QoS-Aware Middleware for Web Services Composition*

Liangzhao Zeng[1] , Boualem Benatallah[2], Anne H.H. Ngu[3],
Marlon Dumas[4], Jayant Kalagnanam[1], Henry Chang[1]

[1] IBM T.J. Watson Research Center, Yorktown Heights NY, USA
{lzeng,jayant,hychang}@us.ibm.com

[2] School of Computer Science & Engineering, University of New South Wales, Sydney, Australia
boualem@cse.unsw.edu.au

[3] Department of Computer Science, Southwest Texas State University, San Marcos TX, USA
angu@swt.edu

[4] Centre for IT Innovation, Queensland University of Technology, Brisbane, Australia
m.dumas@qut.edu.au

Abstract

The paradigmatic shift from a Web of manual interactions to a Web of programmatic interactions driven by Web services is creating unprecedented opportunities for the formation of online Business-to-Business (B2B) collaborations. In particular, the creation of value-added services by composition of existing ones is gaining a significant momentum. Since many available Web services provide overlapping or identical functionality, albeit with different Quality of Service (QoS), a choice needs to be made to determine which services are to participate in a given composite service. This paper presents a middleware platform which addresses the issue of selecting Web services for the purpose of their composition in a way that maximizes user satisfaction expressed as utility functions over QoS attributes, while satisfying the constraints set by the user and by the structure of the composite service. Two selection approaches are described and compared: one based on local (task-level) selection of services, and the other based on global allocation of tasks to services using integer programming.

Keywords: D.2.2.c Distributed/Internet based software engineering tools and techniques, H.3.5 Web-based services

*This paper is an extended and revised version of reference [36]

1 Introduction

Web services are self-described software entities which can be advertised, located, and used across the Internet using a set of standards such as SOAP, WSDL, and UDDI [10]. Web services encapsulate application functionality and information resources, and make them available through programmatic interfaces, as opposed to the interfaces typically provided by traditional Web applications which are intended for manual interactions.

The emergence of Web services (e.g., for order procurement, customer relationship management, finance, accounting, human resources, supply chain and manufacturing) has created unprecedented opportunities for organizations to establish more agile and versatile collaborations with other organizations. Widely available and standardized Web services make it possible to realize Business-to-Business Interoperability (B2Bi) by inter-connecting Web services provided by multiple business partners according to some business process: a practice known as Web Services Composition [8, 4]. For example, a high level financial management Web service can be created by composing more specialized Web services for payroll, tax preparation, and cash management.

Our work aims at advancing the current state of the art in technologies for Web service composition, by addressing the following key issues:

- **Quality of Service (QoS) modeling.** In the presence of multiple Web services with overlapping or identical functionality, users unavoidably discriminate Web service offerings based on their QoS. QoS is a broad concept that encompasses a number of non-functional properties such as price, availability, reliability, and reputation [26]. These properties apply both to stand-alone Web services and to Web services composed of other Web services (i.e., *composite Web services*). In order to reason about Web services, a framework is needed which captures their QoS from a user's perspective. Such framework must take into account the fact that QoS involves multiple dimensions, and the fact that the QoS of composite services is determined by the QoS of its underlying *component services*.

- **QoS-driven composition of Web services.** When creating a composite service, and subsequently when executing it following a user request, the number of component services involved in this composite service may be large, and the number of Web services from which these component services are selected is likely to be even larger. On the other hand, the QoS of the resulting composite service executions is a determinant factor to ensure customer satisfaction, and different users may have different requirements and preferences regarding this QoS. For example, a user may require to minimize the execution duration while satisfying certain constraints in terms of price and reliability, while another user may give more importance to the price than to the execution duration. A QoS-aware approach to service composition is therefore

needed, which maximizes the QoS of composite service executions by taking into account the constraints and preferences set by the users.

- **Composite service execution in a dynamic environment.** Web services operate autonomously within a rapidly changing environment. As a result, their QoS may evolve relatively frequently, either because of internal changes or because of changes in their environment (i.e., higher system loads). In particular, during the execution of a composite service, some component services may update their QoS properties on-the-fly, others may become unavailable, and still others may emerge. Consequently, approaches where Web services are statically composed are inappropriate. Instead, a dynamic composition approach is needed, in which runtime changes in the QoS of the component services are taken into account.

In this paper, we present AgFlow [37, 36, 35]: a middleware platform that enables the quality-driven composition of Web services. In AgFlow, the QoS of Web services is evaluated by means of an extensible multi-dimensional QoS model, and the selection of component services is performed in such a way as to optimize the QoS of the composite service executions. Furthermore, the AgFlow adapts to changes that occur during the execution of a composite service, by revising the execution plan in order to optimize the QoS given a set of user requirements and a set of candidate component services. The salient features AgFlow are:

- A *multi-dimensional QoS model* which captures non-functional properties that are inherent to Web services in general, e.g. availability and reliability. This model defines a number of QoS properties and methods for attaching values for these properties in the context of both stand-alone and composite Web services. The model is intended to be extended in order to fit the purposes of specific application domains.

- Two alternative *QoS driven service selection approaches* for composite service execution: one based on local optimization and the other on global planning. The local optimization approach performs optimal service selection for each individual task in a composite service without considering QoS constraints spanning multiple tasks and without necessarily leading to optimal overall QoS. The global planning approach on the other hand considers QoS constraints and preferences assigned to a composite service as a whole rather than to individual tasks, and uses integer programming to compute optimal plans for composite service executions.

- An *adaptive execution engine* which reacts to changes occurring during the execution of a composite service (e.g., component services that become unavailable or change their predicted QoS), by re-planning the execution in order to ensure that the QoS is optimal given the available information about the component services.

The remainder of the paper is organized as follows. Section 2 provides an overview of the AgFlow system and basic concepts of the underlying service composition model. Section 3 describes the proposed service quality model. In Section 4, two alternative service selection approaches are presented and compared. An implementation of the service composition and service selection model is then presented in Section 5, and experimental results are documented in Section 6. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2 Preliminaries

In this section, the AgFlow's system architecture is presented first, then some basic concepts and definitions are explained.

2.1 System Architecture

The architectural diagram of the AgFlow system is presented in Figure 1. There are three distinct components in the AgFlow system, namely, *Web services*, *service broker*, and *service composition manager*.

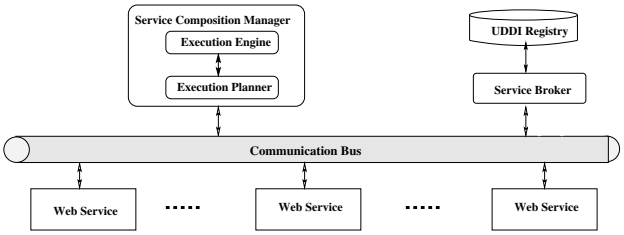


Figure 1: AgFlow's architecture

The service broker allows providers to register their service descriptions in an UDDI registry. A service description contains meta-data that describe, among others, the meaning, type, content, capability, QoS and location of a Web service. By searching the repository, the service broker can answer queries about which Web services can execute a given task. The UDDI registry also contains service ontologies that specify a common language for communications among Web services providers and requesters.[1]

The service composition manager contains an execution engine. It contacts the service broker to locate appropriate Web services, selects and integrates Web services to execute composite services. The service composition manager also provides an interface which allows users to access the AgFlow

[1]Note that our proposal does not deal with the issue of searching multiple federated UDDI registries. For a discussion on the issues raised by federated UDDI registries, the reader is referred to [32].

system. There are two types of users in AgFlow: composite service designers and end users. Designers can define templates of composite services using a graphical design tool. End users can create, control and monitor composite services.

2.2 Service Ontologies and Service Description

A service ontology (see Figure 2) specifies a common language agreed by a community (e.g., automobile industry). It defines a terminology that is used by all participants in that community. Within a community, service providers describe their services using the terms of the community's ontology, while service requesters use the terms of the ontology to formulate queries over the registry(ies) of the community.

Concretely, a service ontology specifies a *domain* (e.g., `Automobile`, `Healthcare`, `Insurance`), a set of *synonyms*, used to facilitate flexible search for the domain (e.g, the domain `Automobile` may have synonyms like `Car`), and a set of *service classes* that are used to define the properties of services. A service class is further specified by its *attributes* and *operations*. For example, the attributes of a service class may include access information such as URL. Each operation is specified by its name and signature (i.e., inputs and outputs). A service ontology also specifies a *service quality model* that is used to describe non-functional properties of services, e.g., execution duration of an operation. The service quality model consists of a set of *quality dimensions* (or *criteria*). For each quality criterion, there are three basic elements: its definition, the service elements (e.g., services or operations) to which it is related, and how to compute or measure the value of the criteria. The service quality model is presented in section 3.

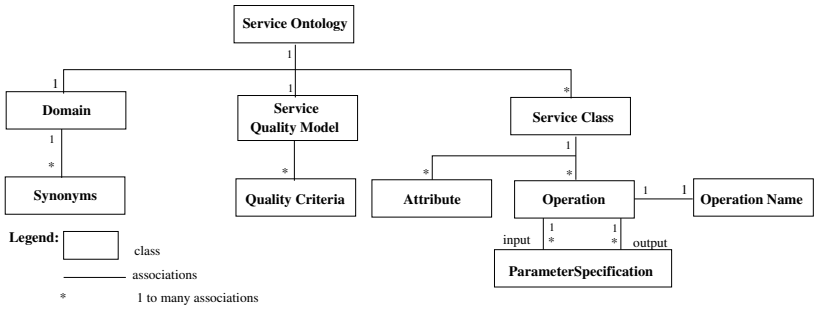


Figure 2: UML class diagram for service ontologies

Service ontologies are organized in a tree structure. The root service ontology can be used by all the communities. Child nodes automatically inherit the elements of their parents including service classes, and can extend service classes that are used in their own communities. Service ontologies can be used to facilitate the specification of composite services. Specifically, in order to participate in a composite service, providers need to publish their Web services as advertisements in the service repository associated to the composite service. There are two important elements in a service

description:

- **Service ontology and service class.** A Web service provider needs to specify which service ontology is used and which service classes are supported. For example, a travel service provider may specify that it supports the service ontology `Trip-planning` and the service class `FlightTicketBooking`. The service ontology specifies the concepts and terminology used in the service description, and service class describes the capabilities (e.g., operations) of Web services and how to access them.

- **Service Level Agreements (SLA).** An SLA defines the terms and conditions of service quality that a Web service delivers to service requesters. The major constituent of an SLA is the QoS information. There are a number of criteria (e.g., execution duration, reliability) that contribute to a Web service's QoS in a SLA as discussed later in the paper. Some Web service providers publish QoS information in SLAs. Other Web service providers may not publish their QoS information in their service descriptions for confidential reasons. In this case, service providers need to provide interfaces that only authorized requesters can use to query the QoS information.

2.3 Composite Service Specifications

A composite service is specified as a collection of *generic service tasks* described in terms of service ontologies and combined according to a set of control-flow and data-flow dependencies. AgFlow uses statecharts [16] to represent these dependencies.[2] This choice is motivated by several reasons. First, statecharts possess a formal semantics, which is essential for analysing composite service specifications. Second, statecharts are a well-known and well-supported behavior modeling notation, following their integration into the Unified Modeling Language (UML). Finally, statecharts offer most of the control-flow constructs found in existing process modeling languages (branching, concurrent threads, structured loops) and they have been shown to be suitable for expressing typical control-flow dependencies [11]. Hence, it is possible to adapt the QoS-driven service selection mechanisms developed using statecharts, to fit other alternative languages such as the Business Process Execution Language for Web Services (BPEL4WS) [2].

A statechart is made up of states and transitions. Transitions of a statechart are labeled with events, conditions, and operations. States can be *basic* or *compound*. Basic states (also called *tasks* in the sequel) are labeled with an operation name of a given service class (which is defined in a service ontology). Intuitively, when the basic state is entered, the operation that labels this state is invoked over one of the services belonging to the designated service class.

[2]In the remainder the paper, we use the terms *composite service specification* and *statechart of a composite service* interchangeably.

Compound states on the other hand provide means to structure the statechart into regions, and to express concurrent execution of regions. Compound states come in two flavors: OR-states and AND-states. An OR-state contains a single region whereas an AND-state contains several regions (separated by dashed lines) which are intended to be executed concurrently. Accordingly, OR-states are used as a grouping mechanism for modularity purposes, while AND-states are used to express concurrency: they encode a fork/join pair. The initial state of a statechart is denoted by a filled circle, while the final state is denoted by two concentric circles: one filled and one unfilled.

A simplified statechart W specifying a "Travel Planner" composite Web service is depicted in Figure 3. In this composite service, a search for attractions is performed in parallel with a flight and an accommodation booking. After these searching and booking operations are completed, the distance from the hotel to the accommodation is computed, and either a car or a bike rental service is invoked. Note that when two transitions stem from the same state (e.g., t_4), they denote a conditional branching, and the transitions should therefore be labeled with disjoint conditions.

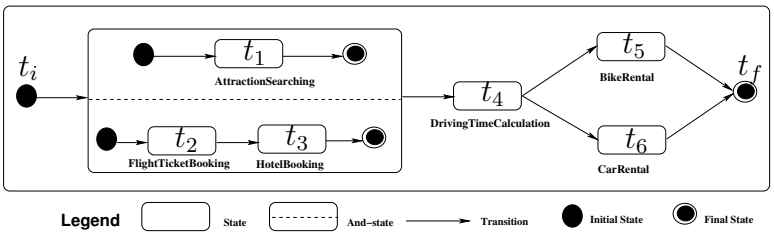


Figure 3: Statechart of a "Travel Planner" composite service.

Instance variables can be associated to a composite service specification to capture data-flow aspects. These variables can be used to express branching conditions and to provide (store) input (output) parameters to (from) the service operations invoked in each task. They can also be manipulated in the actions attached to the transitions of the statechart. Note that the data-flow perspective is not relevant for the purposes of AgFlow since the methods for allocating services to tasks only need to consider the control-flow dependencies and the QoS of the component services. Data-flow is relevant for the execution of composite services by orchestration engines such as Self-Serv [4].

2.4 Execution paths and plans

In this section, we define two concepts used in the remainder of the paper: *execution path* and *execution plan*. To simplify the discussion, we initially assume that all the statecharts that we deal with are acyclic. If a statechart contains cycles, a technique for "unfolding" it into an acyclic statechart needs to be applied beforehand. Details of the unfolding process are given in Section 4.2.3.

Definition 1 (*Execution path*). An execution path of a statechart is a sequence of states [t_1, t_2, .. t_n], such that t_1 is the initial state, t_n is the final state, and for every state t_i ($1 < i < n$):

- t_i is a direct successor of one of the states in [t_1,...,t_{i-1}]
- t_i is not a direct successor of any of the states in [t_{i+1},...,t_n]
- There is no state t_j in [t_1, ..., t_{i-1}] such that t_j and t_i belong to two alternative branches of the statechart.
- If t_i is the initial state of one of the concurrent regions of an AND-state *AST*, then for every other concurrent region *C* in *AST*, one of the initial states of *C* belongs to the set $\{t_1, ..., t_{i-1}, t_{i+1},...,t_n\}$. In other words, when an AND-state is entered, all the concurrent branches of this AND-state are executed.

□

This definition relies on the concept of a *direct successor* of a state. Roughly stated, a basic state t_b in a statechart is a direct successor of another basic state t_a if there is a sequence of adjacent transitions[3] going from t_a to t_b without traversing any other basic state. In other words, the first transition in the sequence stems from t_a, the last transition leads to t_b, and all intermediate transitions stem from and lead to either compound, initial, or final states (but are not incident to a basic state).

Since it is assumed that the underlying statechart is acyclic, it is possible to represent an execution path as a Directed Acyclic Graph (DAG) as follows.

Definition 2 (*DAG representation of an execution path*). Given an execution path [t_1, t_2, .. t_n] of a statechart ST, the DAG representation of this execution path is a graph obtained as follows:

- The DAG has one node for each task $\{t_1, t_2, .. t_n\}$.
- The DAG contains an edge from task t_i to task t_j iff t_j is a direct successor of t_i in the statechart ST.

□

If a statechart contains conditional branchings, it has multiple execution paths. Each execution path represents a sequence of tasks to complete a composite service execution. Figure 4 gives an example of statechart's execution paths. In this example, since there is one conditional branching after task t_4, there are two paths, called W_{e1} and W_{e2} respectively. In the execution path W_{e1}, task t_5 is executed after task t_4, while in the execution path W_{e2}, task t_6 is executed after task t_4.

As stated earlier, each basic state of a statechart describing a composite service is labeled with an invocation to an operation provided by a given service class. Actual Web services belonging to the required service classes are selected during the execution of the composite service. Hence, it is possible to execute an execution path of a statechart in different ways by allocating different Web

[3]Two transitions are adjacent if the target state of one is the source state of the other.

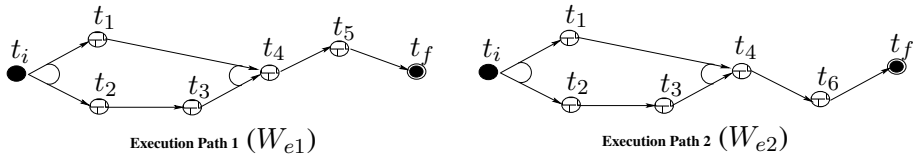


Figure 4: DAG representation of the execution paths of the statechart of Figure 3.

services to the basic states in the path. The concept of execution plan defined below captures the various ways of executing a given execution path.

Definition 3 (*Execution plan*). A set of pairs $p = \{< t_1, s_1 >, < t_2, s_2 >, \ldots, < t_n, s_n >\}$ is an execution plan of an execution path W_e iff:

- $\{t_1, t_2, \ldots t_n\}$ is the set of tasks in W_e.
- For each pair $< t_i, s_i >$ in p, service s_i belongs to the service class associated with task t_i. In other words, service s_i provides the operation required by task t_i.

□

3 Web Service Quality Model

In a Web environment, several Web services may provide similar functionality with different non-functional property values (e.g., different prices). In the composition model presented in the previous section, such Web services will typically be grouped together in a single community. To differentiate the members of a community during service selection, their non-functional properties need to be considered. For this purpose, we adopt a Web service quality model based on a set of quality criteria (i.e., non-functional properties) that are applicable to all Web services, for example, their pricing and reliability. Although the adopted quality model has a limited number of criteria (for the sake of illustration), it is extensible: new criteria can be added without fundamentally altering the service selection techniques built on top of the model. In particular, it is possible to extend the quality model to integrate non-functional service characteristics such as those proposed in [26], or to integrate service QoS metrics such as those proposed by [31].

In this section, we first present the quality criteria in the context of elementary (i.e. stand-alone) services, before turning our attention to composite services. For each criterion, we provide a definition, indicate its granularity (i.e., whether it is defined for an entire service or for individual service operations), and provide rules to compute its value for a given service.

3.1 Quality Criteria for Elementary Services

We consider five generic quality criteria for elementary services: (1) *execution price*, (2)*execution duration*, (3) *reputation*, (4) *reliability*, and (5) *availability*.

- **Execution price**. Given an operation op of a service s, the execution price $q_{pr}(s, op)$ is the fee that a service requester has to pay for invoking the operation op. Web service providers either directly advertise the execution price of their operations, or they provide means for potential requesters to inquire about it.

- **Execution duration.** Given an operation op of a service s, the execution duration $q_{du}(s, op)$ measures the expected delay in seconds between the moment when a request is sent and the moment when the results are received. The execution duration is computed using the expression $q_{du}(s, op) = T_{process}(s, op) + T_{trans}(s, op)$, meaning that the execution duration is the sum of the processing time $T_{process}(s, op)$ and the transmission time $T_{trans}(s, op)$. Services advertise their processing time or provide methods to inquire about it. The transmission time is estimated based on past executions of the service operations, i.e., $T_{trans}(s, op) = \frac{\sum_{i=1}^{n} T_i(s,op)}{n}$, where $T_i(s, op)$ is a past observation of the transmission time, and n is the number of execution times observed in the past.

- **Reliability.** The reliability $q_{rel}(s)$ of a service s is the probability that a request is correctly responded within the maximum expected time frame indicated in the Web service description. Reliability is a measure related to hardware and/or software configuration of Web services and the network connections between the service requesters and providers. The value of the reliability is computed from data of past invocations using the expression $q_{rel}(s) = N_c(s)/K$, where $N_c(s)$ is the number of times that the service s has been successfully delivered within the maximum expected time frame, and K is the total number of invocations.

- **Availability.** The availability $q_{av}(s)$ of a service s is the probability that the service is accessible. The value of the availability of a service s is computed using the following expression $q_{av}(s) = T_a(s)/\theta$, where T_a is the total amount of time (in seconds) in which service s is available during the last θ seconds (θ is a constant set by an administrator of the service community). The value of θ may vary depending on a particular application. For example, in applications where services are more frequently accessed (e.g., stock exchange), a small value of θ gives a more accurate approximation for the availability of services. If the service is less frequently accessed (e.g., online bookstore), using a larger θ value is more appropriate. Here, we assume that Web services send notifications to the system about their running states (i.e., available, unavailable).

- **Reputation.** The reputation $q_{rep}(s)$ of a service s is a measure of its trustworthiness. It mainly depends on end user's experiences of using the service s. Different end users may have different opinions on the same service. The value of the reputation is defined as the average ranking given to the service by end users, i.e., $q_{rep} = \frac{\sum_{i=1}^{n} R_i}{n}$, where R_i is the end user's ranking on a service's reputation, n is the number of times the service has been graded. Usually, end users are given a range to rank Web services. For example, in Amazon.com, the range is $[0, 5]$.

The quality vector of an operation op of a service s is defined by the following expression:

$$q(s, op) = (q_{pr}(s, op), q_{du}(s, op), q_{av}(s), q_{re}(s), q_{rep}(s)) \quad (1)$$

Note that the method for computing the value of the quality criteria is not unique. Other computation methods can be designed to fit the needs of specific applications. The service selection approaches presented in Section 4 are independent of these computation methods.

3.2 Quality Criteria for Composite Services

The quality criteria defined above in the context of elementary Web services, are also used to evaluate the QoS of composite services. Table 1 provides aggregation functions for the computation of the QoS of a composite service CS when executed using plan $p = \{< t_1, s_1 >, < t_2, s_2 >, \ldots, < t_n, s_n >\}$. A brief explanation of each criterion's aggregation function follows.

Table 1: Aggregation functions for computing the QoS of execution plans

Criteria	**Aggregation function**
Price	$q_{pr}(p) = \sum_{i=1}^{N} q_{pr}(s_i, op(t_i))$
Duration	$q_{du}(p) = CPA(p, q_{du})$
Reputation	$q_{rep}(p) = \frac{1}{N} \sum_{i=1}^{N} q_{rep}(s_i)$
Reliability	$q_{rel}(p) = \Pi_{i=1}^{N}(e^{q_{rel}(s_i)*z_i})$
Availability	$q_{av}(p) = \Pi_{i=1}^{N}(e^{q_{av}(s_i)*z_i})$

- **Execution price**: The execution price $q_{pr}(p)$ of an execution plan p is a sum of the execution prices of the operations invoked over the services that participate in p. In the equation for the execution price given in Table 1, $op(t_i)$ denotes the operation invoked by task t_i.

- **Execution duration**: The execution duration $q_{du}(p)$ of an execution plan p is computed using the Critical Path Algorithm (CPA) [28]. Specifically, the CPA is applied to the the execution path W_e of execution plan p, seen as a project digraph. The critical path of a project digraph is a path from the initial state to the final state which has the longest total sum of weights labeling its

nodes. In the case at hand, a node corresponds to a task t in W_e, and its weight is the execution duration of the service operation invoked by t, that is: $q_{du}(sv_p(t), op(t))$, where $sv_p(t)$ is the service assigned to task t in plan p, and $op(t)$ denotes the operation invoked by task t. A task that belongs to the critical path is called a *critical task*, while a service assigned to a task that belongs to the critical path is called a *critical service*.

Figure 5 provides an example of a critical path. This figure depicts an execution path as a project digraph, and an associated execution plan p, where $p = \{ < t_1, s_1 >, < t_2, s_2 >, < t_3, s_3 >, < t_4, s_4 >, < t_5, s_5 > \}$. For each service, its execution duration is shown next to it. There are two *project paths* in this project digraph, where project path 1 is $< t_1, t_4, t_5 >$ and project path 2 is $< t_2, t_3, t_4, t_5 >$. The execution time of project path 1 (project path 2) is 37 seconds (62 seconds). The critical path is therefore path 2 and the execution duration of the plan is 62 seconds. Task t_2, t_3, t_4 and t_5 are critical tasks while services s_2, s_3, s_4 and s_5 are critical services.

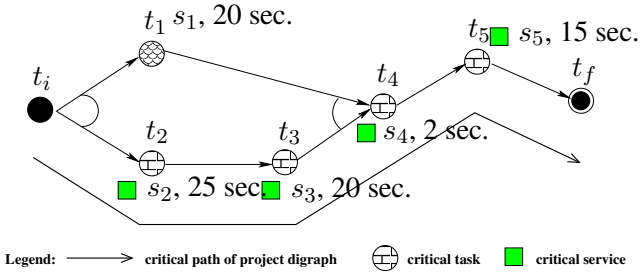


Figure 5: Example of a Critical Path

- **Reputation**: The reputation $q_{rep}(p)$ of an execution plan p is the average of the reputations of the services that participate in p.

- **Reliability**: The reliability $q_{rel}(p)$ of an execution plan p is the product of the factors $e^{q_{rel}(s_i)*z_i}$, where z_i is equal to 1 if service s_i is a critical service in the execution plan p, or 0 otherwise. If $z_i = 0$, i.e., service s_i is not a critical service, then $e^{q_{rel}(s_i)*z_i} = 1$, and hence, the reliability of service s_i will not affect the overall plan's reliability.

- **Availability**: The availability $q_{av}(p)$ of an execution plan p is given by the function $e^{q_{av}(s_i)*z_i}$, where $q_{av}(s_i)$ is the availability of service s_i.

Given these functions, the quality vector of a composite service's execution plan is defined as:

$$q(p) = (q_{pr}(p), q_{du}(p), q_{av}(p), q_{re}(p), q_{rep}(p)) \quad (2)$$

4 QoS-Driven Service Selection for Web Service Composition

In this section, we present two service selection approaches, namely *local optimization* and *global planning*.

4.1 Service Selection by Local Optimization

In this approach, the selection of the Web service that will execute a given task of a composite service specification is done at the last possible moment and without taking into account the other tasks involved in the composite service. When a task actually needs to be executed, the system collects information about the QoS of each of the Web services that can execute this task (namely the *candidate Web services* for this task). After collecting this QoS information, a *quality vector* is computed for each of the candidate Web services, and based on these quality vectors, the system selects one of the candidate Web services by applying a Multiple Criteria Decision Making (MCDM)[6] technique. This selection process is based on the *weight* assigned by the user to each criterion, and a set of user-defined constraints expressed using a simple expression language. Examples of constraints that can be expressed include duration constraints and price constraints. However, constraints can only be expressed on individual tasks, and not on combinations of tasks. In other words, it is not possible to express the fact that the sum of the durations for two or more tasks should not exceed a given threshold.

To illustrate the local optimization approach, we use the 5 quality dimensions discussed earlier, but other quality dimensions can be used instead without any fundamental changes. The dimensions are numbered from 1 to 5, with 1 = `price`, 2 = `duration`, 3 = `availability`, 4 = `reliability`, and 5 = `reputation`. Given a task t_j in a composite service, there is a set of candidate Web services $S_j = \{s_{1j}, s_{2j}, ..., s_{nj}\}$ that can be used to execute this task. By merging the quality vectors of all these candidate Web services, a matrix $\mathbf{Q} = (Q_{i,j}; 1 \leq i \leq n, 1 \leq j \leq 5)$ is built, in which each row Q_j corresponds to a Web service s_{ij} while each column corresponds to a quality dimension.

A Simple Additive Weighting (SAW) [6] technique is used to select an optimal Web service. There are two phases in applying SAW:

- **Scaling Phase**

  Some of the criteria could be negative, i.e., the higher the value, the lower the quality. This includes criteria such as execution time and execution price. Other criteria are positive, i.e., the higher the value, the higher the quality. For negative criteria, values are scaled according to equation 3. For positive criteria, values are scaled according to equation 4.

$$V_{i,j} = \begin{cases} \frac{Q_j^{max} - Q_{i,j}}{Q_j^{max} - Q_j^{min}} & \text{if } Q_j^{max} - Q_j^{min} \neq 0 \\ 1 & \text{if } Q_j^{max} - Q_j^{min} = 0 \end{cases} \quad (3)$$

$$V_{i,j} = \begin{cases} \frac{Q_{i,j} - Q_j^{min}}{Q_j^{max} - Q_j^{min}} & \text{if } Q_j^{max} - Q_j^{min} \neq 0 \\ 1 & \text{if } Q_j^{max} - Q_j^{min} = 0 \end{cases} \quad (4)$$

In the above equations, Q_j^{max} is the maximal value of a quality criteria in matrix $\mathbf{Q}$, i.e., $Q_j^{max} = Max(Q_{i,j}), 1 \leq i \leq n$. While Q_j^{min} is the minimal value of a quality criteria in matrix $\mathbf{Q}$, i.e., $Q_j^{min} = Min(Q_{i,j}), 1 \leq i \leq n$. By applying these two equations on $\mathbf{Q}$, we obtain a matrix $V = (V_{i,j}; 1 \leq i \leq n, 1 \leq j \leq 5)$, in which each row V_j corresponds to a Web service s_{ij} while each column corresponds to a quality dimension.

As an example, assume that there are eight Web services in S_5 for task t_5 and that their values for service reputation are given by the vector Q_5 = (7.5,8.4,9,8.3,8.7,9.1,9.4,9.2,9.5). Since reputation is a positive criteria, equation 4 is used for scaling and thus Q_5^{max} = 9.5, Q_5^{min} = 7.5, and V_5 = (0, 0.55, 0.45, 0.75, 0.4,0.6, 0.8, 0.95,1).

- **Weighting Phase**

  The following formula is used to compute the overall quality score for each Web service:

$$Score(s_i) = \sum_{j=1}^{5} (V_{i,j} * W_j) \quad (5)$$

  where $W_j \in [0, 1]$ and $\sum_{j=1}^{5} W_j = 1$. W_j represents the weight of criterion j. As stated before, end users express their preferences regarding QoS by providing values for the weights W_j.

For a given task, the system will choose the Web service which satisfies all the user constraints for that task, and which has the maximal score. If there are several services with maximal score, one of them is selected randomly. If no service satisfies the user constraints for a given task, an execution exception will be raised and the system will propose the user to relax these constraints.

4.2 Service Selection by Global Planning

In the local optimization approach, service selection is done for each task individually. Although service selection is locally optimized, the global quality of the execution may be sub-optimal. For example, if two tasks A and B are executed in parallel and need to synchronize upon completion, then it is not worth optimizing the duration of A, if it is known that B takes considerably more time to execute. Instead, it is preferable to optimize (for example) the price of A, while optimizing the duration of B. Furthermore, as explained above, when applying local optimization it is not possible to enforce inter-task constraints over the composite service execution such as: "the total price of the composite service execution should be at most $500".

In this section, we present a global planning approach for Web services selection which overcomes these limitations. We first present a *naive approach* for global planning, and then present a novel

integer programming approach that avoids some obvious computational problems associated with the naive approach.

4.2.1 Optimal Execution Plan of an Execution Path

For each task t_j in an execution path, there is a set of candidate Web services $S_j = \{s_{1j}, s_{2j}, ..., s_{nj}\}$ that can execute task t_j. Assigning a candidate Web service s_{ij} to each task t_j in an execution path leads to a possible execution plan. In the global planning approach, all possible plans associated to a given execution path are generated (at least conceptually speaking) and the one which maximizes the user's preferences while satisfying the imposed constraints is then selected. The selection of an execution plan relies on the application of a MCDM approach to the quality matrix $\mathbf{Q} = (Q_{i,j}; 1 \leq i \leq n, 1 \leq j \leq 5)$ of the execution path. In this matrix, each row corresponds to the quality vector of one possible execution plan for the execution path.

As in the local selection approach, a SAW technique is used to select an optimal execution plan. The two phases of applying SAW are:

- **Scaling Phase**

  As in the previous section, we first scale the values of each quality criterion. For negative criteria, values are scaled according to equation 3. For positive criteria, values are scaled according to equation 4. Note that we can compute the value of Q_j^{max} and Q_j^{min} in these equations without generating all possible execution plans. For example, in order to compute the maximum execution price (i.e., Q_{pr}^{max}) of all the execution plans, we select the most expensive Web service for each task and sum up all these execution prices to compute Q_{pr}^{max}. In order to compute the minimum execution duration (i.e., Q_{du}^{min}) of all the execution plans, we select the service with the shortest execution duration for each task and use CPA to compute Q_{du}^{min}. The computation cost of Q_j^{max} and Q_j^{min} is thus polynomial. After the scaling phase, we obtain the matrix $V = (V_{i,j}; 1 \leq i \leq n, 1 \leq j \leq 5)$.

- **Weighting Phase**

  The following formula is used to compute the overall quality score for each execution plan:

$$Score(p_i) = \sum_{j=1}^{5} (V_{i,j} * W_j) \tag{6}$$

  where $W_j \in [0, 1]$ and $\sum_{j=1}^{5} W_j = 1$. W_j represents the weight of each criterion. End users can give their preferences on QoS (i.e., balance the impact of the different criteria) to select a desired execution plan by adjusting the value of W_j. The global planner will choose the execution path which has the maximal value of $Score(p_i)$ (i.e., $max(Score(p_i))$). If there is more than one

execution plan which has the same maximal value of $Score(p_i)$, then an execution plan will be selected from them randomly.

4.2.2 Handling Multiple Execution Paths

Assume that a statechart has multiple execution paths. For each of these paths, an optimal execution plan can be selected using the method described above. Since each of the selected plans only covers a subset of the statechart, the global planner needs to aggregate these "partial" execution plans into an overall execution plan. For example, for the `Travel Planner` statechart W (see Figure 3), there are two execution paths W_{e1} and W_{e2}, each of which has its own optimal execution plan, say p_1 and p_2. Neither p_1 nor p_2 covers all tasks in W, so they need to be merged somehow.

Assume that statechart W has n tasks (i.e., t_1, t_2, ..., t_k) and m execution paths (i.e., W_{e1}, W_{e2}, ..., W_{em}). For each execution path, the global planner selects an optimal execution plan. Consequently, we obtain m optimal execution plans (i.e., $p_1, p_2, ..., p_m$) for these execution paths. The global planner adopts the following approach to aggregate multiple execution plans into an overall execution plan.

1. Given a task t_i, if t_i only belongs to one execution path (e.g., W_{ej}), then the global planner selects W_{ej}'s execution plan p_j to execute the task t_i. We denote this as $<t_i, p_j>$. For example, in the `Travel Planner` example, task t_5 (i.e., `BikeRental`) only belongs to execution path W_{e2}. In this case, W_{e2}'s execution plan p_2 is used to determine the service that will execute t_5. This fact is denoted by $<t_5, p_2>$.

2. Given a task t_i, if t_i belongs to more than one execution paths (e.g., W_{ej}, W_{ej+1}, ..., W_{em}), then there is a set of execution plans (i.e., p_j, p_{j+1}, ..., p_m) that can be used to execute W_{si}. Hence, the global planner needs to select one of these execution plans. The selection can be done by identifying the *hot path* for task t_i. Here, the hot path of a task t_i is defined as the execution path that has been most frequently used to execute the task t_i in past instances of the composite service. For example, in the `Travel Planner` statechart, task t_4 (`DrivingTimeCalcultation`) belongs to both execution paths W_{e1} and W_{e2}. Assume that the composite service has been executed 25 times, 20 of which have followed execution path W_{e1}, while the other 5 have followed W_{e2}. Since the execution path W_{e1} is used more frequently to execute task t_4 (i.e., W_{e1} is the hot path for t_2), W_{e1}'s optimal execution plan p_1 is used to determine the service that will execute t_4. This is denoted by $<t_4, p_1>$.

The system keeps the execution traces of the composite service using the methods described in [13]. These traces allow the global planner to identify the hot path for each task. In the absence of (enough) traces, a human expert must indicate the hot path

4.2.3 Unfolding Cyclic Statecharts

Hitherto, we have assumed that the statecharts are acyclic. If a statechart contains cycles, these need to be "unfolded" so that the resulting statechart has a finite number of execution paths. The method used to unfold the cycles of a statechart is to examine the logs of past executions in order to determine the maximum number of times that each cycle is taken. The states appearing between the beginning and the end of the cycle are then cloned as many times as the transition causing the cycle is taken.[4]

This unfolding method works only if the "beginning" and the "end" of each cycle in the statechart can be clearly identified. It does not work, for example, if the transitions causing the cycle are located in two different conditional branches as in Figure 6. In this case, it is not possible to determine which is the first state and which is the last state in the cycle (is it t_2 or t_3?). An equivalent statechart which can be unfolded using the above method is shown in Figure 7. In this equivalent statechart, the transition causing the cycle does not crosses the boundaries of any conditional branch. It can

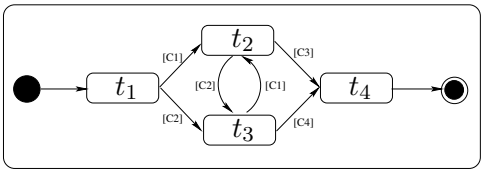


Figure 6: "Unfoldable" statechart

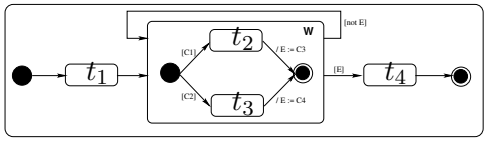


Figure 7: Foldable statechart equivalent to that in Figure 6

be proved that any arbitrary statechart can be transformed into an equivalent statechart in which the cycles do not cross the boundaries of conditional branches (as illustrated above). This proof is similar to that of the theorem stating that any program written using "goto" statements, can be transformed into an equivalent program which only uses structured loops (i.e., "while" statements). [27], for example, describes an algorithm for performing this kind of transformation with minimal number of auxiliary boolean variables. [19] points out to the fact that the algorithms for transforming "arbitrary cycles" into "structured" ones do not always apply in the presence of parallel branches. However,

[4]The idea of cloning the states in a loop for the purpose of transforming a cyclic statechart into an acyclic one has been proposed in [15], where probabilistic models are used to determine how many times the states should be duplicated.

they do apply for statecharts since in statecharts it is not possible to have arbitrary transitions going from a parallel branch to another such as those put forward by [19].

Given the statechart with structured cycles of Figure 7, the unfolding process proceeds by determining the maximum number of times that the transition causing the cycle is taken (i.e. the transition labeled with condition "not E"). If for example this transition has been taken a maximum of two times, then three copies of the compound state W will appear in the resulting acyclic statechart. From this acyclic statechart, we can then generate all the possible execution paths.

4.2.4 Integer Programming Solution

The global planning approach by exhaustive seaching for selecting an optimal execution plan described above, requires the generation of all possible execution plans. Assuming that there are n tasks in a statechart and m candidate Web services for each task, the total number of execution plans is m^n, making this exhaustive searching approach impractical. Accordingly, we propose a method based on Integer Programming (IP) [17] for selecting an optimal execution plan without generating all the possible execution plans.

There are three inputs in an IP problem: a set of *variables*, an *objective function* and a set of *constraints*, where both the objective function and the constraints must be linear. IP attempts to maximize or minimize the value of the objective function by adjusting the values of the variables while enforcing the constraints. The output of an IP problem is the maximum (or minimum) value of the objective function and the values of variables at this maximum (minimum).

The problem of selecting an optimal execution plan is mapped into an IP problem as follows. First, for every Web service s_{ij} that can be used to execute a task t_j, we include in the IP problem an integer variable y_{ij}, such that by convention y_{ij} is 1 if service s_{ij} is selected for executing task t_j, 0 otherwise. We also introduce a set of integer variables x_j, such that x_j denotes the expected start time of task t_j (if t_j is executed at all). This set of variables are used to express the constraints on the execution duration.

Next, since we rely on MCDM and SAW to determine the desirability of an execution plan, we use the following objective function, which is based on equations 3,4, and 6:

$$Max\left(\sum_{l=1}^{2}\left(\frac{Q_l^{max}-Q_{i,l}}{Q_l^{max}-Q_l^{min}}*W_l\right)+\sum_{l=3}^{5}\left(\frac{Q_{i,l}-Q_l^{min}}{Q_l^{max}-Q_l^{min}}*W_l\right)\right) \tag{7}$$

where $W_l \in [0,1]$ and $\sum_{j=1}^{5} W_j = 1$. W_l is the weight assigned to the quality criteria. The remainder of this subsection describes the constraints of the IP problem.

Allocation constraint For each task t_j, there is a set of Web services S_j that can be assigned (allocated) to it. However, for each task t_j, we should only select one Web service to execute this task.

Given that y_{ij} denotes the selection of Web service s_{ij} to execute task t_j, the following constraint must be satisfied:

$$\sum_{i \in S_j} y_{ij} = 1, \ \forall j \in A \tag{8}$$

where A is the set of tasks in the statechart. For example, assume that there are 100 potential Web services that can execute task j. Since only one of them will be selected to execute task j, we have that $\sum_{i=1}^{100} y_{ij} = 1$.

Constraints on Execution Duration, Price, and Reputation Let x_j denote the expected start time of task t_j, p_{ij} denote the execution duration of task t_j when assigned to service s_{ij}, and p_j denote the expected duration of task t_j knowing which service has been assigned to it. Also, let $t_j \rightarrow t_k$ denote the fact that task t_k is a direct successor of task t_j. We have the following constraints:

$$\sum_{i \in S_j} p_{ij} \ y_{ij} = p_j, \forall j \in A \tag{9}$$

$$x_k - (p_j + x_j) \geq 0, \forall t_j \rightarrow t_k, j, k \in A \tag{10}$$

$$q_{du} - (x_j + p_j) \geq 0, \forall j \in A \tag{11}$$

Constraint 9 indicates that the execution duration of a given task t_j must be the execution duration of one of the Web services in A, since one and only one of these services will be selected to execute task t_j. Constraint 10 indicates that if task t_k is a direct successor of task t_j, then the execution of t_k must start after task t_j has been completed. Constraint 11 indicates that the execution of a composite service plan is completed only when all the tasks in the plan are completed.

Now, let z_{ij} be an integer variable that has value 1 or 0: 1 indicates that Web service s_{ij} is a critical service and 0 indicates otherwise. The relationship between the duration of an execution plan and the duration of the critical services of the plan is captured by the following equation.

$$q_{du} = \sum_{j \in A} \sum_{i \in S_j} p_{ij} z_{ij} \tag{12}$$

Similarly, assuming that variable c_{ij} represents the execution price of Web service s_{ij}, we impose the following constraint to capture the total execution price of a composite service:

$$q_{pr} = \sum_{j \in A} \sum_{j \in S_j} c_{ij} \ y_{ij} \tag{13}$$

An alternative constraint 13 is the following:

$$\sum_{j \in A} \sum_{j \in S_j} c_{ij} y_{ij} \leq B, B > 0 \tag{14}$$

where B is the budget set by the user. This constraint indicates that the execution price of the composite service should not be greater than B. By introducing a budget constraint the problem needs to be explicitly solved as an integer programming problem, as opposed to linear programming. This problem is a special case of the knapsack problem and hence NP-hard [22].

Finally, assuming that variable r_{ij} represents the reputation of Web service s_{ij}, we impose the following constraint to capture the overall reputation of an execution plan:

$$q_{rep} = \sum_{j \in A} \sum_{j \in S_j} r_{ij} y_{ij} \tag{15}$$

Note that other criteria with a simple linear aggregation function could be integrated in the same way as the reputation.

Constraints on Reliability and Availability Among the criteria used to select Web services, the availability and the reliability are associated with nonlinear aggregation functions (see Table 1). In order to capture them in the IP problem, we need to linearize them using a logarithmic function. Assume that variable a_{ij} represents the reliability of Web service s_{ij}. Since z_{ij} indicates whether Web service s_{ij} is a critical service or not, the reliability of the execution plan is:

$$q_{rel} = \Pi_{j \in A} \left(\sum_{j \in S_j} e^{a_{ij} z_{ij}} \right)$$

By applying the logarithm function ln, we obtain:

$$ln(q_{rel}) = \sum_{j \in A} ln \left(\sum_{j \in S_j} e^{a_{ij} z_{ij}} \right)$$

Since $\sum_{j \in A} z_{ij} = 1$ and $z_{ij} = 0$ or 1, we have that:

$$ln(q_{rel}) = \sum_{j \in A} \left(\sum_{j \in S_j} a_{ij} z_{ij} \right)$$

Let $q'_{rel} = ln(q_{rel})$, we introduce the following constraint into the IP problem in order to capture the reliability criterion:

$$q'_{rel} = \sum_{j \in A} \sum_{j \in S_j} a_{ij} z_{ij} \tag{16}$$

Similarly, assume that b_{ij} represents the availability of the Web service s_{ij}. We introduce the following constraint:

$$q'_{av} = \sum_{j \in A} \sum_{i \in S_j} b_{ij} z_{ij} \tag{17}$$

where $q'_{av} = ln(q_{av})$.

Constraints on the Uncertainty of Execution Duration Hitherto, we have assumed that the execution duration p_{ij} of a Web service is deterministic. In reality, the execution duration p_{ij} of a Web service s_{ij} is uncertain, in the sense that some deviations between the actual duration and p_{ij} are likely to occur. In order to capture this uncertainty, we assume that p_{ij} is a random variable that follows a normal distribution with mean μ_{ij} and standard deviation σ_{ij}, which can be computed from the history of past executions.

Since $q_{du} = \sum_{i \in A} \sum_{i \in S_j} p_{ij} z_{ij}$ is a linear combination of random variables with normal distribution, q_{du} itself is a random variable with normal distribution [33] and its deviation σ_{du} is:

$$\sigma_{du}^2 = \sum_{j \in A} \sum_{i \in S_j} \sigma_{ij}^2 z_{ij} \tag{18}$$

With this equation at hand, it is possible to extend the objective function in order to incorporate the deviation from the expected execution duration as one of the optimization criteria. The extended objective function is:

$$Max \left(\sum_{l=0}^{2} \left(\frac{Q_l^{max} - Q_{i,l}}{Q_l^{max} - Q_l^{min}} * W_l \right) + \sum_{l=3}^{5} \left(\frac{Q_{i,l} - Q_l^{min}}{Q_l^{max} - Q_l^{min}} * W_l \right) \right) \tag{19}$$

where $Q_0 = \sigma_{du}^2$ and $W_0 \in [0, 1]$ are the values assigned by the user to the criterion "deviation from expected execution duration".

Given the above variables, objective function, and constraints, an IP solver is able to compute the values of y_{ij} corresponding to an optimal execution plan. As a side effect, the IP solver will also provide a tentative schedule for the tasks in the statechart by assigning values to the variables x_j. This assignment of variables can be used by the execution engine, but they are not strictly necessary, since the execution engine can schedule the tasks using its own scheduling mechanisms.

Note that the proposed method for translating the problem of selecting an optimal execution plan into an IP problem is generic, and although it has been illustrated with the 5 criteria introduced in Section 3 (plus the "duration deviation" criterion), other criteria can be accommodated.

4.2.5 Replanning the Execution of Composite Services

When using the global planning approach, an execution plan is built at the beginning of the execution of the composite service. Once the execution has started, several contingencies may occur, e.g., a component service becomes unavailable or the QoS of one of the component services changes significantly. In these situations, a replanning procedure may be triggered in order to ensure that the QoS of the composite service execution remains optimal. In this section, we discuss how this replanning procedure is conducted.

Consider a composite service consists of a set of task $T = \{t_1, t_2, ..., t_n\}$. Based on the execution status, T can be partitioned into four regions: (i) region R_α containing tasks that have been completed;

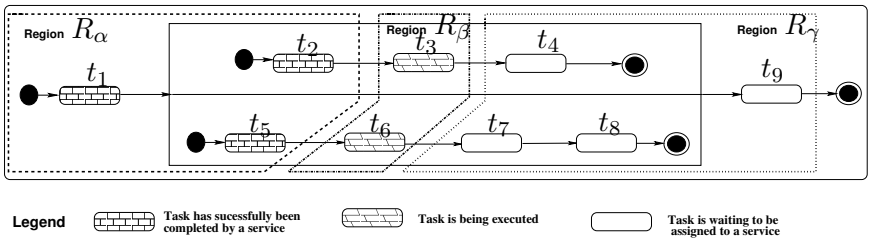


Figure 8: Partition of a composite service into regions for replanning

region R_β containing tasks that are currently being executed; (iii) region R_γ containing tasks that have not yet started; and (iv) region R_δ containing tasks which belong to a conditional branch that is no longer accessible from any of the tasks in R_β (i.e. tasks that have not been executed and will not be executed). An example of a partitioning of a composite service is given in Figure 8. Note that this composite service has no conditional branches, so $R_\delta = \emptyset$.

When using the global planning approach, the composite service execution needs to be replanned in the following cases:

1. One or several exceptions have occurred during the execution/invocation of component services in region R_β. Execution exceptions include (i) component services failing to execute tasks; and (ii) a service not being able to attain its expected QoS.

2. Changes have been reported in the expected QoS of candidate services for tasks in region R_γ. Relevant changes include: (i) services selected during the global planning becoming unavailable or changing their QoS properties; (ii) new candidate services offering better QoS than existing services appearing; and (iii) existing services raising their advertised QoS.

As discussed in Section 4.2.4, there are three inputs in an IP problem: a set of *variables*, an *objective function* and a set of *constraints*. When an execution plan is revised at runtime, the *variables* and the *objective function* remain the same as at beginning of the execution. However, additional constraints need to be introduced in order to capture the current execution status. Specifically, the actual QoS delivered by the tasks in region R_α, and the fact that the tasks in region R_α and R_β cannot be reassigned to new candidate services, need to be encoded in the constraints of the IP problem. For example, assuming that task t_1 has been successfully completed by service s_{41}, that the actual execution duration was 20 seconds, and that the actual execution cost was \$10, the following constraints will be added to the IP problem by the global planner:

$$y_{41} = 1, \tag{20}$$

$$a_{41} = 1, b_{41} = 1, \tag{21}$$

$$p_{41} = 20, c_{41} = 10 \tag{22}$$

Constraint 20 indicates that service s_{41} executed task t_1. Constraint 21 indicates that service s_{41} was available and reliable when invoked. Finally, constraint 22 encodes the actual execution duration and cost of t_1. The introduction of these constraints force the IP solver to select an execution plan which takes into account what has already been accomplished during the execution of the composite service. With these constraints, the IP solver can compute an optimal plan for region R_γ.

4.3 Comparison of Service Selection Approaches

In the following, we compare the local and the global service selection approaches based on the following metrics:

- **QoS of Composite Service.** This is measured along two perspectives:
  - **Quality Criteria.** As we discussed in earlier section, QoS of composite service is measured by a set of quality criteria. And since there are often tradeoffs among different quality criterion (e.g., execution time and cost), it is important that the system is able to find a combination of these dimensions that fits the user's preferences.
  - **Ability to satisfy user's requirements**. Although good quality composite service execution requires achieving optimal QoS of services, the satisfaction of end users' constraints is an equally important aspect. There are two kinds of constraints: constraints on a single task and constraints on multiple tasks. The system's ability to accept both kinds of end users' constraints is key to satisfying end user's requirements.
- **System Cost.** When the system receives a request to execute a composite service from an end user, the system utilizes resources to locate candidate Web services for the required tasks, and to select among them. More specifically, the system consumes network resources (*bandwidth cost*) to contact the service broker(s) in order to identify candidate Web services. It then consumes computational resources (*computational cost*) in order to process the results of this search and select one among the candidate Web services. Note that, when the composite service is being executed, the system also interacts with the Web services in order to orchestrate them, which also consumes bandwidth and some computational resources.

The computational cost of the local optimization approach is polynomial. The bandwidth cost is very limited: for each task, there are two messages that flow between the composite service execution engine and the service broker (`query` and `result`) and three that flow between the execution engine and the selected Web services (`enable`, `start`, and `completed`).

On the negative side, the local optimization approach has two shortcomings:

- It cannot consider global tradeoffs between quality dimensions, especially in the case of composite services involving concurrent threads. For example, in the `Travel Planner` statechart, task t_2 (`AttractionSearching`) and task t_3 (`FlightTicketBooking`) are executed concurrently. If the execution duration of task t_3 is always longer than that of task t_2, the system should select for task t_2 the candidate service that offers the lowest price, regardless of the duration. In the local selection approach however, the system does not take advantage of this fact.

- When selecting Web services, the local optimization approach can consider constraints on individual tasks, but it cannot consider global constraints, i.e., constraints that cover multiple (or all) tasks in the composite service. Also, although it is always able to select a Web service with minimal execution price or minimal execution duration for each task, it fails when both the execution price and execution duration need to be considered at a global level. For example, it cannot enforce a constraint stating that the composite service's execution price cannot exceed $500 and the execution duration cannot exceed 3 days.

The global planning approach overcomes these shortcomings, but at the price of higher computational and bandwidth cost. Indeed, the global planner first needs to select an optimal execution plan using an expensive algorithm (exponential in some cases). It then needs to monitor all the candidate Web services (whether they are included in the plan or not) thereby consuming considerable bandwidth resources. Finally, when it detects exceptions or changes, it may need to revise the execution plan at runtime, again using an expensive algorithm. Another issue with the global planning approach is that users are required to provide relatively complex input (i.e. global constraints and tradeoffs), which some users might have problems formulating.

A quantitative comparison between local optimization and global planning based on experimental results is given in Section 6.

5 Implementation

In this section, we describe the implementation of two major components of the AgFlow system architecture (shown in Figure 1): the service broker and the service composition manager.

5.1 Implementing the Service Broker

There are two meta-data repositories in the AgFlow system, namely service ontology repository and Web service repository. We adopt the UDDI registry to implement both meta-data repositories (see Figure 9). The UDDI specification provides a platform independent way of describing services and discovering businesses. The UDDI data structures provide a framework for the description of basic business and service information, and provide an architecture for an extensible mechanism (i.e.,

tModel) to provide detailed service information using any description language. We define an XML

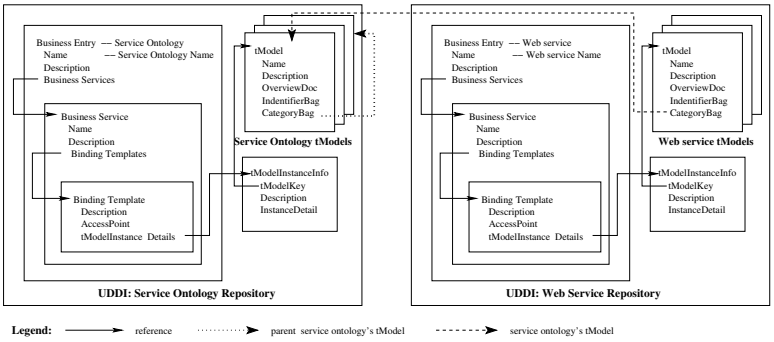


Figure 9: Service ontology repository and Web service repository

schema for service ontologies. Each service ontology is represented as an XML document. Table 4 in Appendix shows an example of a `Trip-planning` service ontology for the domain `tourism`. A separate tModel of type `serviceOntologySpec` is created for each service ontology. The information that makes up a `serviceOntologySpec` tModel is quite simple. There is a tModel key, a name (i.e., service ontology's name), an optional description, and a URL that points to the location of the service ontology description document. Table 5 in Appendix give an example of service ontology's tModel in UDDI registry.

In the Web service repository, we adopt WSDL (Web service description language) [34] to specify services. An example of a WSDL document can be found in Appendix, Table 6. It should be noted that the Web service's tModel contains the key of a service ontology's tModel in `categoryBag`, the `keyValue` is `serviceOntologySpec`. In Appendix, Table 7, an example of Web service tModel for UDDI registry is given.

Using the UDDI API, the service broker provides two kinds of interfaces for both repositories: the publish interface and the search interface. For the service ontology repository, the publish interface allows an ontology engineer to create a new service ontology. It also provides methods to modify the service ontology such as add a new service class, delete an existing service class, etc. The search interface allows service providers and end users to search and browse the existing service ontologies. The search can be based on a service ontology's domain name, synonyms, service class, etc. For the Web service repository, the publish interface allows service providers to publish or advertise their service descriptions. While the search interface allows the user to discover services by service class name, operation name, input and output data.

5.2 Implementation of the Service Composition Manager

The service composition manager consists of two modules, namely the *Execution Planner*, and *Execution Engine*.

- **Execution Planner** is the module that selects a Web service for each task in a composite service using either local optimization or global planning. It provides a specific method called `select()` that can be used by composite services to perform local optimization or IP based global planning. The IP based global planning approach is implemented as an integer programming solver based on IBM's Optimization Solutions and Library (OSL).[5] The advantage of using integer programming is that it can select an optimal execution planning without enumerating all possible execution plans.

- **Execution Engine** is the heart of the service composition manager. It manages composite service executions from beginning to the end. It determines which tasks need to be executed based on the control and data dependencies. It also maintains the state of the composite service, including the execution state and the events/messages triggered by Web services. The prototype uses the execution engine of the Self-Serv system [4].

6 Experimentation

In order to evaluate the proposed service selection approaches, we developed a travel planning application based on the one presented in [38], and conducted experiments using the implemented prototype system. Services were created using IBM's Web Services Toolkit (WSTK).[6] and deployed on a cluster of PCs. All PCs had the same configuration: Pentium III 933MHz with 512M RAM, Windows 2000, Java 2 Enterprise Edition V1.3.0, and Oracle XML Developer Kit. The PCs are connected to a LAN through 100Mbits/sec Ethernet cards.

QoS data is retrieved by the service execution engine in different ways depending on the QoS dimension. The execution duration and the execution cost are retrieved via two operations: getExecutionDuration() and getExecutionDuration() respectively. These operations are defined in the underlying service ontology. The reliability and reputation on the other hand are calculated by the service composition manager using the formulas presented in Section 3.1. For this purpose, the service composition manager logs appropriate QoS information during task executions. Finally, the availability is calculated by the service broker based on the information that it records about the up and down time of each service.

Experiments were conducted in two types of environment: *static* and *dynamic*. In a static environment, there is no change in the QoS of any component service during a given composite service execution. In addition, all component services are able to execute the tasks successfully and in conformance with their expected QoS. In a dynamic environment on the other hand, the QoS of component

[5] `http://www-3.ibm.com/software/data/bi/osl/index.html`
[6] `http://alphaworks.ibm.com/tech/webservicestoolkit`

services may undergo changes during the execution of a composite service. Specifically, existing component services may become unavailable, new component services with better QoS properties may become available, component services may not be able to complete the execution of tasks, or they may complete them but without meeting their expected QoS.

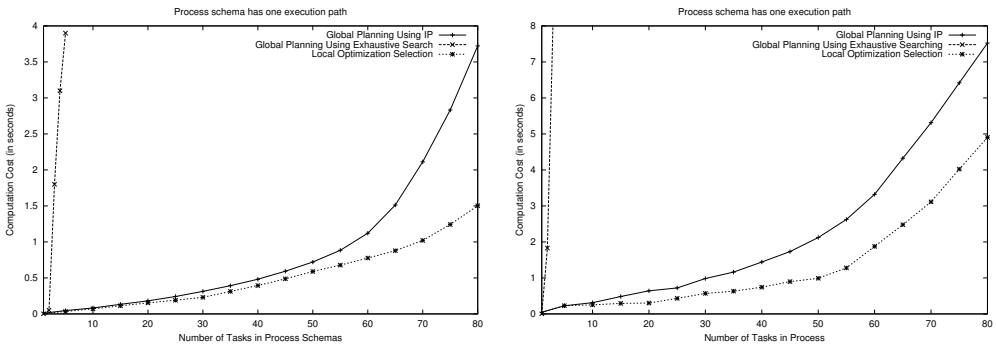
The experiments involved composite services with varying numbers of basic states. The composite services were created by randomly adding states to the composite service shown in Figure 3. The number of states varied from 10 to 80 with steps of 10 (e.g. 10, 20, ... 80). Also, we varied the number of candidate component services per task from 10 to 40 with steps of 10.

6.1 Measuring Computation Cost

The first series of experiments aimed at comparing the three selection approaches previously described (local optimization, global planning by exhaustive searching, and global planning by linear programming) with respect to the computational overhead involved by their planning phase. In other words, we measured the computation cost (in seconds) of selecting component services and computing the expected QoS of composite service executions. The experiments focused specifically on the execution price and duration. For each test case, we executed the composite service 10 times and computed the average computation cost.

6.1.1 Static Environments

In a static environment and when using a global planning approach, once a process execution plan is created the execution planner does not need to replan the process execution. So the global planner is invoked only once during a composite service execution. In the case of local optimization on the other hand, if we assume that the number of tasks that are executed is N, then the service selector in the execution planner is invoked N times to select a component service for each task.



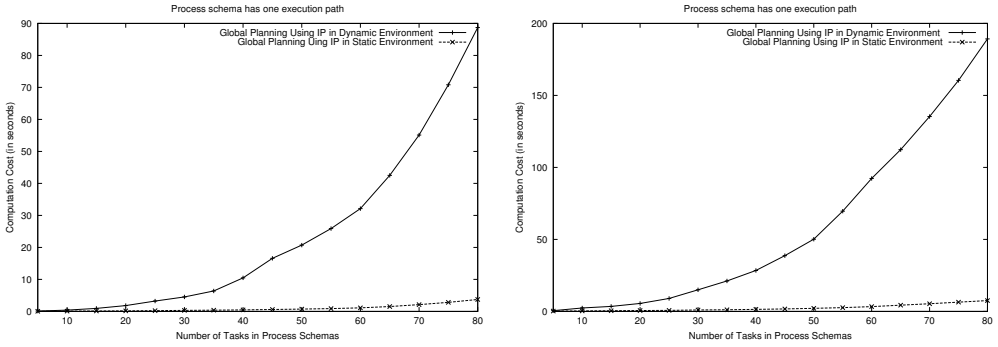
(a) Each Task has 10 candidate services (b) Each Task has 40 candidate services

Figure 10: Experimental results (computation cost) in a static environment, varying the number of tasks in the statechart and the number of candidate services per task.

Figure 10 plots computation cost (in seconds) of selecting services for composite services with only one execution path. In this experiment, we varied the number of tasks and the number of candidate services per task. In all the approaches, the computation cost increases when the number of tasks increases and the number of candidate services increases. The computation cost of global planning by exhaustive searching is very high. The computation cost of global planning by IP is higher than that of local optimization selection. When there are 80 tasks and 40 candidate Web services for each task, the computation cost of the global planning by IP (1.6 seconds) is almost 1.5 times higher than the local optimization approach (0.7 seconds).

6.1.2 Dynamic Environments

Dynamic environments were simulated by randomly changing the QoS of the component services during a composite service execution according to three QoS properties: execution price, duration, and availability. Changes are done so that, if the global planning approach is used, the execution of a composite service needs to be replanned whenever a task is completed. Hence, the global planner needs to be invoked as many times as there are tasks in the composite service.



(a) Each Task has 10 candidate services (b) Each Task has 40 candidate services

Figure 11: Experimental results (computation cost) in a dynamic environment and static environment, varying the number of tasks and the number of candidate services per task.

Figure 11 presents computation cost (in seconds) of service selection for composite services with only one execution path, where we vary the number of tasks and the number of candidate services per task. In both selection approaches, the computation cost increases with the number of tasks and the number of candidate services. The computation cost of global planning using IP is much higher than in a static one. For 80 tasks and 40 candidate services per task, the computation cost of the global planning using IP in a dynamic environment (190 seconds) is almost 25 times higher than in a static one (7.8 seconds). However, this may be considered to be an extreme case, and for reasonably sized cases, global planning delivers acceptable computation cost. Furthermore, the computation cost in a dynamic environment is spread across the lifespan of the composite service execution.

6.2 Measuring QoS of Composite Services

The second series of experiments aimed at evaluating the QoS of composite service executions in both static and dynamic environments. Table 2 presents experiment results on composites' execution duration (i.e., $Q_{du}(CS)$, in seconds) and execution price (i.e., $Q_{price}(CS)$, in dollars) in static environment, where we vary the number of tasks. The experiment results show that the global planning approach yields significantly better QoS than the local optimization approach. For example, the execution duration is consistently shorter when using global planning approach than that of local optimization approach.

Table 3 shows that global planning approach gives better QoS of composite services than that of local optimization approach in most of the cases in dynamic environments. At the same time, the average QoS of composite services in the global planning approach is better than that of the local optimization approach. However, in some cases (see table 3(1), composite service 4), the global planning approach may create worse execution result compared to the local optimization approach. The reason is that the global planning approach makes the decision based on the set of currently available Web services. Since the availability of Web services is dynamic, some services that are selected by the optimal execution plan may become unavailable when the task needs to be executed. Although the system can replan the unexecuted part of composite services, the executed part may become sub-optimal, making the entire composite service execution sub-optimal. Note also that even in dynamic environments, the global planner is still able to handle constraints spanning multiple tasks when performing execution replanning (see Section 4.2.5). For example, in table 3(1), the composite service executions are performed under a execution duration constraint ($Q_{du} < 7500$).

Table 2: Experimental Results on QoS of Composite Services in Static Environments

Composite Service	$Q_{du}(CS)$		$Q_{price}(CS)$	
	GP	LO	GP	LO
1	6523.2	8322.4	1023	1642
2	6634.4	9123.9	1117	1728
3	6843.2	9234.5	1123	1825
4	6432.5	9292.2	1132	1824
5	6347.3	8943.3	1121	1723
6	6512.3	9902.8	1185	1888
7	6451.2	9480.4	1231	1789
8	6440.5	9470.5	1275	1787
9	6970.4	9920.4	1324	1625
10	6890.3	9628.3	1235	1759
11	6590.3	9520.3	1267	1852
12	6890.3	8920.5	1250	1599
Average:	6627.16	9305.9	1191.08	1753.42

1. Composite service has 20 tasks and one execution path.

Composite Service	$Q_{du}(CS)$		$Q_{price}(CS)$	
	GP	LO	GP	LO
1	10212.4	14110.2	2216	3626
2	11221.7	15330.5	2327	3502
3	10945.2	14280.4	2424	3445
4	12535.5	15292.2	2221	3524
5	12340.5	14302.5	2123	3324
6	11350.5	14470.5	2285	3528
7	12445.2	13980.4	2184	3589
8	12440.5	14470.5	2485	3487
9	11970.4	15920.4	2160	3616
10	11250.3	15628.3	2352	3459
11	11490.3	15520.3	2373	3412
12	12890.3	14920.5	2163	3519
Average:	11757.73	14852.22	2276.08	3502.58

2. Composite service has 80 tasks and one execution path.

6.3 Discussion

From the experimental results, we conclude that the IP-based global planning approach leads to significantly better QoS of composite service executions with little extra system cost in static environments.

Table 3: Experimental Results on QoS of Composite Services in Dynamic Environments

Composite Service	$Q_{du}(CS)$		$Q_{price}(CS)$	
	GP	LO	GP	LO
1	6433.2	9322.4	1341	1834
2	6434.9	8123.9	1123	1873
3	6343.6	8654.5	1512	1836
4	7443.8	7302.2	1123	1873
5	6334.3	8463.3	1134	1838
6	7112.7	9474.8	1324	1834
7	6235.4	8674.4	1241	1835
8	6542.7	8354.5	1132	1873
9	6234.7	8363.4	1312	1835
10	6435.2	8733.3	1153	1736
11	5823.6	8340.3	1212	1873
12	7245.1	9473.5	1142	1546
Average:	6609.93	8673.37	1229.08	1815.5

1. Composite service has 20 tasks and one execution path.

Composite Service	$Q_{du}(CS)$		$Q_{price}(CS)$	
	GP	LO	GP	LO
1	11225.5	18346.3	2341	3834
2	11342.5	18324.6	2651	3834
3	12341.4	18345.4	2234	3345
4	13221.4	17344.3	2612	3873
5	16512.6	18734.7	2123	3345
6	12315.2	17232.3	2234	3873
7	13123.6	17234.3	2612	3873
8	13123.7	14235.7	2512	3458
9	12341.9	17623.8	2124	3456
10	13123.2	17232.2	2342	3443
11	13213.9	14343.7	2651	3983
12	12322.8	17623.7	2213	3436
Average:	12850.65	17218.42	2387.42	3646.08

2. Composite service has 80 tasks and one execution path.

For example, in a static environment, a composite service execution with 40 tasks spends: (i) 1.6 seconds for selecting Web services using global planning; (ii) 0.7 seconds using local optimization. In a dynamic environment however, global planning imposes a perceivable overhead. For example, a composite service with 80 tasks spends 190 seconds for selecting Web services using global planning, while it spends 4.9 seconds using local optimization.

These results reinforce the conclusions of the analytical considerations of Section 4.3. If there is no requirement for specifying global constraints, then local optimization is preferable, especially in dynamic environments. On the other hand, global planning is superior when it comes to selecting services that satisfy certain global constraints and which optimize global tradeoffs. Given the fact that, a global selection approach considers both local and global selection constraints, the results clearly demonstrate the benefit of using a LP-based method for global service execution planning, even if we take into consideration the modest planning overhead.

7 Related Work

Web service composition is a very active area of research and development [3, 9]. In this section, we first review QoS management in middleware, then look at some related standards and service composition prototypes.

QoS management is widely discussed in middleware systems [1, 24, 15]. The focus of these work is essentially on the following four issues: QoS specification to allow description of application behavior and QoS parameters; QoS translation and compilation to translate specified application behavior into candidate application configurations for different resource conditions; QoS setup to appropriately select and instantiate a particular configuration; and finally, QoS adaptation to runtime resource fluctuations. Most efforts in QoS-Aware middleware are centered on the network transport and system level. Very limited work has been done at the business process level.

Not much work has been done on QoS-driven service composition. Several standards that aim at providing infrastructure to support Web services composition have recently emerged including SOAP [29], WSDL [34], UDDI [30], and BPEL4WS [2]. SOAP defines an XML messaging protocol for communication among services. WSDL is an XML-based language for describing web service interfaces. UDDI provides the directory and a SOAP-based API to publish and discover services. BPEL4WS provides a process-based language for services composition. It should be noted that the above standards are complementary to our approach. Our approach builds upon the building blocks of these standards (e.g., SOAP, UDDI) to provide a quality-driven and dynamic service composition model. In fact, our proposal could be applied for composite services specified in BPEL4WS, as statecharts basically support the same set of control-flow primitives as BPEL4WS.

Notations for service description and composition have also been proposed in ebXML and DAML-S. DAML-S supports the description of services based on a generic ontology. This ontology supports the specification of composite services as well as preconditions and postconditions on service operations. Unlike our proposal however, DAML-S does not aim at defining specific QoS criteria, nor does it address the issue of dynamic service selection using these criteria.

There are some standardization efforts in the area of QoS of Web services, most notably WSLA (Web Service Level Agreement) [18]. WSLA focuses on specifying and monitoring service level agreements for Web services. However, it does not address the modeling and management of the QoS of composite services.

Previous work has investigated dynamic service selection based on user requirements. Related projects include CMI [14] and eFlow [8]. CMI's service definition model features the concept of a *placeholder activity* to cater for dynamic composition of services. A placeholder is an abstract activity replaced at runtime with a concrete activity type. A selection policy is specified to indicate the activity that should be executed in place of the placeholder. In eFlow, the definition of a service node contains a *search recipe* represented in a query language. When a service node is invoked, a search recipe is executed in order to select a reference to a specific service. However, Market-based workflow, CMI and eFlow focus on optimizing service selection at a single task level. In addition, no QoS model is explicitly supported. Our approach focuses on optimizing service selection at a composite service level. Based on a generic QoS model, a novel service selection approach that uses integer programming techniques has been proposed.

Some work on QoS has been done in the area of workflow. In general, most existing projects in this area focus on specifying and enforcing temporal constraints [12, 5, 15]. Other projects such as METEOR [7] and CrossFlow [21] consider more comprehensive QoS models. METEOR [7] considers four quality dimensions, namely time, cost, reliability and fidelity. However, this work does not focus on the dynamic composition of services. It focuses on analyzing, predicting, and moni-

toring QoS of workflow processes. CrossFlow proposes the use of continuous-time Markov chain to estimate execution time and cost of a workflow instance. It should be noted that these two latter efforts are complementary to ours, insofar as they do not deal with dynamic selection of services. [20] does propose modeling primitives for capturing "points of flexibility" in workflow models where dynamic selection can occur, but does not provide specific approaches or algorithms for performing this selection as in our work.

Other complementary work include [23] and [25] which consider data quality management in cooperative information systems. They investigate techniques to select the best available data from various service providers based on dimensions such as accuracy, completeness, and consistency.

8 Conclusion

AgFlow is a QoS-aware middleware supporting quality driven Web service composition. The main features of the AgFlow system are:

- A service quality model to evaluate overall quality of Web services.
- Two alternative service selection approaches for executing composite services.

AgFlow has been implemented as a platform that provide tools for: (i) defining service ontologies; (ii) specifying composite services using statecharts; (iii) assigning services to the tasks of a composite service. The AgFlow platform has been used to validate the feasibility and benefits of the proposed approaches. In particular, relatively large composite services integrating a large number of candidate component services have been created and used to conduct experiments. The results of these experiments are encouraging: the computational cost of the planning phase for a composite service with 80 tasks is about 8 seconds in a static environment and about 190 seconds in a dynamic environment. In addition, the experiments have shown that the global planning approach leads to better QoS, and specifically, to lower execution prices and execution durations.

References

[1] C. Aurrecoechea, A. T. Campbell, and L. Hauw. A Survey of QoS Architectures. *Multimedia Systems*, 6(3):138–151, 1998.

[2] BEA Systems, Microsoft, and IBM. Business Process Execution Language for Web Services. `ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf`, 2002.

[3] B. Benatallah and F. Casati, editors. *Distributed and Parallel Database, Special issue on Web Services*. Kluwer Academic Publishers, 2002.

[4] B. Benatallah, M. Dumas, Q. Z. Sheng, and A. H. Ngu. Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 297–308, San Jose CA, USA, February 2002. IEEE Press.

[5] C. Bettini, X. Wang, and S. Jajodia. Temporal Reasoning in Workflow Systems. *Distributed and Parallel Databases*, 11(3):269–306, 2002.

[6] H. C.-L and K. Yoon. *Multiple Criteria Decision Making*. Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, 1981.

[7] J. Cardoso. Quality of service and semantic composition of workflows. *Ph.D Thesis, University of Georgia*, 2002.

[8] F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–162, May 2001.

[9] F. Casati, M.-C. Shan, and D. Georgakopoulos, editors. *VLDB Journal, Special issue on E-Services*. Springer-Verlag, 2001.

[10] F. Curbera et al. Unraveling the Web Services: an Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, Mar/Apr issue 2002.

[11] M. Dumas and A. t. Hofstede. UML activity diagrams as a workflow specification language. In *Proceedings of the International Conference on the Unified Modeling Language (UML)*, pages 86–90, Toronto, Canada, October 2001. Springer Verlag.

[12] J. Eder, E. Panagos, and M. Rabinovich. Time Constraints in Workflow Systems. *Lecture Notes in Computer Science*, 1626, 1999.

[13] M.-C. Fauvet, M. Dumas, and B. Benatallah. Collecting and Querying Distributed Traces of Composite Service Executions. In *Proceeding of the 10th International Conference on Cooperative Information Systems (CoopIS)*, Irvine CA, USA, 2002.

[14] D. Georgakopoulos, H. Schuster, A. Cichocki, and D. Baker. Managing Process and Service Fusion In Virtual Enterprises. *Information System, Special Issue on Information System Support for Electronic Commerce*, 24(6):429–456, 199.

[15] M. Gillmann, G. Weikum, and W. Wonner. Workflow management with service quality guarantees. In *Proc. of the ACM SIGMOD international Conference on Management of Data*, pages 228–239, Madison WI, USA, June 2002. ACM Press.

[16] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[17] H. Karloff. *Linear Programming*. Birkhauser, 1991.

[18] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Technical Report RC22456, IBM research, New York, 2002.

[19] B. Kiepuszewski, A. ter Hofstede, and C. Bussler. On structured workflow modelling. In *Proc. of the International Conference on Advanced Information Systems Engineering (CAiSE)*, Stockholm, Sweden, June 2000. Springer Verlag.

[20] J. Klingemann. Controlled flexibility in workflow management. In *Proc. of the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 126–141, Stockholm, Sweden, June 2000. Springer Verlag.

[21] J. Klingemann, J. Wäsch, and K. Aberer. Deriving Service Models in Cross-Organizational Workflows. In *Ninth International Workshop on Research Issues in Data Engineering: Virtual Enterprise, RIDE-VE'99*, Sydney, Australia, March 1999.

[22] S. Martello and P. Toth. *Knapsack Problems : Algorithms and Computer Implementations*. John Wiley and Sons, 2001.

[23] M. Mecella, M. Scannapieco, A. Virgillito, R. Baldoni, T. Catarci, and C. Batini. Managing Data Quality in Cooperative Information Systems. In *Proc. of the 10th International Conference on Cooperative Information Systems (CoopIS)*, Irvine CA, USA, 2002.

[24] K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li. QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments. *IEEE Communications Magazine*, 39(11):2–10, 2001.

[25] F. Naumann, U. Leser, and J. C. Freytag. Quality-Driven Integration of Heterogenous Information Systems. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 447–458, Edinburgh, UK, 1999.

[26] J. O'Sullivan, D. Edmond, and A. ter Hofstede. What's in a Service? *Distributed and Parallel Databases*, 12(2–3):117–133, September 2002.

[27] G. Oulsnam. Unravelling structured programs. *The Computer Journal*, 25(3):379–387, 1982.

[28] M. Pinedof. *Scheduling: Theory, Algorithms, and Systems (2nd Edition)*. Prentice Hall, 2001.

[29] Simple Object Access Protocol (SOAP) . `http://www.w3.org/TR/SOAP`.

[30] Universal Description, Discovery and Integration of Business for the Web, 2000. http://www.uddi.org.

[31] A. van Moorsel. Metrics for the Internet Age: Quality of Experience and Quality of Business. Technical Report HPL-2001-179, HP Labs, August 2001. Also published in 5th Performability Workshop, September 2001, Erlangen, Germany.

[32] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller. METEOR-S WSDI: A scalable infrastructure of registries for semantic publication and discovery of web services. *Journal of Information Technology and Management*, 2(3), 2003. To appear.

[33] D. D. Wackerly, W. Mendenhall, and R. L. Scheaffer. *Mathematical Statistics with Application*. Duxbury Press, 1996.

[34] Web Services Description Language (WSDL). `http://www.w3.org/wsdl`.

[35] L. Zeng. Dynamic Web Services Composition. 2003. Ph.D Thesis, University of New South Wales.

[36] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality Driven Web Services Composition. In *Proceedings of the 12th international conference on World Wide Web (WWW), Budapest, Hungary*. ACM Press, May 2003.

[37] L. Zeng, B. Benatallah, and A. H. H. Ngu. On Demand Business-to-Business Integration. In *Proceedings of Ninth International Conference on Cooperative Information Systems*, Trento, Italy, 2001. Springer Verlag.

[38] L. Zeng, B. Benatallah, A. H. H. Ngu, and P. Nguyen. AgFlow: Agent-based Cross-Enterprise Workflow Management System (Demonstration Paper). In *Proceedings of 27th International Conference on Very Large Data Bases*, Roma, Italy, 2001. Morgan Kauffmann.

Appendix

```
< ontology-service NAME=”Trip-planning-services” VERSION=”1.0” >
   < domain > tourism < /domain >
   < domainSynonym > leisure < /domainSynonym >
   < domainSynonym > trip < /domainSynonym >
   < domainSynonym > journey < /domainSynonym >
   < domainSynonym > travel < /domainSynonym >
   < superDomain > ROOT < /superDomain >
   < variable NAME=BonusPoint TYPE=integer>
   < variable NAME=DrivingTime TYPE=real>
   < variable NAME=Discount TYPE=real>
   <serviceclass NAME = ”FlightTicketBooking”
      SUPERCLASS-OF=”domestic-ticket-booking-service, intl-ticket-booking-service” >
      <serviceDescription > This is a service for booking the flight ticket </serviceDescription >
      <attribute NAME = ”serviceProvider” TYPE= ”string” > </attribute>
      <attribute NAME = ”url” TYPE=”string”> </attribute>
      <operation NAME = ”FindTicket”>
         < inputData NAME= ”DepartingAirport” TYPE=”String” </inputData>
         ...
         < outputData NAME= ”availability” TYPE=”boolean” </outputData>
      </operation>
      <operation NAME = ”Book-ticket”>
         < inputData NAME= ”Flight-schedule” TYPE=”XMLDoc” </inputData>
         ...
         < outputData NAME= ”Ticket-receipt” TYPE=”XMLDoc” </outputData>
         < outputData NAME= ”Confirmation-no” TYPE=”String” </outputData>
      </operation>
   </serviceclass>
   <serviceclass NAME = ”AccomodationBooking”>
      <attribute NAME = ”Hotel” TYPE=”Accomodation”> </attribute>
      <attribute NAME = ”Url” TYPE=”string”> </attribute>
      <operation NAME = ”HotelBooking”>
         ...
   </serviceclass>
   <serviceclass NAME = ”Car-rental-services”>
      <operation NAME = ”CarRental”>
         ...
   </serviceclass>
</ontology-service>
```

Table 4: Simplified service ontology for trip planning

```
<tModel tModelKey="uuid: 84fe307a-fe3e-4fff-a9fb-79140b265177">
    <name>Trip-planning-services</name>
    <description lang="en">XML specifications of service ontology</description>
    <overviewDoc>
        <description lang="en">Service Ontology</description>
        <overviewURL>http://dyflow.cse.unsw.edu.au/ontology/trip-planning-services.xml</overviewURL>
    </overviewDoc>
    <categoryBag>
        <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
            keyName="uddi: A specification" keyValue="specification"/>
        <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
            keyName="uddi: An XML specification" keyValue="xmlSpec"/>
        <keyedReference tModelKey="uuid:7341164E-FC52-4813-9810-22270DB32E0E"
            keyName="uddi: An XML specification for Service Ontology" keyValue="serviceOntologySpec"/>
        <keyedReference tModelKey="uuid:1FC3CA8C-1742-4EF5-B8F0-E93B5D84FDDB"
            keyName="uddi: An XML specification for Parent Service Ontology" keyValue="parentServiceOntologySpec"/>
    </categoryBag>
</tModel>
```

Table 5: tModel for a service ontology

```
<wsdl:definitions name="flightBooking"
targetNamespace="http://dyflow.cse.unsw.ede.au/services/flightBooking.wsdl"
xmlns="http://dyflow.cse.unsw.ede.au/services/flightBooking.wsdl"
xmlns="http://dyflow.cse.unsw.ede.au/ontology/Trip-planning-services.xml"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <xsd:schema targetNamespace="http://dyflow.cse.unsw.ede.au/services/flightBooking.wsdl">
    <xsd:element name="Flight-schedule" type="xsd:string"/>
    ... ...
    <xsd:element name="NumberofPassenger" type="xsd:integer"/>
    <xsd:complexType name="FlightTicketBooking_outParametersType">
      <xsd:sequence>
        <xsd:element name="Flight-schedule" type="xsd:XMLDoc"/>
        ... ...
      </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="FlightTicketBooking_outParameters" type="FlightTicketBooking_outParametersType"/>
    <xsd:complexType name="FlightTicketBooking_inParametersType">
      <xsd:sequence>
        <xsd:element name="DepartingAirport" type="xsd:string"/>
        ... ...
      </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="FlightTicketBooking_inParameters" type="FlightTicketBooking_inParametersType"/>
      </xsd:schema>
  </wsdl:types>
  <wsdl:message name="FlightTicketBookingSoapOut">
    <wsdl:part element="FlightTicketBooking_outParameters" name="Parameters"/>
  </wsdl:message>
  <wsdl:message name="FlightTicketBookingSoapIn">
    <wsdl:part element="FlightTicketBooking_inParameters" name="Parameters"/>
  </wsdl:message>
  <wsdl:portType name="flightBookingPortType">
    <wsdl:operation name="FlightTicketBooking">
      <wsdl:input message="FlightTicketBookingSoapIn" name="FlightTicketBookingInput"/>
      <wsdl:output message="FlightTicketBookingSoapOut" name="FlightTicketBookingOutput"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="flightBookingSOAP" type="flightBookingPortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="FlightTicketBooking">
      <soap:operation soapAction="http://dyflow.cse.unsw.ede.au/services/FlightTicketBooking" style="document"/>
      <wsdl:input name="FlightTicketBookingInput">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="FlightTicketBookingOutput">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
    </wsdl:binding>
  </wsdl:definitions>
```

Table 6: Simplified WSDL document for a Web service

```
<tModel tModelKey="uuid: 9760f81e-badd-492b-99ee-77ea408f6645">
    <name>FlightTicketBooking</name>
    <description lang="en">XML specifications of a Web service </description>
    <overviewDoc>
        <description lang="en">Web service description</description>
        <overviewURL>http://dyflow.cse.unsw.edu.au/services/flightBooking.wsdl</overviewURL>
    </overviewDoc>
    <categoryBag>
        <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
            keyName="uddi: A specification" keyValue="specification"/>
        <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
            keyName="uddi: An XML specification" keyValue="xmlSpec"/>
        <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
            keyName="uddi: types" keyValue="wsdlSpec"/>
        <keyedReference tModelKey="uuid:84FE307A-FE3E-4FFF-A9FB-79140B265177"
            keyName="uddi: An XML specification for Service Ontology" keyValue="serviceOntologySpec"/>
    </categoryBag>
</tModel>
```

Table 7: tModel for a web service