will be discussed in Section 4.3. We introduce the details of local estimation in Section 4.3.*

Given a state $s = (P, V, Q)$, we use the notation $P(s)$, $V(s)$, and $Q(s)$ to denote the process, valuation, and local estimation of the state s respectively. Two states are said to be equal if and only if they have the same process P, the same valuation V and the same QoS attribute vector Q.

Definition 2 (Transition System). *A* transition system *is a tuple $\langle S, s_0, \Sigma, R\rangle$, where*

- *S is a set of states; $s_0 \in S$ is the initial state; Σ is a set of actions*
- *$R \subseteq S \times \Sigma \times S$ is a transition relation*

For convenience, we use $s \xrightarrow{a} s'$ to denote $(s, a, s') \in R$. Given a state $s \in S$, $Enable(s)$ denotes the set of states reachable from s by one transition, formally, $Enable(s) = \{s' | (s' \in S) \wedge (a \in \Sigma) \wedge (s \xrightarrow{a} s' \in R)\}$. An action a is enabled by s if there exists a state s' such that $s \xrightarrow{a} s'$. $Act(s)$ is denoted as the set of actions that can be triggered from s, formally, $Act(s) = \{a | (a \in \Sigma) \wedge (s' \in S) \wedge (s \xrightarrow{a} s' \in R)\}$. An *execution* π is a finite alternating sequence of states and actions $\langle s_0, a_1, s_1, \ldots,$

$s_{n-1}, a_n, s_n\rangle$, where $\{s_0, \ldots, s_n\} \in S$ and $s_i \overset{a_{i+1}}{\rightarrow} s_{i+1}$ for all $0 \leq i < n$. We denote the execution π by $s_0 \overset{a_1}{\rightarrow} s_1 \overset{\cdots}{\rightarrow} s_{n-1} \overset{a_n}{\rightarrow} s_n$. A state s is *reachable* if there exists an execution that starts from the initial state s_0 and ends in the state s. A state s is called a *terminal* state if $Act(s)$ is empty. Given an action $a \in \Sigma$, *A(a)*, *R(a)* and *C(a)* denote the availability, response time and cost of the action a. The transition system is generated based on the formal semantics of service process described in [12]. Given a composite service *CS*, we use $\mathcal{T}(CS)$ to denote the transition system of *CS*.

Definition 3 (Probabilistic Partial Models). *A* probabilistic partial model *is a tuple* $\langle \mathcal{M}, \mathcal{F}, C_g, \mathcal{P} \rangle$*, where* $\mathcal{M} = \langle S, s_0, \Sigma, R \rangle$ *is a transition system,* $\mathcal{F}$ *is a function:* $S \times \Sigma \rightarrow \mathcal{B}$*, where* $\mathcal{B}$ *is the set* $\{True, False, Maybe\}$*, and* $C_g = \langle C_g^R, C_g^A, C_g^C \rangle$ *is the global constraints for the model where* C_g^R *(resp.,* C_g^A*,* C_g^C*) is the global response time (resp., availability, cost) constraint.* $\mathcal{P}$ *is a function:* $S \times \Sigma \rightarrow p$ *where* $p \in \mathbb{R} \cap [0, 1]$.

For convenience, given a composite service *CS*, we use $\mathcal{P}(CS)$ to denote the probabilistic partial model of *CS*. $\mathcal{P}(CS)$ is extended from $\mathcal{T}(CS)$ by mapping values (e.g., $True$, or $Maybe$, 0.5) for transitions on $\mathcal{T}(CS)$. We illustrate how the value on transitions of $\mathcal{P}(CS)$ are decided. Given an action $a \in$ *Act(s)*, $F(s, a)$ denotes whether action $a \in \Sigma$ could be executed from state s, $P(s, a)$ provides the probability of executing the action $a \in \Sigma$ at the state s. Clearly, $F(s, a) = False$ and $P(s, a) = 0$ if $a \notin$ *Act(s)*. $F(s, a) = True$ and $P(s, a) = 1$ if action $a \in$ *Act(s)* and could always be executed regardless the valuation of the variables. Otherwise, $F(s, a) = Maybe$ and $\sum_{a \in MAct(s)} P(s, a) = 1$ where *MAct(s)*$= \{a | a \in$ *Act(s)*$\wedge (\mathcal{F}(s, a) =$ *Maybe*$)\}$. *MAct(s)* represents a set of *Maybe* actions from s, where exactly one of actions $a \in MAct(s)$ would be executed. The execution of a $Maybe$ action depends on the evaluation of the guard (e.g., $\langle$`if`$\rangle$ activity), or dependent on the response from other component services (e.g., $\langle$`pick`$\rangle$ activity). We also use $TAct(s)$ to denote the set of *True* actions enabled by s; formally, *TAct(s)*$= \{a | a \in$ *Act(s)*$\wedge (\mathcal{F}(s, a) =$ *True*$)\}$. For example, actions $if[g_1]$, $elseif[g_2]$, and $else$ (with p_1, p_2 and p_3 as their respective probabilities) are $Maybe$ actions, since the execution of these actions dependent on the evaluation of the guard conditions. In contrast, actions FS and HB are $True$ actions, since both actions are triggered concurrently at state s_0.

Consider the probabilistic partial model of TBS, $\mathcal{P}(TBS)$, as shown in Figure 2b. Recall that a state is represented as (P, V, Q). Since $V = \emptyset$ for all states in $\mathcal{P}(TBS)$, we represent states in $\mathcal{P}(TBS)$ as (P, Q) for simplicity. An edge is shown using solid (resp., dotted) arrow if the triggered action is a $True$ (resp., $Maybe$) action, and an edge is labelled with probability if the triggered action is a $Maybe$ action. Since the probability is 1 if the action is a $True$ action, we omit the 1 in the $\mathcal{P}(TBS)$.

4 ADFLOW Framework

In the following, we introduce a framework for supporting self-adaptation based on runtime information. The goal is to satisfy the system's global constraints with best efforts. We first introduce the architecture of the ADFLOW framework based on asynchronous monitoring. After that, we focus on the local estimation of probabilistic partial model and demonstrate how it can be used for the runtime adaptation.

In the following, Section 4.1 describes the architecture of ADFlow, Section 4.2 introduces the notion of *controllability* for activities. Section 4.3 introduces calculations for pessimistic and probabilistic estimation, and then Section 4.4 shows how the framework adaptively chooses an action based on the probabilistic estimation. Section 4.5 presents the asynchronous monitoring technique used in our approach.

4.1 Architecture of ADFlow

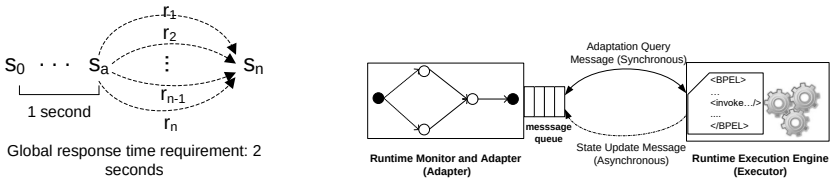
The architecture of ADFLOW is shown in Figure 3b. ADFLOW consists of two essential components: the *Runtime Monitor and Adapter* (ADAPTER) and the *Runtime Execution Engine* (EXECUTOR). The ADAPTER monitors and keeps track of the execution of the programs using the probabilistic partial model, and provides adaptation if needed based on the local estimation of the probabilistic partial model. On the other hand, the EXECUTOR provides the environment for the execution of the service programs.

During the deployment of a composite service *CS* on EXECUTOR, the corresponding *probabilistic partial model* of *CS*, $\mathcal{P}(CS)$, will be automatically generated (before the execution of *CS*), stored and maintained by ADAPTER. As for each action execution of *CS*, ADAPTER will update the *active state pointer* that points to the current execution state $s_a \in S$ of $\mathcal{P}(CS)$. We call s_a the *active state* of $\mathcal{P}(CS)$. During the execution of *CS*, for every action performs by the EXECUTOR (e.g., invocation of a component service), a timer is used to record the duration of the action. Upon completion of the action, a *state update* message containing the information of the completed action and the duration is sent by the EXECUTOR to the ADAPTER, so that ADAPTER could update the current active state of the probabilistic partial model.

4.2 Controllability of Activity

Controllable activities are the activities that could be controlled by ADAPTER. They must be the activities that use *Maybe* actions (i.e., activities ⟨`if`⟩ and ⟨`pick`⟩). The reason for not controlling activities using *True* actions is that, *True* actions of an active state would definitely be executed at some point of the execution. Therefore, it will not provide any improvement for QoS of the composite service by controlling *True* actions. For example, consider TBS at the initial state s_0 in Figure 2b, the enabled *True* actions $sInv(FS)$ and $sInv(HB)$, *must be* executed at some points for all executions that start from the initial state s_0 and end at the terminal state s_{12}. On the other hand, for *Maybe* actions (e.g., $if[g_1]$), they *may or may not be* executed (e.g., depends on the evaluation of their guards). Suppose ADAPTER detects the possible violation of the global constraints, and if the action to be executed next is controllable by ADAPTER, then ADAPTER could choose an action, that maximizes the chance of satisfying the global constraints, to be executed by EXECUTOR.

Consider TBS with active state at state s_1 in Figure 2b, which has three *Maybe* actions, i.e., $if[g_1]$, $elseif[g_2]$, and $else$. For an ⟨`if`⟩ activity, it is the evaluation of guard conditions that decides which branch to execute. It is a violation of the semantics of the ⟨`if`⟩ activity if EXECUTOR, simply follows a different action (e.g., $elseif[g_2]$) chosen by ADAPTER, without checking the evaluation of the guard condition. For this purpose, we extend the ⟨`if`⟩ activity with an attribute `ctr`, so that users are allowed to specify whether the ⟨`if`⟩ activity is controllable by ADAPTER. If `ctr` is set to `true`, then EXECUTOR would send an *Adaptation Query* message to ADAPTER to consult which



(a) ADFLOW Example **(b)** ADFLOW Architecture

Fig. 3: ADFLOW

action to be executed next. ADAPTER would either select an action to be executed or decide not to control if there is no potential violation of the global constraints detected, and then replies to EXECUTOR. If ADAPTER chooses an action, EXECUTOR would disregard the valuation of guard condition and execute the action that is chosen by ADAPTER.

Given an activity P, $Ctrl(P) \in \{true, false\}$ denotes the controllability of P, which is defined recursively with Equation (1). If P is a sequential activity $P_1; P_2$, the controllability of P is decided on the controllability of process P_1. For a concurrent activity $P = P_1|||P_2$, P is controllable if either activity P_1 or activity P_2 is controllable, since activities P_1 and P_2 are triggered at the same time. For conditional activity $P = C([g_1]P_1, [g_2]P_2, ...)$, the controllability is decided by the user-specified controllability of the conditional activity C.

$$Ctrl(P) = \begin{cases} Ctrl(P_1) & if\ P(s) = P_1; P_2 \\ Ctrl(P_1) \vee Ctrl(P_2)\ \ if\ P(s) = P_1|||P_2 \\ Ctrl(C)\ if\ P(s) = C([g_1]P_1, [g_2]P_2, ...) \end{cases} \quad (1)$$

4.3 Local Estimation

In this section, we introduce the *local estimation* and the method to calculate it. The local estimation of a state s provides an estimation of QoS from two perspectives, pessimistic and probabilistic, for all executions starting from state s.

Pessimistic estimation. The pessimistic estimation of a QoS attribute a provides a conservative estimation of the attribute a for all executions starting from the state s. For example, the pessimistic estimation of state s for the response time attribute is the maximum response time that is required for all executions starting from state s. The pessimistic estimation is used to help ADAPTER to decide whether to take over the composite service at the active state s_a. For example shown in Figure 3a, suppose the total response time from the initial state s_0 to state s_a takes 1 second, and the global constraints for the response time is 2 seconds. If the pessimistic estimation of the response time at state s_a is r seconds, where $r > 1$, then the runtime adaptation is required. The reason is that since $1 + r > 2$ seconds, there exists an execution path that could violate the global constraint of the response time.

Probabilistic estimation. The probabilistic estimation of the QoS attribute a provides the expected value for the attribute a for all possible executions starting from state s. The probabilistic estimation is used to guide the ADAPTER to choose an action to be executed next in order to maximize the chances to satisfy the global constraints. The local estimation $Q(s)$ of a state s is represented by a vector $\langle L_R(s), L_A(s), L_C(s)\rangle$, where $L_R(s)$, $L_A(s)$ and $L_C(s)$ represent the local estimation of response time, avail-

ability and cost for the state s respectively. The local estimation of a QoS attribute is a vector $\binom{pe}{pr}$, where $pe, pr \in \mathbb{R}$ represent the pessimistic and probabilistic estimation of the QoS attribute respectively. Henceforth, we denote the pessimistic and probabilistic estimation of the response time of a state s by $L_R^{pe}(s)$ and $L_R^{pr}(s)$ respectively. We define $L_A^{pe}(s)$, $L_A^{pr}(s)$, $L_C^{pe}(s)$, and $L_C^{pr}(s)$ in a similar manner.

Different QoS attributes might have different aggregation functions for different compositional structures. For QoS attributes (e.g., cost, availability) that only make use of summation and multiplication aggregation functions, we only require *backward value propagation* (discussed in our technical report [3])for calculating the local estimation. For QoS attributes (e.g., response time) that involve the usage of maximization or minimization aggregation functions, *backward tagging propagation* (discussed in our technical report [3]) need to be applied, before backward value propagation.

4.4 Runtime Adaptation

Given a set of *Maybe* actions, ADAPTER needs a metric to decide the best action for execution. The *local optimality value* of an action a, denoted by $L(a)$ is used to provide a value that represents the worthiness of choosing the action a. In this section, we introduce the calculation of local optimality value, and the adaptation algorithm.

Local Optimality Value We first introduce the notion of QoS optimality value of an action a which will be used for calculation of local optimality value for the action a.

Given a state s, and an action $a \in MAct(s)$, the *QoS optimality value* of the action a, denoted by $Q(a)$, is the expected QoS of all (finite) executions by executing the action a at s. It is calculated using a Simple Additive Weighting (SAW) method [24]. For the purpose of normalization, the action a compares the probabilistic estimations of its QoS attributes with the maximum and minimum probabilistic estimations of all enabled *Maybe* actions. The calculation of $Q(a)$ is provided in Equation (2), where $w_i \in \mathbb{R}^+$ is the weight with $\sum_{i=1}^{3} w_i = 1$. The *local optimality value* of an action a, denoted by $L(a)$, is calculated using Equation (3), where $S_r(a), S_a(a), S_c(a) \in \{true, false\}$ denote whether the execution of action a could allow potential satisfaction of global constraints of response time, availability and cost respectively. Function $f_b(b)$ takes an input $b \in \{true, false\}$. When b is true, $f_b(b)$=1, otherwise, $f_b(b)$=0. The local optimality value of the action a ranges from 0.5 to 1 if $S_r(a) \wedge S_a(a) \wedge S_c(a)$, otherwise $L(a)$ ranges from 0 to 0.5. Therefore, it could guarantee that the local optimality values of actions that could possibly satisfy the global constraints are higher than the one that could not.

$$
\begin{aligned}
Q(a) = {} & w_1 \cdot \frac{U_{Max}^{(r)}(s) - a.probtag}{U_{Max}^{(r)}(s) - U_{Min}^{(r)}(s)} \\
& + w_2 \cdot \frac{A(a) \cdot L_A^{pr}(s') - U_{Min}^{(a)}(s)}{U_{Max}^{(a)}(s) - U_{Min}^{(a)}(s)} \\
& + w_3 \cdot \frac{U_{Max}^{(c)}(s) - (C(a) + L_C^{pr}(s'))}{U_{Max}^{(c)}(s) - U_{Min}^{(c)}(s)}
\end{aligned}
\quad with \quad
\begin{aligned}
U_M^{(r)}(s) &= \underset{a \in MAct(s)}{\mathrm{M}}(a.probtag) \\
U_M^{(a)}(s) &= \underset{a \in MAct(s)}{\mathrm{M}}(A(a) \cdot L_A^{pr}(s')) \\
U_M^{(c)}(s) &= \underset{a \in MAct(s)}{\mathrm{M}}(C(a) + L_C^{pr}(s')) \\
M &\in \{\min, \max\}
\end{aligned}
\tag{2}
$$

$$
L(a) = 0.5 \cdot Q(a) + 0.5 \cdot f_b(S_r(a) \wedge S_a(a) \wedge S_c(a)) \tag{3}
$$

Algorithm 1: Algorithm $ChooseAction$

input : s, the active state
input : $ctime$, current time
input : $stime$, execution start time
input : c, cost that has been incurred so far
output: a, the next action to execute

1 **if** $Ctrl(P(s))$ **then**
2 $\quad S_r \leftarrow ((ctime - stime + L_R^{pe}(s)) \leq C_g^R)$;
3 $\quad S_a \leftarrow (L_A^{pe}(s) \geq C_g^A)$; $S_c \leftarrow ((c + L_C^{pe}(s)) \geq C_g^C)$;
4 $\quad$ **if** $\neg(S_r \ \wedge \ S_a \ \wedge \ S_c)$ **then**
5 $\quad\quad$ return $\underset{a \in MAct(s)}{\operatorname{argmax}}(0.5 \cdot Q(a) + 0.5 \cdot f_b(S_r(a) \wedge S_a(a) \wedge S_c(a)))$;
6 return $\emptyset$;

Adaptation Algorithm The adaptation algorithm is shown in Algorithm 1, which is used to choose the action to execute next. In Algorithm 1, the variable $s \in S$ is the active state reached by the execution, $ctime$ and $stime$ are the current time and start time of the execution respectively, and $c \in \mathbb{R}_{\geq 0}$ is the cost that has been incurred from the initial state to state s. Line 1 checks whether Runtime Adapter could control the activity $P(s)$. If $P(s)$ is controllable, then the algorithm proceeds in checking the potential satisfaction of global constraints. In line 2, it calculates the potential satisfaction of global constraint of response time, S_r, by checking that the duration of execution so far ($ctime - stime$) added with the pessimistic estimation of state s ($L_R^{pe}(s)$) is not larger than the global constraint of response time C_g^R. If the result is false, then there exists an execution that could violate C_g^R; otherwise, any execution from state s would allow satisfaction of C_g^R. The calculation of S_a and S_c (line 3) can be described in a similar manner.

If not all the global constraints for response time, availability and cost are detected to be satisfiable based on the pessimistic estimation (line 4), then the algorithm will return a best action with the highest local optimality value (line 5). Otherwise, the algorithm will return an empty action (line 6), which signals that an adaptation is not required.

4.5 Asynchronous Monitoring

ADAPTER might require to deal with multiple concurrent state update messages due to the concurrent execution of activities in the composite service (recall that service composition supports the parallel composition). Synchronous communication between the ADAPTER and the EXECUTOR for each state update message could result in high overhead and the parallel execution in the EXECUTOR can be "sequentialized". To be efficient, ADFLOW adopts an *asynchronous monitoring* mechanism. That is, asynchronous communication is used between the ADAPTER and the EXECUTOR during normal situations, and synchronous communication is used when it is necessary. In particular, all the state update messages are sent *asynchronously* to the message queue, and the ADAPTER updates states in batches on the probabilistic partial model. Synchronous communication is used only when the EXECUTOR encounters controllable activities. In such a case, an *adaptation query* message is sent to the message queue synchronously (i.e., the EXECUTOR waits for the reply before continuing execution) to consult whether there is a need for adaptation before their execution. The asynchronous monitoring of

ADFLOW is shown in Figure 3b. We have shown that synchronous monitoring has effectively reduced the overhead for monitoring (see Section 5 for the evaluation).

5 Evaluation

To reduce the external noise and control the non-functional aspect of a service, we make use of controlled experiment to evaluate our approach. We aim to answer the following research questions:

RQ 1. What is the *overhead* of ADFLOW?

RQ 2. What is the *improvement* provided by ADFLOW on the conformance of global constraints?

RQ 3. How is the *scalability* of ADFLOW?

The evaluation was conducted using two different physical machines, which are connected by a 100 Mbit LAN. One machine is running ApacheODE [1] to host the Runtime Engine to execute the service program, configured with Intel Core I5 2410M CPU with 4GiB RAM. The other machine is to host the Runtime Adapter, configured with Intel I7 3520M CPU with 8GiB RAM.

We use two case studies in this paper to evaluate our approach: Travel Booking Services and Large Service. Component services used in both services are real-world services that are set up on the server.

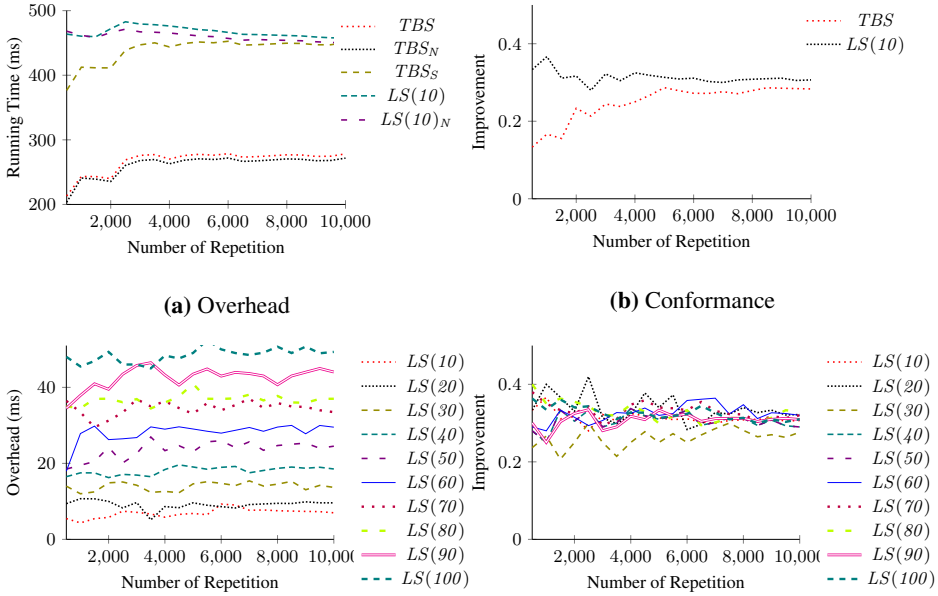
Travel Booking Service (TBS). This is the running example that has been used through out the paper.

Large Service (LS). To evaluate the scalability of our approach, we construct a large service LS with sequential execution of k *base activities*. The base activity is constructed by sequential execution of a synchronous invocation, followed by a controllable conditional activity with three branches which one branch has a better QoS, and subsequently followed by a concurrent activity. We denote the composite service with sequential execution of k base activities as *LS(k)*, which would consult ADAPTER for adaptation for k times since there are k controllable conditional activities.

5.1 Setup of Controlled Experiments

Given a composite service CS, we denote all component services that are used by CS as S_{CS}. Given a component service $s_i \in S_{CS}$, we use $R_e(s_i)$, $A_e(s_i)$, and $C_e(s_i)$ to denote the estimated response time, availability and cost of the component service s_i, which are either recorded in SLA or predicted based on historical data.

To test the composite service under controlled situation, we introduce the notion of *execution configuration*. An execution configuration which defines a particular execution scenario for the composite service. Formally, an execution configuration E is a tuple (M, Q), where M decides which path to choose for $\langle$`if`$\rangle$ and $\langle$`pick`$\rangle$ activities and Q is a function that maps a component service $s_i \in S_{CS}$, to a vector $\langle R(s_i), A(s_i), C(s_i)\rangle$. $R(s_i)$, $A(s_i)$ and $C(s_i)$ are QoS values for response time, availability, and cost of s_i. We discuss how an execution configuration $E = (M, Q)$ is generated. M is generated based on the probabilities of each branch of the conditional activities. Q is generated based on conformance parameter $p_c \in \mathbb{R} \cap [0, 1]$ and the estimated QoS attribute values. Given a composite service CS, we denote the estimated value of response time for a component service $s_i \in S_{CS}$ as $R_e(s_i)$. $R(s_i)$ will be assigned with a value from [0, $R_e(s_i)$] normally with the probability of p_c, and assigned



(a) Overhead

(b) Conformance

(c) Scalability (Conformance)

(d) Scalability (Improvement)

Fig. 4: Experiment Results

with a value from $[R_e(s_i), 3 \cdot R_e(s_i)]$ normally with the probability of $1 - p_c$. Values $A(s_i)$ and $C(s_i)$ are generated similarly.

Given a composite service CS, and an *execution configuration* E, we denote a run as $r(CS, A, E)$, where the second argument $A \in \{\text{ADFLOW}, \emptyset\}$ is the adaptive mechanism where $\emptyset$ denotes no adaptation. Two runs $r(CS, A, E)$ and $r(CS', A', E')$, are *equal*, iff $CS = CS'$, $A = A'$ and $E = E'$. Noted that all equal runs have the *same* execution paths, aggregated response times, availabilities costs.

5.2 Evaluation

We conduct three experiments **E1**, **E2**, and **E3**, to answer the research question **RQ1**, **RQ2**, and **RQ3**, respectively. Each experiment is repeated for 10000 times, and a configuration generation E is randomly generated for each repetition. We show the experiments and their results in the following.

E1: The overhead of our approach mainly comes from two sources: the asynchronous monitoring and synchronous adaptation. Given a composite service CS, in order to measure the overhead, we first generate an execution configuration $E = (M, Q)$ for an adaptive run $r(CS, \text{ADFLOW}, E)$. Adaptive run may not select a branch according to M, since the selection of a branch could also be decided by the ADAPTER, in the case where ADAPTER decides to control a controllable conditional structure. Therefore, after the adaptive run, we modifies M to M', according to the actual conditional branch selected by the ADAPTER. Then, using the M', we perform the non-adaptive run $r(CS, \emptyset, E')$, where $E' = (M', Q)$. These ensure that both adaptive run and non-adaptive run have the same execution, which allow effective measurement of the overhead introduced by ADFLOW. In this experiment, we set the conformance of

each component service to 0.8. We compare the overhead of the following:
No Adaptation. Execution of the service program without the adaptation, for which we append the name of case studies with a subscript N, i.e., TBS_N, $LS(10)_N$.
Synchronous Adaptation. Runtime adaptation using synchronous monitoring (in contrast to our asynchronous monitoring approach) with ADFLOW, for which we append the name of case studies with a subscript S, e.g., TBS_S, $LS(10)_S$.
ADFLOW approach. Runtime adaptation using ADFLOW, for which the case studies are specified without any subscript, e.g., TBS, $LS(10)$.
Results. The experiment results can be found in Figure 4a. Note that due to the space constraint, the result of $LS(10)_S$ is not shown in our results. The average running time of TAS with adaptation is 278.28 ms and the average running time of TAS without adaptation is 271.69 ms; therefore the overhead is only 6.59 ms, 2.3% of the running time. In contrast, the overhead for synchronous monitoring is 179.12 ms for TAS. On the other hand, the average running time of LS(10) is 457.65 ms and the average running time of LS(10) without adaptation is 450.66 ms; therefore, average overhead is 6.99 ms. In contrast, the overhead for the adaptation using synchronous monitoring is around 1100 ms. The results show that our approach has a little overhead, and compared to the adaptation using synchronous monitoring, our approach reduces the overhead noticeably.
E2: In this experiment, we measure the improvement for the conformance of global constraints due to ADFLOW. Given a composite service CS, a randomly generated execution configuration E, two runs $r(CS, \text{ADFLOW}, E)$ and $r(CS, \emptyset, E)$ are conducted. N_{se} is the number of executions that satisfy global constraints for composite service with ADFLOW, and N_e is the number of executions that satisfy global constraints for composite service without ADFLOW. The improvement is calculated by the formula $Improvement = (N_{se} - N_e)/10000$. We perform the experiment for 10000 times.
Results. The experiment results can be found in Figure 4b. We notice that although the improvement fluctuates at the beginning, ADFLOW always provides an improvement, compared to no adaptation. We also notice that the improvement provided by ADFLOW starts to converge when the number of repetition grows. Overall, our approach improves 0.283 over TBS_N and improves 0.3 over $LS(10)_N$. The experiment results show that our approach noticeably improves the conformance of global constraints.
E3: We compare the overhead and improvement with respect to the size of LS, ranging from 10 to 100.
Results. The experiment results can be found in Figure 4c and 4d. In Figure 4c, the overhead increases with the size of LS, due to the reason that more synchronous adaptations are required with the size of the composite service increases. Nevertheless, we still have low overhead compared to the total running time, which is around 1% – 3%. In Figure 4c, we observe that the improvement for each case studies fluctuates between 0.2 – 0.42 at the beginning. The improvement starts to converge when the number of repetition grows. On average, the improvement for the case studies is between 25 % – 32 %. This is consistent to our observations in experiment **E2**. Together, these show our approach scales well.

6 Related Work

In [5], Cardellini *et al.* propose to use a set of service components to implement the functionality of a component service adaptively. Their work focuses on adapting a

single service for the purpose of decreasing response time and increasing availability. In [17], Moser *et al.* propose a framework that uses non-intrusive monitoring based on aspect-oriented programming (AOP), to detect failure service and replace them at runtime. In [15], Irmert *et al.* present the CoBRA framework to provide runtime adaptation, where the infeasible component services are replaced at runtime. In [18], Mukhija and Glinz propose an approach to adapt an application by recomposing its components dynamically, which implemented by providing alternative component compositions for different states of the execution environment. This work is orthogonal to our approach, they adopt point adaptation strategy, while we adopt workflow adaptation strategy.

Our work is also related to the non-functional aspect of Web service composition. In [13], Fung *et al.* propose a message model tracking model to support QoS end-to-end management. In [16], Koizumi *et al.* present a business process performance model which integrates the Timed Petri model and statistical model to estimate process execution time. Epifani et al. [9] present the KAMI approach to update model parameters by exploiting Bayesian estimators on collected runtime data. These aforementioned works are concerned with the prediction of QoS attributes, while our work focuses on runtime adaptation based on QoS attributes. In [20], given the response time requirement of the composite service, Tan *et al.* propose a technique to synthesize the local time requirement for component services that are used to compose the service. In [7,6,23,19], we focus on verification of combined functional and non-functional properties of the web service composition based on QoS of each component service. In [22,21], we propose to solve the optimal selection problem and recovery problem so that it could satisfy the requirements. The aforementioned works are orthogonal to this work.

7 Conclusion

In this paper, we have presented ADFLOW, a novel approach for monitoring and self-adapting the running of Web service composition to maximize its ability to satisfy the global constraints. ADFLOW uses workflow adaptation strategy, by selecting the best path for execution when necessary. In addition, ADFLOW adopts asynchronous monitoring to reduce the overhead. The evaluation has shown the efficiency and effectiveness of our approach. In particular, given a composite service, we achieve 25% – 32% of average improvement in the conformance of non-functional requirements, and only incur 1% – 3% of overhead with respect to the execution time. For future work, we plan to investigate the applicability our approach to other domains such as sensor networks [4].

References

1. Apache ODE. http://ode.apache.org/.
2. Microservices. http://microservices.io/patterns/microservices.html.
3. Technical report. http://tianhuat.bitbucket.org/technicalReport.pdf.
4. I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Sensor networks. *Y, et a1. A survey on IEEE Communications Magazine*, 40(8):102–114, 2002.
5. V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. L. Presti, and R. Mirandola. Moses: A framework for qos driven runtime adaptation of service-oriented systems. *TSE*, 38(5):1138–1159, 2012.

6. M. Chen, T. H. Tan, J. Sun, Y. Liu, and J. S. Dong. Veriws: a tool for verification of combined functional and non-functional requirements of web service composition. In *ICSE*, pages 564–567, 2014.
7. M. Chen, T. H. Tan, J. Sun, Y. Liu, J. Pang, and X. Li. Verification of functional and non-functional requirements of web service composition. In *ICFEM*, pages 313–328, 2013.
8. R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (WSDL) version 2.0. http://www.w3.org/TR/wsdl20/.
9. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE*, pages 111–121, 2009.
10. A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *WCET*, 2007.
11. M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *ICSE*, pages 573–583, 2012.
12. H. Foster. *A rigorous approach to engineering Web service compositions*. PhD thesis, Citeseer, 2006.
13. C. K. Fung, P. C. K. Hung, G. Wang, R. C. Linger, and G. H. Walton. A study of service composition with qos management. In *ICWS*, pages 717–724, 2005.
14. M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. Simple object access protocol (SOAP) version 1.2. http://www.w3.org/TR/soap12/.
15. F. Irmert, T. Fischer, and K. Meyer-Wegener. Runtime adaptation in a service-oriented component model. In *SEAMS*, pages 97–104. ACM, 2008.
16. S. Koizumi and K. Koyama. Workload-aware business process simulation with statistical service analysis and timed petri net. In *ICWS*, pages 70–77, 2007.
17. O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *WWW*, pages 815–824, 2008.
18. A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *ARCS*, pages 124–138, 2005.
19. T. H. Tan. Towards verification of a service orchestration language. *ISSRE*, pages 36–37, 2010.
20. T. H. Tan, É. André, J. Sun, Y. Liu, J. S. Dong, and M. Chen. Dynamic synthesis of local time requirement for service composition. In *ICSE*, pages 542–551, 2013.
21. T. H. Tan, M. Chen, É. André, J. Sun, Y. Liu, and J. S. Dong. Automated runtime recovery for qos-based service composition. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*, pages 563–574, 2014.
22. T. H. Tan, M. Chen, J. Sun, Y. Liu, É. André, Y. Xue, and J. S. Dong. Optimizing selection of competing services with probabilistic hierarchical refinement. In *ICSE*, pages 85–95, 2016.
23. T. H. Tan, Y. Liu, J. Sun, and J. S. Dong. Verification of orchestration systems using compositional partial order reduction. In *ICFEM*, volume 6991, pages 98–114, 2011.
24. K. Yoon and C. Hwang. *Multiple attribute decision making: an introduction*. Number 102-104. Sage Publications, Incorporated, 1995.