Adaptation in Web Service Composition and Execution

Girish Chafle Koustuv Dasgupta Arun Kumar Sumit Mittal Biplav Srivastava
IBM India Research Laboratory, New Delhi, India
{cgirish,kdasgupta,kkarun,sumittal,sbiplav}@in.ibm.com

Abstract

Web services simplify enterprise application integration by facilitating reuse of existing components for creating new services. In a dynamic environment, it is imperative to design a Web Service Composition and Execution (WSCE) system that adapts to failure of component services or changes in their QoS offerings. In this paper, we motivate a staged approach for adaptive WSCE ($\mathcal{A}$-WSCE) that cleanly separates the functional and non-functional requirements of a new service, and enables different environmental changes to be absorbed at different stages of composition and execution. We use Synthy, a prototype service creation environment, to implement our solution and demonstrate its effectiveness.

1 Introduction

The demand for quickly delivering new applications and adaptively managing them is a business imperative today. For example, given the intense competition in the telecom sector, mobile telephony service providers need to continually develop compelling applications to attract and retain end-users, with quick time-to-market. However, as the applications run, network configurations and QoS offerings may change, new service providers and business relationships may emerge and existing ones may be modified or terminated. The challenge, therefore, is to design robust and responsive systems that address these changes effectively while continually trying to optimize the operations of a service provider.

Web services assist the transition of businesses towards being on-demand and responsive by facilitating seamless business–to–business or enterprise application integration [13]. In our previous work, we studied the problem of Web Service Composition and Execution (WSCE). Specifically, we proposed a principled web service composition approach [1, 2] that drives the process from specification to deployment and execution of a composite workflow. The solution introduces a differentiation between web service *types* and *instances*, and uses efficient techniques to create the desired functionality as well as select components that optimize the non–functional attributes.

Although our previous work provides an end–to–end service composition environment, it lacks the ability to adapt to changes. We believe that this is a critical requirement in today's dynamic business environments. A naive approach to handle the problem is to re–compose from scratch every time a change occurs. However, this may not be a feasible solution due to multiple reasons: (i) a significant amount of intervention from the developer (ii) inability to cope up with the rate at which changes occur, or simply (iii) the complexity involved in exploring the search space for composition.

With this in mind, we present a novel solution for creating and executing Web services, that adapts to changes in the operating environment at various stages of composition, deployment and runtime. Key to our approach is the use of multiple workflows that realize the same composition in terms of functional and non–functional requirements. In case of a failure or QoS changes, the system recovers by replacing the failed components (or workflows) with available alternatives. We provide a feedback mechanism between the stages, along with a ranking function at each stage, to guide the selection of alternatives based on operating conditions.

Our main contributions are: (a) A characterization of the problem of Adaptive Web Service Composition and Execution($\mathcal{A}$–WSCE). (b) A staged solution that allows adaptation by generating multiple workflows at different stages, and selectively deploying them based on feedback mechanisms and suitable ranking functions. (c) Incorporation of the proposed system within an end–to–end service creation environment that demonstrate the effectiveness of the solution. Note that, the proposed solution is complementary to the various approaches for handling business and system related faults.

2 Motivation

Consider a travel agency service that uses services offered by various airlines, railways, road transport companies, hotels, car rental services and courier companies to provide its own Travel Planner service. To satisfy a user's travel request, the travel agency triggers a workflow that first books travel tickets, followed by booking accommodation in a hotel and finally renting a car. Once done, the reservation confirmations are delivered through courier.

Consider now the composite workflow for the Travel Planner service. For each action (service type) in the template for this workflow, there may be multiple individual services available from different vendors. These alternative instances offer same functionality but may vary in terms of their non–functional capabilities. For example, different airlines may offer a flight from New Delhi to New York but may vary in terms of flight time, ticket price and no. of stopovers. Further, the QoS values of the service instances can change dynamically depending upon the run–time conditions at the service provider's site. In such cases, it might be imperative to switch to either a new instance of the same type or to an altogether new service providing similar functionality. If one transport service (say, air travel) fails to provide service (e.g. tickets may not be available due to over booking) then it might be useful to switch to another one (rail, for instance) that provides similar functionality.

To incorporate changes in the environment, adaptation in web service composition and execution can be carried out at various levels. At one level, alternative instances for failed service can be explored. These would offer the same functionality but may differ w.r.t. QoS guarantees. In situations where none of the available instances are able to meet the QoS requirements, a different template (comprising of different types to meet the same functionality) needs to be tried. This is another level of adaptation. Intuitively, the frequency of corrective actions requiring replacement of faulty instances would be relatively more compared to actions requiring template replacement. Similarly, the need to account for changes in QoS values of the instances would be relatively more frequent as compared to need to replace instances altogether. Thus, to adapt effectively to changes in environment: (1) A composite service needs to be responsive to external events, (2) The adaptation technique should account for different rates of changes in the environment, and (3) The adaptation needs to be applied *incrementally* so as to avoid the prohibitively expensive overheads of recomposition from scratch.

3 Adaptive WSCE

Web service composition and execution is the process of realizing the requirements of a new Web service using existing component services, and executing the composite service on an execution infrastructure. The requirements are specified by an end–user, and can be decomposed into two parts. The first part deals with the desired functionality of the composite service, called the *functional requirements*. The second part involves specification of the *non-functional requirements* that relate to issues like performance and availability. The input is specified using an appropriate language, e.g., OWL expression for functional requirements, Quality-of-service (QoS) specification for non–functional requirements.

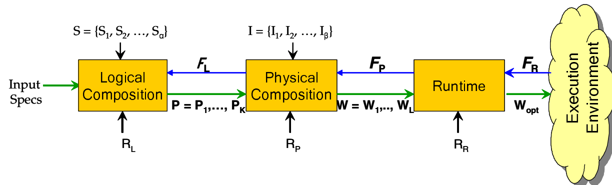


Figure 1. $\mathcal{A}$-WSCE: Solution Overview

We differentiate between web service *types*, which are groupings of similar (in terms of functionality) web services, and the actual web service *instances* that can be invoked. We believe that the separate representation of type definitions from instance definitions helps in handling different requirements separately, and allows us to work efficiently with a large collection of services. Types and instances can be advertised in a registry (like UDDI). We denote,

1. **S** = $\{S_1, ..., S_\alpha\}$: Set of α web service types.
2. **I** = $\{I_1, ..., I_\beta\}$: Set of β service instances. Assuming each service type has M instantiations, $\beta = M \times \alpha$.

Problem Definition:
The Adaptive Web Service Composition and Execution ($\mathcal{A}$-WSCE) problem can be defined as follows: Given the specifications of a new service, create and execute a workflow that satisfies the functional and non-functional requirements of the service, while being able to *continually* adapt to dynamic changes in the environment.

3.1 Solution Overview

We follow a staged approach where the functional composition (based on types) is decoupled from the non–functional composition (based on instances). While doing so, we increase the number of choices by passing multiple (feasible) compositions across the stages. Finally, for adaptation, we intelligently select amongst the alternatives available at each stage based on the operating conditions. Figure 1 gives an overview of our solution.

3.1.1 Creating Alternatives for Adaptation

The composition first proceeds to generate abstract workflows (*templates*[1]) based on service types (**Logical Composition**), which are subsequently concretized into executable workflows by selecting the appropriate web service instances (**Physical Composition**). We denote,

1. **P** = $\{P_1, ..., P_K\}$: Set of K abstract workflows selected after logical composition.

2. **W** = $\{W_1, ..., W_L\}$: Set of L executable workflows selected after physical composition.

Note that, a composition approach that does not use the distinction between types and instances can build the executable workflow set **W**, with each workflow of maximum length λ, by searching in O($L.\beta^\lambda$) space. The staged approach structures its solution search to a space of $O(K.\alpha^\lambda) + O(L.M^\lambda)$. This provides scalability to the composition process.

Having generated multiple workflows at each stage, we need a suitable way to *rank* them to aid in selection. We believe that the ranking function should take into account (a) the structural properties of the workflow and (b) its predicted (or monitored) performance based on operating conditions. With this in mind, we introduce three ranking functions, $\mathcal{R}_L$, $\mathcal{R}_P$, and $\mathcal{R}_R$ which work at the Logical, Physical and Runtime stages, respectively.

Function $\mathcal{R}_L$: assigns a score to each abstract workflow P_i. It can be defined using – length of the workflow, user comprehensibility of the workflow etc. While generating **P**, we can select the *best* K abstract workflows in accordance with $\mathcal{R}_L$.

Function $\mathcal{R}_P$: assigns a score to each executable workflow W_i. To define this function, we can use the QoS values of the instantiated workflows in **W**. $\mathcal{R}_P$ can be used to select the *best* L workflows to be included in **W**.

Function $\mathcal{R}_R$: helps in picking the *best* workflow, W_{OPT}, for execution based on *current* QoS of the component service instances in **W**.

3.1.2 Selecting Alternatives for Adaptation

While a workflow is executing, different changes occur in the environment. Some of these changes might affect the executing workflow, through either component services failure or degradation of their QoS values. When a workflow fails, we first explore other available options to replace the faulty one. Otherwise, we exploit the staged nature of composition to initiate an "incremental" recovery process. Specifically, if all available workflows at the runtime stage fail, we generate a fresh set of executable workflows **W** by invoking the physical composition. If at any time the physical composition stage fails, a new set of abstract workflows **P** are generated by logical composition.

Note that, new functionality in terms of new service types being available can be communicated to the logical composition stage. Similarly, if new instances corresponding to an already existing service type come up, the same can be incorporated at the physical composition stage. Transient variations of the QoS values of executing services can be monitored and periodically aggregated at the runtime stage. To build a truly adaptive system, we prescribe that each stage performs periodic *health* checks to evaluate the current set of alternatives, and change them (if required) based on changes in the environment. To aid the process, we define a feedback mechanism in our solution with the help of the following feedback functions.

Function $\mathcal{F}_R$: carries information about measured QoS values and failure of services from the execution environment to the runtime stage. It helps $\mathcal{R}_R$ to select W_{OPT} from **W**.

Function $\mathcal{F}_P$: carries aggregated QoS information over a period of time to the physical stage. Such updates help in effectively ranking the workflows in **W** by invoking $\mathcal{R}_P$.

Function $\mathcal{F}_L$: carries information about service types to the logical stage. If all instances of a particular service type are found to be unavailable, this service type can be removed from consideration for generating further templates, till new instances become available.

3.2 Understanding the Search Space

In AI, the planning discipline looks at how plans can be automatically generated given a set of actions, the initial state and the desired goal state. Web service composition can be seen as a planning problem where each service corresponds to actions, and the initial and goal states are suitably defined to correspond to the requirements of the new service. In this context, we use the established framework of refinement search[10] to shed some light on the adaptation process. The semantics of a plan P in [10] is formalized in terms of its *candidate set*, denoted by $\langle\langle P \rangle\rangle$, which is the set of sequence of instantiated actions that are compatible with the constraints of the plan. The purpose of a planning algorithm is to find *minimal* candidates[2].

Controlling the search space of adaptation: The web service type corresponds to an action and the web service instance corresponds to an executable, instantiated action. After the Logical Composition stage, we generate a set **P** of K selected abstract workflows. Each abstract workflow P_i has $|P_i|$ service types in it and its candidate set is $\langle\langle P_i \rangle\rangle$.

[1] A workflow is *abstract* if it does not associate service instances to the actions specified, otherwise it is *executable*. We use the terms plan, template and abstract workflow interchangeably.

[2] A candidate is minimal if removal of any of its constituent instances would lead to an unsound, incorrect execution.

After logical composition, the search space of physical composition is restricted to the candidate set of $\langle\!\langle \mathbf{P} \rangle\!\rangle = \sum \langle\!\langle P_i \rangle\!\rangle$. In this stage, each plan is instantiated into a set of executable workflows by replacing the web service types with instances, the output being $\langle\!\langle \mathbf{W} \rangle\!\rangle$ ($\subseteq \langle\!\langle \mathbf{P} \rangle\!\rangle$) *candidates for execution*. In the Runtime stage, only one of the workflows has to be run. In the event of change, the executor needs to consider only the L pre–selected candidates for *immediate* adaptation [3]. If it is not able to meet the end-to-end QoS from **W**, the executor will inform the Physical Composition stage. The latter does not have to consider the full search space and only focus on $\langle\!\langle \mathbf{P} \rangle\!\rangle$ - $\langle\!\langle \mathbf{W} \rangle\!\rangle$ candidates for immediate adaptation. If the Physical Composition stage is not able to generate a workflow that meets the end-to-end QoS from **P**, it will inform the Logical Composition stage. The latter does not have to consider the full search space and can remove $\langle\!\langle \mathbf{P} \rangle\!\rangle$ from its immediate consideration.

Generation and selection of multiple workflows: While passing multiple workflows across stages, one would want to (a) limit the number of workflows passed and (b) minimize the number of times a subsequent stage refers to the previous stage for purposes of recovery from failures and adaptation to changing QoS. A problem similar to generation of multiple plans is faced in case–based reasoning and recommender systems about how to select the alternatives and retrieve cases or recommendations to the user. The established solution there is to choose a diverse set (dissimilar cases) from the case-base but retrieve cases back to the user based on similarity [16]. For us, this translates to finding set of workflows (**P**, **W**) that have maximum candidate sets. However, finding such optimal workflow sets would not be easy [18]. In practice, we can model similarity and dis-similarity using simple *Hamming-distance* that measures the number of different web service types or instances in the workflows.

4 System Design

We describe the system architecture for our solution. The issues we have to address are incorporating feedback into the system, generating multiple workflows at different stages, ranking the workflows and selecting the best.

4.1 Architecture

As shown in Figure 2, the **Logical Manager** and the **Physical Manager** are configured with details of available web service types and instances respectively. The **Runtime Manager** maintains a registry that receives information

[3]The L plans cannot be removed from all future considerations because the change (e.g., failure) can get reversed in future and then, these plans get back as selection contenders.

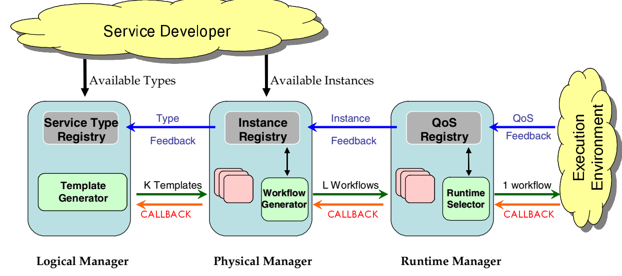


Figure 2. $\mathcal{A}$–WSCE: System Architecture

about QoS changes and/or failures of service instances through a monitoring component.

During the composition process, the Logical Manager invokes the `Template Generator` to create K workflow templates that meet the functional requirements of the desired service. These templates are stored by the Physical Manager and instantiated into executable workflows by selecting an appropriate service instance for each service type in a template. The selection process is based on optimization of the non–functional requirements of the composite service, and is performed by the `Workflow Generator` in the Physical Manager. L executable workflows are passed from the Physical Manager to the Runtime Manager, which stores them and invokes `Runtime Selector` to select the best one (in terms of overall composite QoS).

To make the system sensitive to changes in the environment, there is a *feedback channel* associated with each stage. For example, the execution environment provides periodic feedback about monitored QoS values of instances to the Runtime. The QoS changes are aggregated and sent periodically by the Runtime Manager to the Physical Manager. Also, if the instances of a particular service type are unavailable for a certain period, the information is passed by Physical Manager to Logical Manager. This ensures that this type is not used by the Template Generator in subsequent template creations. Whenever the failed instance(s) recovers, the information is fed back to the previous stages, thereby qualifying them again for inclusion in the composition process. Finally, information about availability of new service types and instances is fed into the *Type* and *Instance* registries, respectively.

Our system also provides a *callback mode* for situations where all available workflows at a given stage have failed. This mode is used by the Runtime Manager to invoke the Physical Manager if no available workflows can be executed (due to failure of instances and/or QoS violations). It is also used by the Physical Manager to request the Logical Manager for generating new templates, if none of the available workflows can be concretized to executable ones.

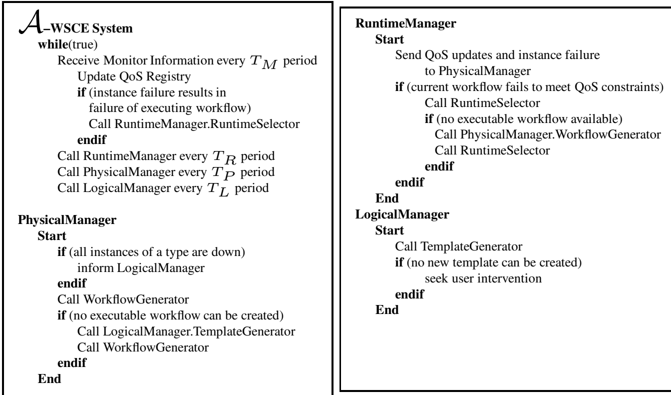


Figure 3. $\mathcal{A} - WSCE$: Algorithm for Adaptation

4.2 Algorithm for Adaptation

The adaptation algorithm summarized in Figure 3 tries to incorporate different changes in the environment at different stages. The monitoring infrastructure sends information about QoS changes and instance failures to the run-time stage every T_M interval. The Runtime Manager runs every T_R interval to incorporate the QoS changes and select the best workflow to execute. Physical Manager, on the other hand, is invoked every T_P period to generate a fresh set of executable workflows which are then passed to the Runtime Manager. Finally, the Logical Manager creates new templates every T_L period. We believe that in a realistic setting, $T_M < T_R < T_P < T_L$. In addition, the *callback mode* is used when all available workflows at a particular stage have failed.

4.3 Generation of Multiple Workflows

4.3.1 Generation of Multiple Templates

Templates (abstract workflows) in the logical composition stage can be created either manually, or by using planning techniques that work on semantically annotated web services. In Synthy [1], a prototype architecture for end–to–end composition, we use contingent planning methodology to automate the task of generating templates for a given service specification.

In order to generate K templates, a decision has to be made about which K plans should be selected. In the absence of any other information, we can select the first K plans generated by the planner. Alternatively, the planner can generate all possible templates and the developer has the flexibility to choose K among them depending on her expertise and requirements. A third option is to randomly select a template and then recursively select subsequent templates that are *maximally distant* from the already included ones, using a Hamming distance. We employ this strategy in the current system.

4.3.2 Generation of Multiple Executable Workflows for a Template

The problem of generating multiple executable workflows can be divided into two steps. The first, handled in this subsection, deals with creation of multiple workflows for a single template. The second, detailed in the next subsection, is concerned with formation of **W** by selectively picking workflows corresponding to different templates.

We have developed an optimization framework [1] which helps us to generate an executable workflow for a template by selecting an instance for each type in the template. The basic idea is to optimize QoS while meeting the constraints specified for the composite service. Specifically, we optimize a normalized function, Q ($0 < Q < 1$), that captures the end–to–end values of each of the QoS dimensions (*cost*, *response time*, and *availability* in our case) according to some relative weights. The framework is built on the QoS model and mathematical formulation proposed by Zeng et al. [21]. Due to lack of space, we omit the details here.

The optimization framework is suited for generation of only one workflow for a given template (abstract workflow). However, we are presented with the challenge of generating multiple executable workflows ($\leq L$) for the same template[4]). Our solution is to intelligently remove a few instances from consideration while repeatedly invoking the optimization module, until (atmost) L templates are generated. We outline two strategies for this purpose:

$\mathbf{0-reuse}$: It does not allow any duplication of instances among the multiple workflows generated. Hence, two workflows selected with this strategy will have Hamming distance of λ between them (where λ is the length of the workflow). For implementation, we remove the services selected at each invocation of the framework from further consideration, the next invocation now essentially binds a new set of services to the template.

$\mathbf{k-reuse}$: It allows at most $\lambda - k$ duplication of instances among the workflows. In other words, two workflows selected with $k - reuse$ have a Hamming distance of at least k between them. For $k - reuse$, we randomly pick k services and prevent them from being used in subsequent workflows.

It is important to note that although we evaluate in this paper only the strategies outlined above, various others can be implemented and seamlessly incorporated into our composition and adaptation mechanism.

[4]It might not be possible to generate L workflows for a template

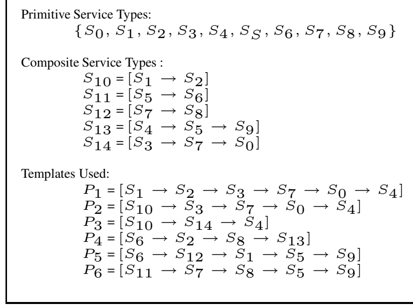


Figure 4. Service Types and Templates

4.3.3 Selection of Workflows from Different Templates

Having generated multiple workflows for each template, our task is now reduced to selecting L workflows from this set. We are driven by the motivation of maximizing the total sum of the QoS values of the selected workflows, while ensuring that each template gets a minimum representation in the selected set. Providing minimum representation for each template can help in providing more flexibility for selection of workflow to execute at the runtime stage. In the following, we propose this task of selection as an IP optimization problem, and use the $CPLEX$ solver [5] for the same.

Let Q_{ji} denotes the QoS of the executable workflow i for a template j. Let y_{ji} be an integer variable, such that y_{ji} is 1 if workflow i is selected for template j, 0 otherwise.

Objective Function

$$Maximize: \sum_{j=1}^{K} \sum_{i=1}^{L} y_{ji} * Q_{ji} \quad (1)$$

Constraints

Total Workflows: For the total number of workflows selected to be less than or equal to L we have the following constraint:

$$\sum_{j=1}^{K} \sum_{i=1}^{L} y_{ji} \leq L \quad (2)$$

Template Representation: There should be a minimum representation for each of the templates passed from logical to the physical stage. If κ denotes minimum representation for each template then we have the following constraint:

$$\sum_{i=1}^{L} y_{ji} \geq \kappa, \forall_{j=1,\ldots K} \quad (3)$$

5 Evaluation

To demonstrate the effectiveness of the proposed solution, we have implemented $\mathcal{A}$-WSCE, within the Synthy service composition framework [1]. Our main goal was to study how effectively $\mathcal{A}$-WSCE realizes a truly adaptive system for both service composition and execution.

To this end, we conducted an extensive set of experiments that tests the robustness and responsiveness of $\mathcal{A}$-WSCE. In the absence of standard benchmarks for WSCE systems, we performed our experiments in a simulated setting. The experimental parameters were chosen keeping in mind the dynamics of a real execution environment.

5.1 Experimental Setup

For our experiments, we consider the problem of composing and executing a Web service S, whose functional and non–functional requirements have been specified.

Types and Instances: For ease of simulation, we assume that there are 15 distinct service types (functionalities) in the domain – some are primitive types, while the others are composite types (i.e. obtained from functional composition of primitive types). Each service type has 5 different instances that are currently deployed. Each instance has QoS parameters (cost, availability and response time) specified according to a random distribution. Each instance of a particular service type S_i has a response time that follows a normal distribution with mean μ^i_{rt} and variance σ^i_{rt}. Similarly, the cost is assigned using a normal distribution with mean μ^i_{cost} and variance σ^i_{cost}. Finally, the availability of an instance is assigned using an exponential distribution with μ^i_{av}. The μ_i's for a type are chosen randomly from a predetermined range, i.e. [1000-1400], [100ms–200ms], and [0.6–0.9] for cost, response time, and availability, respectively[6]. The values for σ^i_{cost} and σ^i_{rt} are set to $500\$$ and $50ms$, respectively.

Templates and Executable Workflows: From the functional specifications, we first generate six templates (abstract workflows) for the composite service S, using the strategy described in Section 4. Table 4 gives the set of templates used for the experiment. We do not evaluate the quality of the templates, but focus on the ability of $\mathcal{A}$-WSCE to switch between them, depending on runtime conditions. Initially, the templates are used to stitch together L executable (composite) workflows that satisfy the non–functional requirements, and passed to the Runtime Manager. The Runtime Manager uses one of them to be executed at a given point of time. Periodically, the system performs *health* checks at each stage of composition, to ensure that the QoS commitments are met. In case of a failure or QoS violations, the system takes corrective measures (as detailed in Figure 3) and replaces the currently executing workflow with a potentially better one.

[5] http://www.ilog.com/products/cplex/

[6] The mean value of a composite type is obtained by aggregating the mean values of its primitive components [6].

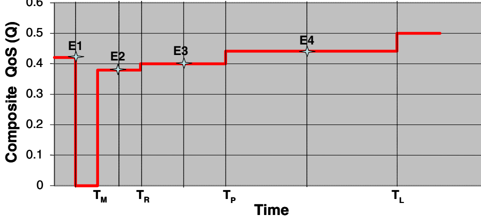


Figure 5. Responsiveness of $\mathcal{A}$-WSCE system

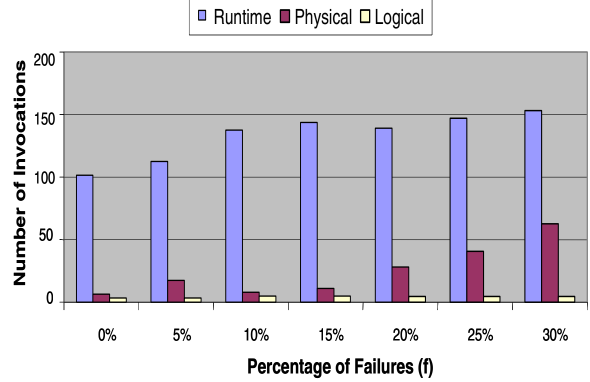


Figure 6. Robustness of $\mathcal{A}$-WSCE system

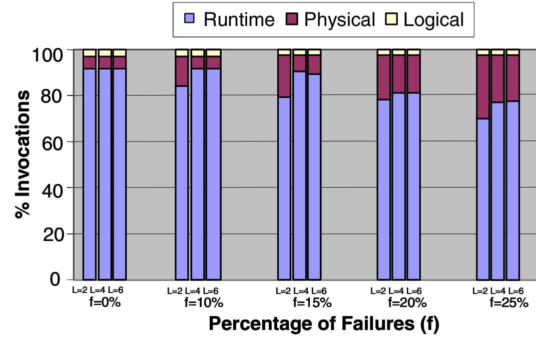


Figure 7. Effect of Multiple Executable Workflows

Runtime events: To test the robustness of the system, we periodically inject "change" events in the runtime. The events of interest are: (i) E_1–failure of an instance, (ii) E_2–QoS change of an instance, (iii) E_3–availability of a new instance, and (iv) E_4–availability of a new service type. For (i), we randomly pick a fraction of the total instances ($f\%$) and make them unavailable for a certain (random) amount of time. For (ii), we periodically pick 10 instances and change the QoS values, according to the distributions specified earlier. Finally, there are periodic events that instruct the different Managers to perform health checks. Note that, the periodicity at which these checks are performed are chosen to be $100T$ (T_R), $4800T$ (T_P), and $14400T$ (T_L), for the Runtime, Physical, and Logical Managers, respectively (where T corresponds to the duration of a clock tick in the simulation).

Using this setup, we next study the behavior of the $\mathcal{A}$-WSCE system and shed some light on some of these important questions: (1) Is the system responsive to QoS changes in the environment?, (2) Is the system robust to failure of instances?, (3) What are the trade–offs in passing multiple workflows from one stage to another?, (4) How do the $0 - reuse$ vs $k - reuse$ strategies for generating executable workflows, compare with each other?

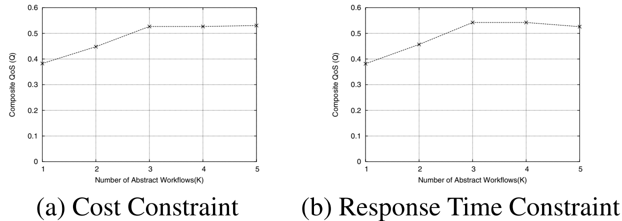
5.2 Results and Observations

We study the adaptive behavior of $\mathcal{A}$-WSCE in the the first experiment. We start with an initial composite workflow for S, and periodically introduce the runtime events discussed earlier. Figure 5 gives a snapshot of the system behavior and illustrates how $\mathcal{A}$-WSCE handles each of the changes in the execution environment. By looking at the composite QoS, one can verify how our solution continually strives for a better solution, if one is available. Note that, the time to respond to a change depends on the (pre–configured) frequency of health checks at the different Managers. In future, we plan to study mechanisms to dynamically schedule these checks, possibly based on the rate of change at each stage.

In the next experiment, we illustrate the robustness of $\mathcal{A}$-WSCE in terms of instance failures. Figure 6 shows the number of invocations of the Runtime, Physical and Logical stages (of composition) with increasing number of failures. Note that, the results are obtained with $K = 2$ and $L = 4$. We observe that most of the failures are handled at the Runtime and the number of "callbacks" to the Physical stage are significantly low, while the number is even smaller for "callbacks" to the Logical Manager. This shows that $\mathcal{A}$-WSCE can efficiently handle runtime failures, while avoiding the prohibitively expensive approach of composition from scratch.

Figure 7 shows the effect of varying the number of workflows (L) passed from the Physical to the Runtime stage. As before, we measure the percentage of callbacks to the Physical stage with varying intensity of failures. We observe that as L increases from 2 to 4, the number of callbacks are reduced–however, increasing the number to 6, leads to nominal savings in terms of callbacks. Given that the generation of each workflow involves potential intervention from the developer (for data type matching etc. [12]), it is desirable to keep the value of L small.

In the next experiment, we study the effect of varying



(a) Cost Constraint (b) Response Time Constraint

Figure 8. Effect of Multiple Abstract Workflows

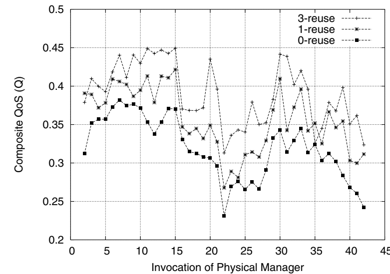


Figure 9. Comparison of Strategies

the number of templates (K) on the system behavior. Accordingly, we vary the number of templates that are passed from the Logical to Physical stage from $1, \cdots, 5$. For each case, we generate 4 executable workflows that are passed to the Runtime. For each value of K, we measure the average QoS of the composite service, in the presence of changes in the execution environment. Finally, the experiments are conducted in two settings–one in which the developer specifies a constraint in terms of the total cost, and the other with a constraint on total response time. Interestingly enough, the QoS improves with increasing number of templates, but the trend stops beyond a certain value of K (i.e. $K = 3$). This, in combination with our previous result, clearly points to the fact that there is value in carefully choosing the values of K and L, to ensure that the system continues to adapt to environmental changes, while incurring minimal composition overheads.

We compare the $0 - reuse$ and $k - reuse$ strategies in our final set of experiments. Our effort is to verify whether multiple workflows generated at the Physical stage should be *mutually disjoint* in terms of the instances. To do so, we consider $0 - reuse$, $1 - reuse$ and $3 - reuse$ and measure the average QoS of the (three) workflows, generated by each of the strategies. As shown in Figure 9, we note that the average QoS increases with increasing k. However, one should keep in mind that, the system becomes more prone to failures with the reuse of instances. In future, we plan to investigate ways to derive a suitable value of k, for a given class of business applications.

Summary Our experimental evaluation demonstrates that the $\mathcal{A}$-WSCE system is responsive to changes in the environment, and sufficiently robust to handle large number of failures. Furthermore, there is a clear benefit in passing multiple workflows from one stage to another, in terms of the adaptivity of the system. However, the amount of redundancy that is incorporated at each stage should be chosen carefully, such that it introduces minimal composition overhead while guaranteeing the desired level of resiliency.

6 Related Work

Grid Computing: Handling of faults has been addressed in the context of scientific workflows by the Grid Computing community. [20] provides a taxonomy of grid workflows with focus on workflow design, scheduling, fault management and data movement. Grid Workflow [8] is a flexible failure handling framework for Grid environments. Separate task level failure handling and workflow level techniques are used to respond effectively to these different kinds of failures. However, the developer needs to precisely specify the fault handling process. Our solution, on the other hand, automates the process of selecting an alternative workflow on the occurrence of a failure.

Workflow and Databases: An overview of fault handling in workflows is presented in [3]. Workflow transformations have been used in [7] to obtain semantically equivalent yet structurally different workflows. However, since the changes are only structural and the set of components remains static, it does not cater to situations such as fault handling and dealing with QoS variations. Romanovsky et al. [15] employ coordinated atomic actions to allow exception handling by associating handlers with such actions. They argue that these actions enable developers to deal with application-specific semantics more effectively. A two stage compiler and a rule-based run-time system is presented in [9]. Their workflow compiler determines data dependencies and translates the high-level workflow schema into a set of rules that can be interpreted by the run-time system. In Verma et al. [19], workflows are modeled as web processes and the adaptation approach is to reconfigure the web processes in case of failures, without violating process constraints. On the other hand, we present a staged approach to handle different environmental changes at different stages, including dynamic generation of new composite service templates.

Artificial Intelligence Techniques: The AI community has used planning and other techniques to provide exception handling in workflow systems. In [14], the authors present an exception handling mechanism based on infeasible workflows that integrates workflow failure and

contextual information for improved handling of exceptions. Our focus in the paper was on an orthogonal problem of adaptation by efficiently replacing failed instances or templates with available alternatives, and for web services. Berry et al [4] had identified earlier how techniques from AI in reactive control, scheduling and planning can be used in adaptive workflow engines to deal with ever-changing and unpredictable environments[4].

Distributed Systems Management: The systems management community has addressed the problem from the point of view of providing QoS guarantees, meeting SLAs and improving scalability of the system. A mechanism for enabling adaptation that responds to changes in the QoS values of running instances is presented in [11]. The adaptation mechanism is triggered off by predefined rules. However, the workflow is specified by the system administrator rather than composed semi-automatically since the focus there was towards general-purpose IT management. A CORBA based fault-tolerant application composition and execution environment for distributed applications has been presented in [17]. The Workflow Execution Service component provides reliable task scheduling having been built on top of CORBA Object Transaction Service. The system is applicable to long running workflows, where tasks eventually get completed provided failed nodes eventually recover and network partitions eventually heal. As part of the JOpera project, a platform for process execution has been designed to support autonomic scalability [5]. The system provides adaptation by reconfiguring an executing process and employs a three phase autonomic controller. However, the notion of workflow templates is missing and adaptation has scalability as the sole aim whereas our system provides fault-resilience as well as optimization of the executing composite service.

7 Conclusion

In this paper, we presented $\mathcal{A}$-WCSE, a staged approach for adaptive Web Service composition and execution. The proposed solution allows multiple workflows to be passed from one stage to another, which in turns enables the system to react effectively and incrementatlly to changes in the operating environment. The end result is a robust and responsive solution for composite Web services.

References

[1] V. Agarwal, G. Chafle, K. Dasgupta, N. Karnik, A. Kumar, S. Mittal, and B. Srivastava. Synthy: A system for end to end composition of web services. *J. Web Semantics,Vol.3:4*, 2005.

[2] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava. A Service Creation Environment based on End to End Composition of Web Services. In *Proc. 14th WWW*, May 2005.

[3] G. Alonso, C. Hagen, D. Agrawal, A. Abbadi, and C. Mohan. Enhancing the Fault Tolerance of Workflow Management Systems. *IEEE Concurrency*, 8(3):74–81, 2000.

[4] P. Berry and K. L. Myers. Adaptive Process Management: An AI Perspective. In *Proc. Workshop "Towards Adaptive Workflow Systems", Conference on Computer Supported Cooperative Work*, Seattle, USA, 1998.

[5] G. A. C. Pautasso, T. Heinis. Autonomic Execution of Service Compositions. In *Proc. of ICWS*, 2005.

[6] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of Service for Workflows and Web Service Processes. *Journal of Web Semantics*, 1:281–308, 2004.

[7] J. Eder and W. Gruber. A Meta Model for Structured Workflows Supporting Workflow Transformations. In *Proc. 6th ADBIS, UK.*, 2002.

[8] S. Hwang and C. Kesselman. GridWorkflow : A Flexible Failure Handling Framework for the Grid. In *Proc. of the 12th IEEE Intl. Symp. on HPDC*, 2003.

[9] M. Kamath and K. Ramamritham. Failure Handling and Coordinated Execution of Concurrent Workflows. In *ICDE*, 1998.

[10] S. Kambhampati. Refinement Search as Unifying Framework for Analyzing Planning Algorithms. In *Proc. KR*. 1994.

[11] A. Kumar, N. Karnik, and R. C.K. Moving from Data Modeling to Process Modeling in CIM. In *Proc. 9th Intl. Symp. on Integrate Network Management (IM)*, 2005.

[12] A. Kumar, B. Srivastava, and S. Mittal. Information Modeling for End to End Composition of Web Services. In *ISWC*, 2005.

[13] F. Leymann. Web Services: Distributed Applications Without Limits. In *Proc. Intl. Conf. BPM*, 2003.

[14] Z. Luo, A. P. Sheth, K. Kochut, and J. A. Miller. Exception handling in workflow systems. *Applied Intelligence*, 13(2):125–147, 2000.

[15] A. Romanovsky, P. Periorellis, and A. F. Zorzo. Structuring Integrated Web Applicatoins for Fault Tolerance. In *Proc. of the 6th ISADS*, Pisa, Italy, 2003.

[16] H. Shimazu. Expertclerk: Navigating shoppers' buying process with the combination of asking and proposing. In *Proc. of IJCAI'01*, Seattle, Washington, USA, 2001.

[17] S. K. Shrivastava and S. M. Wheater. A transactional workflow based distributed application composition and execution environment. In *Workshop on Support for composing distributed applications, SIGOPS98*.

[18] B. Srivastava, S. Kambhampati, M. B. Do, and T. Nguyen. Finding inter-related plans. In *Proc. ICAPS 2006 Workshop on Plan Analysis and Management*, 2006.

[19] K. Verma, K. Gomadam, A. P. Sheth, J. A. Miller, and Z. Wu. The Meteor-S Approach for Configuring and Executing Dynamic Web Processes. (TR6-24-05), 2005.

[20] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3), 2005.

[21] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality Driven Web Services Composition. In *Proc. 12th WWW*, May 2003.