International Journal of Cooperative Information Systems
Vol. 23, No. 3 (2014) 1450004 (20 pages)
© World Scientific Publishing Company
DOI: 10.1142/S021884301450004X

Dynamic Transaction Aware Web Service Selection

Kanchana Rajaram*,‡, Chitra Babu*,§ and Arun Adiththan†,¶

*Department of Computer Science and Engineering
SSN College of Engineering
Anna University, Chennai – 603110, Tamil Nadu, India

†Department of Computer Science
Graduate School and University Center
City University of New York (CUNY)
New York, NY 10016, USA
‡rkanch@ssn.edu.in
§chitra@ssn.edu.in
¶arunadiththan@gmail.com

Received 17 May 2012
Accepted 19 March 2014
Published 20 June 2014

Web service composition, that recursively constructs a composite web service out of the existing services based on a business workflow has been acknowledged as a promising approach to meet the user demands, whenever a single service alone cannot fulfil the needs. In view of frequent failures in the internet environment where the composed service is executed, reliability of the composed service must be ensured. The reliability is determined by the behavioral or transactional properties of component services. The component services for each activity of the workflow must be selected based on their behavior so that their execution results in a consistent termination. Service selection must happen at run-time in order to consider the services available in a service registry at the time of execution. Towards this need, a dynamic transaction aware web service selection approach is proposed in this paper. Further, whenever user requirements change, a long running transaction must be interrupted and cancelled which is not addressed by any of the existing works. Hence, service cancellability property is proposed in this paper and incorporated in the dynamic selection approach. The overhead of the proposed run-time selection approach is assessed and the impact of increased services on its performance is also measured.

Keywords: Web services; SOA; workflow; transaction; dynamic selection; reliability.

1. Introduction

Service-oriented computing (SOC)[1] is an emerging paradigm that facilitates a new generation of software applications that can run over heterogeneous network infrastructures. In a dynamic environment, services need to be discovered and bound on the fly so that, collectively, they can fulfil the intended business goals. Service-oriented architecture (SOA)[2] provides an effective approach to build business

‡Corresponding author.

applications wherein service interfaces are published and discovered on demand. Web services are the predominant standard-based way of realizing SOA. Web services are described using web services description language (WSDL),[3] an XML-based language. Service providers publish their services in an XML-based registry over the Internet using universal description discovery and integration (UDDI)[4] and the consumers discover the required service from the registry to consume the services.

In general, business processes involve collaboration of multiple business organizations. Workflows can be used to model a business process. A workflow consists of various predefined activities, depending on the way in which the business should be conducted. Web service composition involves combining services developed by different organizations offering diverse functionalities with different transactional behavior, to offer a more complex service. There have been various approaches for composition of services[5,6] wherein the services are discovered either at design-time (static) or at run-time (dynamic). If the behavior of the service is ignored while selecting the services involved in a composition, it may affect the overall behavior of the composite service resulting in an inconsistent termination. For example, if a compensatable business process or a composite service is required, each of its component services must be compensatable. If every individual service that is composed, does not exhibit compensatable behavior, the overall composite service cannot be compensated. Hence, while composing the services, only the services with appropriate behavior that will result in a reliable execution of the overall composition, must be chosen.

The behavioral or transactional properties of component services determine the reliability of the composed service. If the services are selected and composed statically, during the execution of services, newer versions of services and services provided by new service developers that were deployed after the composition, may not be considered. Moreover, services discovered at design-time may not be available during the execution of the application. Such static composition is suitable only when the functional requirements of the component services are completely known at design-time and do not often vary. However, in business-to-business (B2B) and business-to-consumer (B2C) application scenarios, users would like to specify requirements based on the outcome of the previous service execution. Moreover, business policies change frequently depending on business trends and growth. Hence, services must be discovered dynamically at run-time. Existing works on dynamic composition of web services[7–12] do not address reliable execution of the composed services. Hence, in this paper, a transaction aware web service selection approach is proposed that selects web services on the fly, with appropriate transactional properties resulting in a reliable execution of the composed service.

Whenever user requirements or business policies change during the life span (ranging from one day to one month or more) of a long running service, the service execution must be cancelled to accommodate the new requirements or to incorporate the changed policies. Otherwise, execution of the service to completion with the older requirements or policies is no longer meaningful. In case of a cancelable long running service consumption, the consumer has only the overhead associated with the compensation of the completed portion of the service. Alternatively, in case of a non-cancelable long running service consumption, the consumer needs to wait until the completion and has to incur considerably additional overhead to compensate the entire execution. This is because the successful execution will no longer be meaningful or useful in the scenario of changed requirements. For example, let us consider an order processing service that is a composite long running service (of 2–6 weeks duration) comprising of stock updation and invoice preparation. After placing an order, if the user preferences change during the invoice preparation, the order must be immediately cancelled and the completed stock updates should be rolled back. Suppose the order processing service is not *cancelable*, the user has to wait for its completion and then undoing the effect of both the component services. Thus, selection of a non-cancelable long running service involves a higher overhead for the consumer. Hence, cancellability of a service enables flexible usage for the consumers with lesser overhead. In addition, cancellation of a process supports cancellation recovery which compensates the completed portions of the process. This motivated us to propose a new transactional property named *cancelable* to be considered for long running services.

The rest of the paper is structured as follows. The existing literature related to the proposed work is discussed in Sec. 2. An illustrative application to support the motivation for the present work is discussed in Sec. 3. Section 4 illustrates in detail, the proposed approach along with the experimental analysis of its performance. Section 5 concludes the paper highlighting the contributions.

2. Existing Work

Recently, web service selection has triggered extensive research efforts. In this section, literature related to the technique proposed in this paper is discussed.

2.1. *Transaction aware web service selection*

Existing works on transaction aware web service selection are based either on accepted termination state (ATS) model or transactional properties. The approaches based on the ATS model[13–16] require the specification of ATS, which is a non-trivial and difficult task, as it is manually specified by the user. Moreover, the set of input ATSs are assumed to be correct and adequate. Inadequacy of dependencies is addressed by Gaaloul *et al.*[17] by analyzing the workflow logs to subsequently improve and correct the related recovery mechanisms. However, this approach is reactive since design gaps are detected only after the execution.

There are a few works on static selection of web services based on transactional properties. A mediator-based approach[18] is proposed to gather functionally similar web services with different transactional properties. This approach resolves the heterogeneity among the web services. Li *et al.*[19] proposed an approach to derive transactional properties of a composite web service from the transactional properties of its components. Haddad *et al.*[20] proposed an algorithm, namely transactional quality of service (TQoS), for web service selection based on transactional as well as quality of service (QoS) properties. TQoS approach satisfies the requirement of transactional behavior of composite services first and then applies the QoS driven selection approach proposed by Zeng *et al.*[21] Our proposed selection algorithm considers only transactional properties, although it can be extended with local optimization on QoS properties. Our algorithm is influenced by the TQoS approach proposed by Haddad *et al.*[20] However, in contrast to the TQoS approach that performs only design-time selection of services, the proposed selection algorithm is dynamic. Moreover, TQoS approach considers only two risk levels or recoverability levels whereas our selection approach considers finer granularities of recoverability so as to enable flexible web service selection based on the recovery requirements. In contrast to all the above works, our proposed selection approach introduces an additional transactional property, named as *cancelable*.

2.2. *Transactional properties of web services*

Bhiri *et al.*[13] proposed a transactional approach to ensure failure atomicity of the composite web service using ATS model. They considered cancellation dependencies among services. Liu *et al.*[22] proposed a framework, FACTS, for fault-tolerant composition of transactional web services. It considered cancelable and compensatable behaviors of transactional web services. However, cancelable web services considered by both the above works, enable only internal cancellation of the service upon failure of another service running in parallel. Neila *et al.*[23] proposed a transaction model for reliable service composition and execution, namely, WS-SAGAS. However, WS-SAGAS model did not consider cancellation of long running services due to interruption of a transaction at run-time.

None of the existing literature for web service selection based on transactional properties consider external interruption of a long running transactions in a business application. However, this is important when dealing with frequent changes in business policies and user requirements. To address this issue, *cancelable* property of services has been introduced in the present work to enable cancellation of a long running service. Cancelable service offers flexibility to the user and minimizes the overhead in using the service.

3. Motivating Example

The drawbacks of the existing works that motivate the need for dynamic transaction aware web service selection can be better understood through the illustrative B2B application of vehicle purchase system (VPS). The VPS application involves business processes that integrate the services provided by multiple governmental departments such as birth and death registration (BDR) and income tax (IT) and business organizations such as vehicle dealers, banks, and vehicle manufacturers.

In order to avoid visiting multiple vehicle dealer sites to search for a suitable vehicle, a unified access point in the form of a web portal is provided to improve the user experience. The complex workflow of VPS involves simple activities as well as nested workflows. In VPS, the services need to be discovered on the fly based on the preferences of customers. For example, based on the outcome of enquiries with multiple vehicle dealers, the user decides a specific dealer and vehicle. The behavior of the individual services largely determines the reliability of the overall process. For example, if payment service is not compensatable, the outcome of entire purchase business process would be inconsistent upon failure of order processing service since, reimbursement of payment is not possible.

The control flow among the different activities of VPS as depicted in Fig. 1 is described using workflow patterns[24] such as sequence, parallel split, synchronization, exclusive choice, and simple merge. The sequence pattern enables a workflow activity after the completion of another activity (e.g. an order for the vehicle is placed after obtaining the payment). The parallel split (AND-Split) pattern describes a point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel. Synchronization pattern (AND-Join) describes a point in a workflow where multiple parallel activities converge into one single thread of control, thus synchronizing multiple threads (e.g. enquiring multiple vehicle dealers in parallel, for the availability of the required model before choosing a specific model). In contrast to parallel routing patterns, the conditional routing pattern, exclusive choice (XOR-Split) allows only one selected thread of control to be activated (e.g. after verifying the stock, if the required vehicle is available, a delivery note is prepared; otherwise, the vehicle is ordered from the manufacturer). The simple merge (XOR-Join) describes a point in the workflow where two or more alternative branches come together without synchronization (e.g. stock is verified either after getting payment for an order or after receiving new vehicles from the manufacturer).

Details of the vehicle such as make, model, color, and budget along with the personal details of the customer such as social security number (SSN), name, address, and permanent account number (PAN) are gathered through a portal interface.

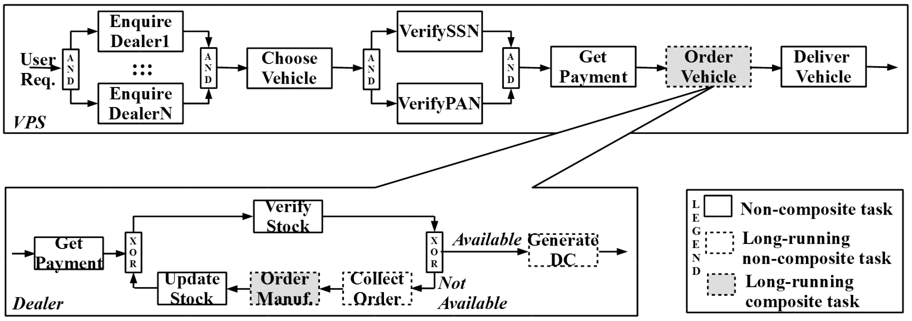


Fig. 1. Vehicle purchase system.

The availability of the vehicle with the specified details is ascertained from different dealers in parallel. The customer then chooses a particular dealer based on budget, color preferences, etc. SSN and PAN are mandatory for valid citizenship and sales tax payment. The customer details are verified with the BDR department. Similarly, PAN is verified with the IT department. If the customer details are found to be valid and the requested vehicle can be delivered by a specific dealer, payment is received from the customer. On successful payment, a purchase order is placed with the vehicle dealer chosen earlier. When the requested vehicle is ready, it is delivered to the customer.

When an order for a vehicle is received by the dealer, the order processing starts with receiving payment for the order. The stock is then checked for availability of the ordered vehicle. If the required vehicle is not available in the stock, it needs to be obtained from the manufacturer. The orders for the vehicles which are not available in the stock are accumulated for a day or a week before submitting them to the respective manufacturer. After receiving the vehicle from the manufacturer, the dealer updates the stock and generates the corresponding delivery challans. The vehicles are then delivered by the dealer to customers who have placed the order.

Few instances which emphasize the consideration of transactional properties of services involved in VPS application are listed below:

- The service corresponding to *GetPayment* activity should be *retriable* so as to restart it upon its failure, since, the vehicle dealer in general is interested in receiving the payment through different modes such as cheque, credit card, or cash.
- The service corresponding to the activity *OrderVehicle* should be *cancelable*; it is a long running composite service that may take more than a week to complete; it must be cancelled by interrupting the transaction involving it, when the user changes any of the preferences such as color and model, or if the user decides not to buy the vehicle at all. Otherwise, the vehicle that was ordered earlier is no longer of interest.
- Whenever the vehicle dealer cannot obtain the ordered vehicle from the manufacturer as it has become obsolete, the advance amount received from the customer must be refunded and hence, payment service should be *compensatable.*

The proposed approaches in this paper for transaction aware selection of services are intended to facilitate a reliable service provisioning by the vehicle sellers.

4. Proposed Approach for Transaction Aware Web Service Selection

An architecture has been proposed in the present work that enables dynamic selection and composition of web services based on their behavioral properties. The architecture is presented in the following subsection and the approach is discussed elaborately in the subsequent subsections.

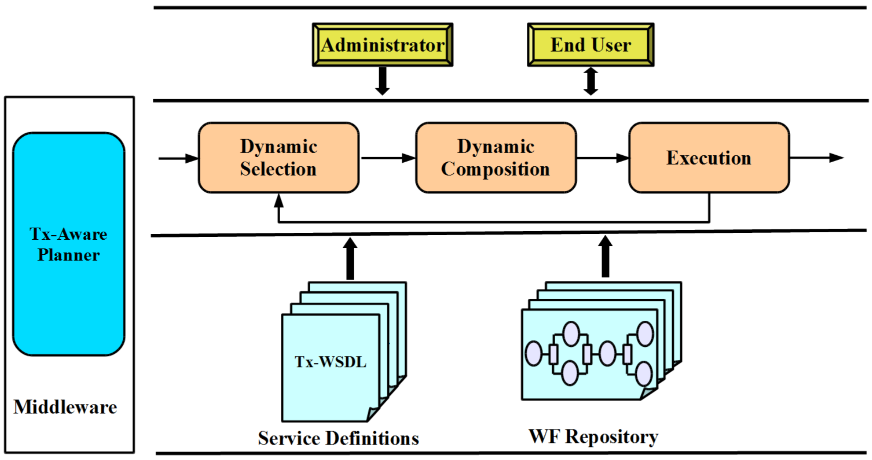


Fig. 2. System architecture.

4.1. *System architecture*

The architecture of the proposed system is depicted in Fig. 2. It takes two inputs: a set of business policies and user preferences. A workflow represents a set of business processes — simple activities or another nested workflow spanning multiple organizations and the control flow among them. For example, the workflow of VPS consists of simple activities such as payment and a nested workflow for order processing. The workflow is generated by business administrators as a template and is stored in the workflow repository. Business policies obtained from administrators are used to derive the transactional requirements of each activity in the principal workflow. The functional requirements are specified by the end users. If an activity is a workflow (say WF1) in itself, then the middleware service *Tx-AwarePlanner* determines the required transactional properties of the web services corresponding to each activity in WF1 based on the transactional requirements of WF1. Based on these requirements, appropriate web services are discovered by *Tx-AwarePlanner* from the service registry at run-time. It must be noted that the web service definitions should be deployed along with their transactional properties,[25] using extended WSDL named as, Tx-WSDL, in the service registry. The service endpoints of suitable services that are returned as output are used for invoking the services. Whenever business policies change during the execution of a long running service, the execution is cancelled by interrupting its transaction and the dynamic selection and composition are performed again to accommodate the changes.

4.2. *Cancelable web services*

The behavioral properties of a given web service (WS) have an impact on other services in a composition. The failure of a component service may require retrying the same service or undoing the effects of a completed component service, so that the execution of the composed service terminates in a consistent state. The extended transaction model[26,27] for multi-database systems, proposed the CPR model comprising of three transactional properties namely *compensatable* (cp), *pivot* (p), and *retriable* (r). A WS is compensatable if, upon its successful completion, its effects can be semantically undone. A retriable WS can be invoked a finite number of times in case of an internal failure. A WS is a pivot service, if it cannot be semantically undone on successful completion; if it fails, it leaves no trace. A pivot WS is neither compensatable nor retriable.

In general, B2B applications involve composition of several long running processes. The user preferences and business policies change very frequently according to the changing user requirements and business scenarios. Whenever such changes occur, transactions involving long running services need to be interrupted in order to incorporate the changes, since the execution of the service to completion with the earlier preferences or policies is no longer meaningful. The ability to cancel a service allows more flexibility in varying the preferences and business policies. Moreover, a cancelable service minimizes the overhead involved in using it since, at any point of time, it can be cancelled and the execution can be terminated. Hence, in addition to the properties supported by the CPR model, *cancelable* (cc) property has been introduced in this paper, to enable interruption of a transaction. In the present work, a cancelable web service with cancellation property is proposed as follows:

Definition 1 (Cancelable WS). A WS is cancelable if it can be cancelled by an external entity during its execution. Upon cancellation, it leaves no trace.

A cancelable WS enables cancellation recovery, upon its cancellation, by providing a cancellation logic. In general, execution of long-running transactions alone are interrupted externally. A long-running transaction is not constrained by the isolation property and it is likely that a certain amount of work would have already been completed at the time of interruption. If this completed portion is not compensated, the long-running service may not terminate in a consistent state. Hence, the cancellation logic may not only terminate the cancelled service, but may also compensate its completed portions. For example, *CollectOrder* service is a long running cancelable service whose cancellation cancels its execution and rolls back the updates for the received orders in the database.

4.3. *Deriving transactional properties of web services*

A WS may have more than one of the basic transactional properties (cp, p, r, cc). A successfully completed retriable service that is capable of semantically undoing its effects, upon failure of another service is compensatable retriable with properties cpr. Alternatively, a completed retriable service which is incapable of performing a rollback, is pivot retriable with properties pr. A cancelable service that is also compensatable can be rolled back after its successful completion and has properties $cpcc$. Alternatively, a cancelable service which cannot be compensated is pivot cancelable with properties *pcc*. A cancelable service which is guaranteed to succeed after a finite number of re-trials (necessitated by internal failures) is cancelable retriable. Such a cancelable retriable service exhibits *cpccr* properties, if it is capable of compensating itself, upon failure of another service, otherwise it is pivot and has properties *pccr*. Thus, the set of possible transaction properties for a WS is {*p*, *pr*, *cp*, *pcc*, *cpr*, *pccr*, *cpcc*, *cpccr*}. A transactional web service (TWS) is a simple (non-composite) web service with a valid transactional property. The transactional properties of a TWS describe its behavior. Hence, there can be eight different types of TWS based on its behavior.

Whenever several web services are composed, it is important to ensure that the composition is reliable and terminates in a consistent state. While composing TWSs of known behavior, it must be possible to predict the behavior of the composite web service (CWS) and check whether the CWS is valid and will result in a reliable execution. Transactional properties of the component services and the manner in which they are composed viz. in parallel or in sequence, influence the transactional properties of a CWS. A CWS with valid transactional properties that lead to a reliable execution is termed as a transactional composite web service (TCWS). A TCWS is atomic (a),[20] if it cannot be rolled back on successful completion. However, when one of its components fails or gets cancelled in the middle, all completed component services are compensated to reach a consistent state. For instance, in the aforementioned example of a vehicle purchase system, the composite service for VPS is atomic; after the requested vehicle is delivered to the customer, it cannot be compensated; when the long running VPS transaction is interrupted during the execution of its long running component service for order processing, due to changed user requirements, all the previous component services that have been completed must be compensated in order to achieve a consistent termination. When n services $S_1, S_2, \ldots, S_n$ are composed in sequence, the resultant TCWS, is *atomic*, if one of the following conditions is satisfied:

Condition 1: All $S_i, 2 \leq i \leq n$ are retriable.

Condition 2: If any $S_i, 2 \leq i \leq n$ is not retriable or is cancelable, all the preceding services $S_j, j<i$ must be compensatable.

When n services are composed in parallel, the resulting TCWS is atomic only when one of the services say S_i is a pivot or a cancelable service and other concurrent services $\mathrm{S}_j, j \neq i$ are compensatable and retriable. A TCWS is compensatable, if all of its component services are compensatable. In this paper, two more types of TCWS based on the cancelable property are proposed as follows:

Definition 2 (Cancelable TCWS). An atomic or a compensatable CWS is cancelable, if all of its component services are cancelable and will have transactional properties *acc* or *cpcc* respectively. Cancellation recovery is enabled by a cancelable TCWS.

Definition 3 (Retriable TCWS). An atomic or a compensatable or a cancelable CWS is retriable, if all of its component services are retriable and will exhibit transactional properties *ar*, *cpr*, or *accr*/*cpccr*, respectively. A retriable TCWS enables forward recovery.

For example, at the time of interrupting VPS transaction during the execution of *OrderVehicle* service that has transactional properties *atomic cancelable*, any of the services in the dealer workflow (say *CollectOrder* service) may be in execution. The cancellation logic of *OrderVehicle* invokes cancellation operation of the service in execution, *CollectOrder* and then invokes the compensation operation of the completed services such as *VerifyStock* and *GetPayment*.

The set of possible transactional properties for a TCWS is {*a*, *ar*, *cp*, *acc*, *cpr*, *accr*, *cpcc*, *cpccr*}. A CWS is non-atomic ($\bar{a}$), if it cannot be compensated after its successful execution and also, its completed component services cannot be undone after a failure or cancellation of one of the components in the middle. Thus, a non-atomic CWS is not reliable.

4.4. *Transaction aware web service selection*

A middleware service *Tx-AwarePlanner* has been proposed for dynamic web service selection based on its transactional capabilities. It is essential to ensure that the composition results in a TCWS, by selecting the component TWSs with appropriate transactional properties. For simplicity, only two component services S1 and S2 which may be simple or composite have been considered. The component services of the CWS may be executed sequentially (S1;S2) or concurrently (S1$\|$S2). When the composition of S1 and S2 results in a TCWS, it has property (*ar*), if both S1 ad S2 are retriable; the TCWS has property (*acc*), if both S1 ad S2 are cancelable; the TCWS has property (*cp*), if both S1 and S2 are compensatable.

In a sequential composition, if S1 has any of the transactional properties *p/a*, or *pcc/acc*, then S2 must have properties *pr/ar* or *cpr* to make the TCWS atomic (*a*). If S1 has any of the transactional properties *pr/ar*, or *pccr/accr*, then S2 must have transactional properties *pr/ar* or *cpr* to make the TCWS atomic retriable (*ar*). If S1 is compensatable (*cp/cpr/cpcc/cpccr*), it results in a TCWS irrespective of the transactional properties of S2; if S2 is non compensatable ($p/a, pr/ar, pcc/acc, pccr/accr$), then the TCWS is atomic (*a/ar/acc/accr*); if S2 is also compensatable ($cp, cpr, cpcc, cpccr$), then the resulting TCWS will also be compensatable.

In a parallel composition, if S1 has any of the transactional properties *p/a*, or *pcc/acc*, then S2 must be compensatable retriable *cpr* to make the TCWS atomic (*a*). If S1 is pivot retriable or atomic retriable *pr/ar*, then S2 must have transactional properties *pr/ar* or *cpr* to make the TCWS atomic retriable (*ar*). If S1 is cancelable retriable *pccr/accr*, then S2 must be compensatable retriable *cpr* to make the TCWS atomic retriable (*ar*). If S1 has any of the transactional properties ($cp, cpcc, cpccr$), it results in a compensatable TCWS ($cp/cpcc/cpr/cpccr$) only if

S2 is compensatable. If S1 is compensatable retriable (*cpr*), the resulting parallel composition is a TCWS irrespective of the transactional properties of S2; if S2 has any of the properties (*p*/*a*, *pcc*/*acc*), the TCWS is atomic (*a*); if S2 has properties *pr*/*ar* or *pccr*/*accr*, then the TCWS is atomic retriable (*ar*); the TCWS is compensatable (*cp*) or compensatable retriable (*cpr*), if S2 has properties *cp*/*cpcc* or *cpr*/*cpccr* respectively.

The transactional properties of the composite service were derived from the transactional properties of the component services and formally verified using a set of theorems.[28] These properties are presented in Table 1. This table is useful in two ways:

(1) To verify whether a specific composition results in a TCWS, when the transactional requirements of its component services are known.
(2) To find out the transactional requirements of the individual component services of a TCWS whose transactional property is known.

Table 1. Transactional properties of TCWS.

S1	S2	S1;S2	S1||S2
p/a	p/a	ā	ā
	pcc/acc	ā	ā
	pr/ar	a	ā
	pccr/accr	ā	ā
	cp	ā	ā
	cpcc	ā	ā
	cpr	a	a
	cpccr	ā	ā
pcc/acc	p/a	ā	ā
	pcc/acc	ā	ā
	pr/ar	a	ā
	pccr/accr	ā	ā
	cp	ā	ā
	cpcc	ā	ā
	cpr	a	a
	cpccr	ā	ā
pr/ar	p/a	ā	ā
	pcc/acc	ā	ā
	pr/ar	ar	ar
	pccr/accr	ā	ā
	cp	ā	ā
	cpcc	ā	ā
	cpr	ar	ar
	cpccr	ā	ā
pccr/accr	p/a	ā	ā
	pcc/acc	ā	ā
	pr/ar	ar	ā
	pccr/accr	ā	ā
	cp	ā	ā
	cpcc	ā	ā
	cpr	ar	ar
	cpccr	ā	ā
cp	p/a	a	ā
	pcc/acc	a	ā
	pr/ar	a	ā
	pccr/accr	a	ā
	cp	cp	cp
	cpcc	cp	cp
	cpr	cp	cp
	cpccr	cp	cp
cpcc	p/a	a	ā
	pcc/acc	acc	ā
	pr/ar	a	ā
	pccr/accr	acc	ā
	cp	cp	cp
	cpcc	cpcc	cpcc
	cpr	cp	cp
	cpccr	cpcc	cpcc
cpr	p/a	a	a
	pcc/acc	a	a
	pr/ar	ar	ar
	pccr/accr	ar	ar
	cp	cp	cp
	cpcc	cp	cp
	cpr	cpr	cpr
	cpccr	cpr	cpr
cpccr	p/a	a	ā
	pcc/acc	acc	ā
	pr/ar	ar	ā
	pccr/accr	accr	ā
	cp	cp	cp
	cpcc	cpcc	cpcc
	cpr	cpr	cpr
	cpccr	cpccr	cpccr

An approach is proposed for deriving the transactional requirements of services involved in the principal workflow from the business policies specified by the administrator in Sec. 4.4.1. For a workflow activity that itself represents another workflow with a given transactional requirement, the transactional requirements of its component activities must be derived. An algorithm for this is presented in Sec. 4.4.2.

4.4.1. *Extraction of transactional requirements from business policies*

For the given workflow activity, all the Tx-WSDL documents of available services are discovered from the registry. Each of these Tx-WSDL documents is parsed to extract the transactional capabilities. The transactional requirements extracted from business policies along with the functional requirements are then matched with the capabilities of the services to discover a suitable service. Though the business policies represent the transactional requirements of some of the services, the recoverability level plays an important role in selecting a TWS. The recoverability level describes the degree to which a service may be recovered in case of a failure or an interruption. In our earlier work,[29] TWSs were grouped into different levels of recoverability based on their recovery cost. The recoverability level and the corresponding transactional properties of a TWS/TCWS are tabulated in an increasing order of their recovery cost, as shown in Table 2.

If the administrator deems a particular service to be very important, then the service should be guaranteed to succeed and it should have the property r. Business policies obtained from the administrator, through a questionnaire and the required recoverability level, together abstract the transactional requirements or properties. The business analyst is provided with recoverability options for an activity, depending on the transactional requirement of the previous activity so that the resultant composite service after all the matching services are discovered and composed, is at least atomic. While obtaining the recoverability level for an activity, it is restricted to only certain options such that any of those choices does not result in a non-atomic composition. For example, if the transaction property of a service assigned to an activity is extracted as *cp*, the service associated with the next sequential activity can be assigned any possible transactional property to yield a valid TCWS (see Table 1). Alternatively, for a service assigned to an activity composed in parallel

Table 2. Recoverability levels.

Recoverability level	Properties
1	Pivot/atomic (p/a)
2	Compensatable (cp)
3	Cancelable (pcc/acc)
4	CompensatableCancelable (cpcc)
5	Retriable (pr/ar)
6	CompensatableRetriable (cpr)
7	CancelableRetriable (pccr/accr)
8	FullyRecoverable (cpccr)

Table 3. Valid recoverability levels for deriving a TCWS.

Tx-Property	Recoverability Level of Next Service	
	Composed in Sequence	Composed in Parallel
p/a	5, 6	6
pcc/acc	6	6
pr/ar	5, 6	5, 6
pccr/accr	5, 6	5
cp	1, 2, 3, 4, 5, 6, 7, 8	2, 4, 6, 8
cpcc	1, 2, 3, 4, 5, 6, 7, 8	2, 4, 6, 8
cpr	1, 2, 3, 4, 5, 6, 7, 8	1, 2, 3, 4, 5, 6, 7, 8
cpccr	1, 2, 3, 4, 5, 6, 7, 8	2, 4, 6, 8

with another activity having compensatable property, assigning *p/a*, *pcc/acc*, *pr/ar*, or *pccr/accr* property will result in a non-atomic composition. The non-atomic compositions shown in Table 1 are not desirable and hence, such unreliable compositions are avoided. For each transactional property of a service assigned to a workflow activity, the possible transactional properties of the service to be assigned to the subsequent activity are derived from Table 1 and their respective recoverability levels are tabulated in Table 3. For example, given a pivot service, the next sequentially composed service must be either retriable (pr/ar) or compensatable retriable (cpr) to result in a consistent outcome (see Table 1). The corresponding recoverability levels from Table 2 are tabulated in Table 3.

4.4.2. *Extracting transactional requirements of component services*

Whenever the current workflow contains a component that by itself is a composite process, it involves another workflow comprising of multiple activities. The middleware service, *Tx-AwarePlanner* assigns transactional requirements for the component activities of the composite workflow, in such a way that the required transactional property of the composition is satisfied, using Algorithm 4.1. The algorithm takes a workflow, WF and its transaction property, *Reqd_TxP* as input. In a WF, the transactional property of any component activity depends on the transaction property of the overall WF and the transactional property of the composition of all the previous activities. The transactional requirements are assigned from left to right in sequential patterns and from top to bottom in split patterns. A workflow of n activities can be visualized as a composition of a composite process involving $n-1$ activities and n^{th} activity as illustrated in Fig. 3. At any point of time, composition involves two services. Hence, the workflow is structured internally as a binary tree. In each node of the tree representing a workflow activity, the type of the activity (*Type*), its transactional requirement (*TxP*), and the name of the service to be selected (*sName*) are stored. The different types of activity are *simple*, *sequence*, *parallel*, and *composite*. The *simple* activities are non-composite whereas composite activities are represented by workflows. Activity types such as *sequence* and *parallel* represent a sequentially composed process and a concurrently composed process

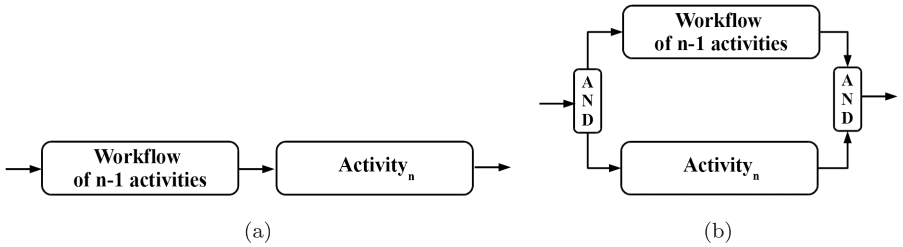


Fig. 3. Transactional property assignment — Step n. (a) Sequential composition and (b) parallel composition.

respectively. The *Set_Seq* (*Set_Par*) in the algorithm represents a set of properties that can be assigned to the left composition or activity of WF that is sequentially (concurrently) composed. The function *any*(*set*) returns any one of the transactional properties from *set*. Based on the required transactional property of a TCWS, the members of the *Set_Seq* and *Set_Par* are different in accordance with Table 1.

The algorithm processes each activity of the WF and terminates when the transactional property of the WF is assigned. When an activity of a WF is another WF that is either sequentially or concurrently composed, the algorithm is invoked recursively. For a simple or composite activity, any of the transactional properties from the respective set is assigned. Based on the transactional property assigned to the left composition/activity, a suitable property is assigned to the right activity that would result in the required property for their composition. For each type of transaction property, the algorithm has three parts as enumerated below:

(1) Assign transactional property to the left composition/activity of workflow, W.
(2) Assign transactional property to the right activity of W based on step 1.
(3) Assign transactional property to the current workflow, W.

Algorithm 4.1 AssignTxP(WF, Reqd_TxP)

```
Input    WF          {Workflow}
         Reqd_TxP    {Required transactional property of WF}
Output   TCWS        {WF with transactional properties assigned to all of its activities}
begin
Initialize Set_Seq
Initialize Set_Par
while WF.Type ≠ "simple" AND WF.Type ≠ "composite" AND WF.TxP == NULL do
  if WF.left.Type == "sequence" then
    AssignTxP(WF.left, any(Set_Seq))
  else if WF.left.Type == "parallel" then
    AssignTxP(WF.left, any(Set_Par))
  else if WF.left.Type == "simple" OR WF.left.nodeType == "composite" then
    {Assign any property from respective set to the left composition/activity of WF}
    if WF.Type == "sequence" then
      WF.left.TxP ← any(Set_Seq)
    else if WF.Type == "parallel" then
```

```
            WF.left.TxP ← any(Set_Par)
        end if
    end if
    {Initialize rightTxP based on the property assigned to left composition/activity}
    Initialize rightTxP
    if WF.right.Type == "parallel" OR WF.right.Type == "sequence" then
        AssignTxP(WF.right, any(rightTxP))
    else
        {Assign any property from rightTxP to the right activity of WF}
        WF.right.TxP ← any(rightTxP)
    end if
    WF.TxP ← Reqd_TxP {Assign TxP to WF}
end while
end
```

Algorithm 4.1 involves traversing all the nodes of the binary tree used to represent the workflow WF. The workflow activities (say n of them) represent the leaf nodes of the binary tree. The binary tree which is constructed by the algorithm has nodes of degree 0 (non-composite services in leaf nodes) or 2 (composite services composed from two component services) only. Thus, the total number of nodes in the binary tree is $n + (n + 1)$. Hence, the time complexity of the algorithm would be $O(n)$.

4.5. *Performance analysis*

In order to evaluate the performance of our selection approach, experiments were conducted by implementing it on a 2.4 GHz Intel Core2 Duo processor machine with 4 GB RAM, operating with Ubuntu, using J2EE middleware and Netbeans IDE. The clients, domain services of different service providers and middleware services are deployed in different machines connected through an intranet. In order to test the heterogeneity among the WS, the services have been created with application servers such as Glassfish, Apache Tomcat 6.0, JBoss Application Server 5.1, database servers like MySQL 5.1, Derby and operating systems like Windows and Linux. OpenUDDI and NovellUDDI browsers have been used to implement the service registry. Three experiments were conducted with the following objectives:

(1) To observe the performance of the proposed dynamic selection algorithm by increasing the service registry size as well as the workflow size.
(2) To find the impact of the transactional property or type of the selected service on the performance of the proposed selection algorithm.
(3) To compare the proposed *run-time* selection approach with *design-time* selection approach by adapting these approaches in the VPS prototype.

4.5.1. *Experiment* 1

Five arbitrary workflows with 5, 15, 25, 35 and 45 activities were constructed. The services for the activities of a workflow were selected from the registry at runtime

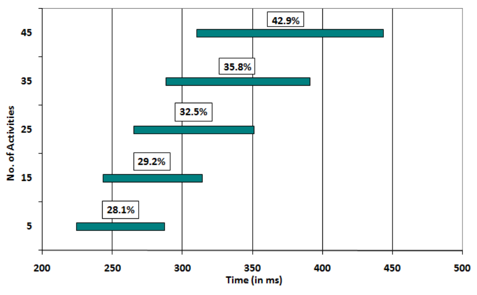
Table 4. Performance analysis of selection algorithm.

		Number of Services Per Activity					*%increase*
		25	50	100	150	200	
Number of activities	5	224	237	244	270	287	28.1
	15	243	257	296	307	314	29.2
	25	265	292	309	320	351	32.5
	35	288	307	321	345	391	35.8
	45	310	339	366	410	443	42.9
	%increase	38.4	43	50.0	51.9	54.4	

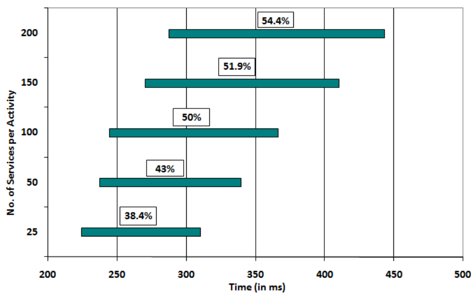
Note: Selection time in milliseconds.

according to the proposed approach of transaction-aware selection. The number of suitable services for each of the activity in the workflow was varied from 25 to 200. For each of the five workflows, the execution times were measured for 10 sample executions and the maximum values are shown in Table 4. Our selection algorithm takes only 0.443 s to select the services, for each of the 45 activities of a workflow from a registry containing 9000 services, with an average of 200 services per activity. It emerges that for a 72-fold increase in the service registry size (from 125 to 9000), the selection effort increases merely by a factor of 2 (0.2–0.4 s).

From this experiment, it is also observed that, the execution time increases with the increase in the number of activities of a workflow as well as the number of available services for selection. Figure 4(a) shows the impact of increasing the number of available services per activity on execution time taken by the selection approach. The number of services available for selection varies uniformly from 25 to 200 for each workflow. The execution time increases by 34% on an average, for an eight-fold increase (700%) in the number of available services per activity from 25 to 200. The impact of increase in the number of activities in a workflow on the execution time is shown in Fig. 4(b). The number of activities uniformly varies from 5 to 45, for a given number of services per activity. For instance, with 200 services available for an activity, it is observed that, for a nine fold increase (800%)



(a)



(b)

Fig. 4. Impact analysis on selection time. (a) Impact of service registry size and (b) impact of workflow size.

in the number of activities from 5 to 45, the execution time increases only by 54%. The increase in execution time for a 800% increase in the number of activities is observed to be 48% on an average.

4.5.2. *Experiment* 2

A workflow with 25 activities was constructed and the performance of the selection approach that results in a TCWS with the required transactional property among the possible eight, was analysed. The experiment was carried out by increasing the number of services for each activity from 25 to 200 with 3 to 25 services per each transactional property. The experiment was repeated 10 times for each of the eight properties of TCWS and the average is computed. The computed values are shown in Table 5. The execution time incurred by the proposed middleware service for selection, *Tx-AwarePlanner* ranges from 319 to 349 ms for 200 services per activity. This clearly demonstrates that the performance of the proposed selection approach is uniformly the same irrespective of the transactional property of the required TCWS.

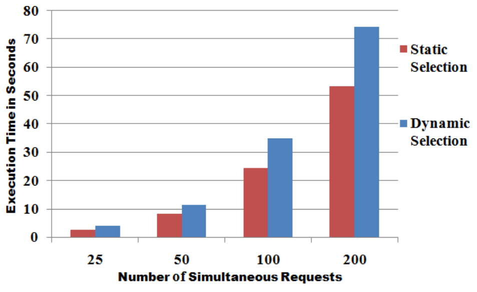
4.5.3. *Experiment* 3

In order to compare dynamic and static selection approaches, both of them were adapted in VPS prototype and throughput of the application in terms of number of requests completed per second was measured. For dynamic selection, the workflow template for the VPS was used. The web services registry has 25 services available to choose from, for each workflow activity. For design-time selection, instead of the workflow template, a concrete workflow where all the services were discovered statically for each of the activities was considered. The databases accessed by each of the services in VPS contained 10,000 records on an average. The experiment was performed by varying the number of simultaneous user requests from 25 to 200. Each execution was repeated ten times and the average values have been used in plotting the graphs shown in Fig. 5. It can be seen that, the average execution time of the application is found to be 40% more with dynamic selection as compared

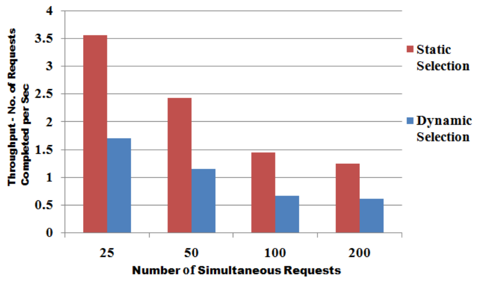
Table 5. Selection time analysis for various types of TCWS.

		Number of Services in WS Registry							
		625	1250	1875	2500	3125	3750	4375	5000
Transactional property	a	256	271	275	281	285	290	302	319
	cp	265	280	285	296	303	309	319	321
	cc	275	283	287	290	297	304	317	323
	cpcc	268	273	275	280	297	304	317	323
	r	271	292	295	300	308	311	321	326
	cpr	275	281	288	290	312	327	331	338
	ccr	278	273	280	283	311	328	336	344
	cpccr	280	284	290	291	314	331	338	349

Note: Selection time in milliseconds.



(a)



(b)

Fig. 5. Transaction aware dynamic selection versus static selection. (a) Execution effort and (b) throughput.

to static selection of services. In addition, the average throughput of VPS with dynamic selection is approximately half of that is associated with static selection of services. The degradation in performance of VPS with dynamic selection approach is due to the fact that the selection effort is a part of execution effort in contrast to the static selection process where the services are selected in design-time itself.

The additional overhead associated with the run-time selection can be justified due to the advantages offered by this dynamic approach, such as considering the present state of the services and flexibility for the user to specify preferences based on the execution of previous services. Further, there is also less increase (34%) in selection time even with a substantial increase in the number of available services (700%).

5. Conclusion

In the context of service-oriented B2B applications, transaction aware web service selection is a prerequisite to achieve reliable execution of compositions. In this work, a web service selection approach supporting transaction aware WS composition has been presented. The component services are selected by matching their functional as well as transactional capabilities with the user's preferences on functional and transactional requirements. The present work makes two key contributions. First, an approach has been proposed for selecting the suitable services at run-time based on their transactional capabilities. Second, cancelable services have been proposed. Cancelable property enables external interruption of long running business transactions resulting in reduced overhead.

The run-time selection algorithm has been experimented for the vehicle purchase system prototype and is compared with its static selection counterpart. Since the runtime selection approach introduces the selection overhead during execution, the overall performance of the application is decreased by 40% in terms of execution time and by 50% in terms of throughput. However, even when workflow size is increased by 800% and service registry size is increased by 700%, the performance of the proposed selection algorithm decreases only by 54% which has rather marginal impact on the overall performance of the application. In comparison with the existing static selection approach, the proposed dynamic selection approach is worthwhile for the advantages and flexibility that it offers.

The proposed dynamic selection approach is suitable for B2B applications where the user preferences and business policies frequently change, since the user can cancel the service execution at any point of time as well as user preferences can be decided based on the outcome of a previous service execution. The dynamic composition of services selected using the proposed transaction aware selection approach is guaranteed to result in a reliable execution.

Our algorithm can be easily extended to select services with the best QoS properties locally, in addition to satisfying the transactional requirements globally.

References

1. M. P. Singh and M. N. Huhns, *Service Oriented Computing: Semantics, Processes, Agents* (John Wiley and Sons Ltd., England, UK, 2005).
2. T. Erl, *Service Oriented Architecture: Concepts, Technology, and Design* (Prentice Hall PTR, New Jersey, USA, 2005).
3. C. Roberto, M. Jean-Jacques, R. Arthur and W. Sanjiva, Web services description language (WSDL), Version 1.5. 2007.
4. UDDI. Universal Description Discovery and Integration. 2004. http://www.uddi.org/pubs/uddi-v3.htm.
5. D. Schahram and S. Wolfgang, A survey on web services composition, *Int. J. Web and Grid Serv.* **1**(1) (2005) 1–30.
6. J. Rao and X. Su, A survey of automated web service composition methods, in *Proc. Conf. Semantic Web Services and Web Process Composition SWSWPC* (Springer-Verlag, Berlin, Heidelberg, 2004), pp. 43–54.
7. K. Rajesh, K. Ferhat and H. Glitho, A business model for dynamic composition of telecommunication web services, *IEEE Commun. Magazi.* **45**(7) (2007), 36–43.
8. V. Agarwal, G. Chafle, S. Mittal and B. Srivastava, Understanding approaches for web service composition and execution, in *ACM Conf. COMPUTE 2008* (2008), pp. 1:1–1:8.
9. F. Mustafa and T. L. McCluskey, Dynamic web service composition, in *Int. Conf. Computer Engineering and Technology* (2009), pp. 463–467.
10. P. P. W. Chan and M. R. Lyu, Dynamic web service composition: A new approach in building reliable web service, in *22nd Int. Conf. Advanced Information Networking and Applications* (2008), pp. 20–25.
11. L. Zeng, A. H. Ngu, B. Benatallah, R. Podorozhny and H. Lei, Dynamic composition and optimization of web services, *Int. J. Distrib. Parallel Databases* **24** (2008) 45–72.
12. F. Casati and M. Shan, Dynamic and adaptive composition of e-services, *Inform. Syst.* **26**(3) (2001) 143–163.
13. S. Bhiri, O. Perrin and C. Godart, Ensuring required failure atomicity of composite web services, in *Proc. 14th Int. Conf. World Wide Web* (*WWW2005*) (2005), pp. 138–147.
14. S. Bhiri, O. Perrin and C. Godart, Transactional patterns for reliable web services compositions, in *ACM Int. Conf. Web Engineering ICWE* (2006), pp. 137–144.

15. F. Montagut, R. Molva and S. T. Golega, Automating the composition of transactional web services, *Int. J. Web Serv. Res.* **5**(1) (2008) 24–41.
16. F. Montagut, R. Molva and S. T. Golega, The pervasive workflow: A decentralized workflow system supporting long-running transactions, *IEEE Trans. Syst., Man, Cybernet. Part C* **38**(3) (2008) 319–333.
17. W. Gaaloul, K. Gaaloul, S. Bhiri, A. Haller and M. Hauswirth, Log-based transactional workflow mining, *Int. J. Distrib. Parallel Databases* **25** (2009) 193–240.
18. P. F. Pires, M. R. F. Benevides and M. Mattoso, Building reliable web services composition, *Web Databases Web Serv.* **2593** (2003) 59–72.
19. L. Li, C. Liu and J. Wang, Deriving transactional properties of composite web services, in *IEEE Int. Conf. Web Services* (*ICWS*) (2007), pp. 631–638.
20. J. E. Haddad, M. Manouvrier and M. Rukoz, TQoS: Transactional and QoS-aware selection algorithm for automatic web service composition, *IEEE Trans. Serv. Comput.* **3** (1) (2010) 73–85.
21. L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam and H. Chang, QoS-aware middleware for web services composition, *IEEE Trans. Software Eng.* **30**(5) (2004) 311–327.
22. A. Liu, Q. Li, L. Huang and M. Xiao, FACTS: A framework for fault-tolerant composition of transactional web services, *IEEE Trans. Serv. Comput.* **3**(1) (2010) 46–59.
23. B. L. Neila, K. Takashi and Y. Haruo, WS-SAGAS: Transaction model for reliable web-services composition specification and execution, *DBSJ Lett.* **2**(2) (2001) 1–4.
24. W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski and A. P. Barros, Workflow patterns, *Distrib. Parallel Databases* **14** (2003) 5–51.
25. K. Rajaram, A. Adiththan and C. Babu, Tx-policy: Transactional policies for reliable web service composition, in *Proc. Int. Conf. Workshop on Emerging Trends in Technology* (2011), pp. 738–743.
26. S. Mehrotra, R. Rastogi, H. F. Korth and A. Silberschatz, A transaction model for multidatabase systems, in *12th Int. Conf. Distributed Computing Systems* (*ICDCS92*) (1992), pp. 56–63.
27. A. Zhang, M. Nodine, B. Bhargava and O. Bukhres, Ensuring relaxed atomicity for flexible transactions in multidatabase systems, *ACM SIGMOD Record* **23** (1994) 67–78.
28. R. Kanchana and B. Chitra, Reliable compositions using cancelable web services, *ACM SIGSOFT Software Eng. Notes* **39**(1) (2014) 1–6.
29. R. Kanchana, B. Chitra and A. Arun, Specification of transactional requirements for web services using recoverability, *Int. J. Inform. Technol. Web Eng.* **8**(1) (2013) 51–65.