DYNAMIC SERVICE COMPOSITION: A DISCOVERY-BASED APPROACH

JONATHAN LEE*, SHANG-PIN MA†, YING-YAN LIN‡, SHIN-JIE LEE§ and YAO-CHIANG WANG¶

Software Engineering Lab,
Department of Computer Science and Information Engineering,
National Central University, Taiwan

* yjlee@selab.csie.ncu.edu.tw
† albert@selab.csie.ncu.edu.tw
‡ slose@selab.csie.ncu.edu.tw
§ jilee@selab.csie.ncu.edu.tw
¶ ycwang@selab.csie.ncu.edu.tw

Service-Orientated Computing (SOC) has become a main trend in software engineering that promotes the construction of applications based on the notion of services. SOC has recently attracted the researchers’ attention and has been adopted industry-wide. However, service composition that enables one to aggregate existing services into a new composite service is still a highly complex and critical task in service-oriented technology. To enhance availability of composite services, we propose a discovery-based service composition framework to better integrate component services in both static and dynamic manner, including (1) to devise a notion of service availability especially for composition; (2) to develop a dynamic service composition (DSC) pattern for addressing the issues of service availability; and (3) to extend Contract Net Protocol (ECNP) to coordinate service discovery, composition and invocation based on the composite pattern. The main benefit of the proposed approach is better availability through attaching multiple candidate services for future binding.

Keywords: Contract net protocol; design patterns; service binding; service composition; service discovery; service-oriented architecture (SOA).

1. Introduction

Service-Oriented Computing (SOC) has become a main trend in software engineering that exploits both web services and Service-Oriented Architecture (SOA) as fundamental elements for developing applications. Web services are autonomous, platform-independent computational elements that can be described, published, discovered, and composed using standard protocols to build applications across various platforms and organizations. Recently, researches have been conducted in both industry and academia to alleviate issues for service-oriented technologies, including service discovery, service description and invocation (message passing), service composition, and service management.
Web service technologies are best exploited by composing web services. Composing multiple web services is more beneficial to users than a single service [15]. However, web service composition requires the component web services to be available and to behave correctly. Moreover, web services lack the flexibility to replace a failed service with an alternative [8]. Meanwhile, although availability of web services is a critical issue for building robust SOA-enabled applications, it has been largely neglected [4], especially for service composition.

Menascé and Almeida [17] defined availability as probability that a system is up, and unavailability as the probability that a system is down. PasPallis et al. [21] defined service availability as the percentage that a web service is available and functioning within its operational requirements. Many investigations into web services [4, 26, 1, 21] proposed methods to ensure or enhance web service availability. However, none of these focuses on the availability of service composition. Furthermore, no previous study has proposed neither appropriate measure for evaluating availability of service composition, nor systematic methodology for improving the availability of service composition.

To assure high-availability of service composition, this work argues that the problem caused by dynamics characteristic of web services, namely, services together with their communication links and information sources may appear and disappear dynamically in the Internet [7], should be carefully addressed by taking the following two requirements into account:

- A composite service must be capable of identifying possible component services with respect to a request that serve as replacements to one another. A composite service needs to reserve multiple interface-compliant or redundant services to ensure high-availability.
- A composite service must be capable of querying for new appropriate services dynamically and pro-actively, and re-binding the new service to complete original failed tasks while some component services are missing or malfunctional.

In order to realize these demands, this work proposes a discovery-based service composition framework to improve the integration of component services, and thus increase service availability, including three main features:

1. To define a notion of service availability, in particular, for service composition;
2. To develop a composite pattern, called Dynamic Service Composition (DSC) pattern, to help improve service availability; and
3. To devise an extension of Contract Net Protocol, namely, Extended Contract Net protocol (ECNP), to coordinate the activities of service discovery, service composition, and service invocation.

This paper is organized as follows. Section 2 depicts essential constructs of service composition. Section 3 discusses how to measure the availability of service composition. Section 4 describes the proposed Dynamic Service Composition pattern. Section 5 explains the Extended Contract Net Protocol in detail and presents
a traveling assistant service as an illustrative example. Section 6 outlines some extant research work related to the proposed approach. The final section presents the conclusion and our future work.

2. Constructs of Service Composition

To explore the key features, behaviors and processes of a composite service, two constructs: structure and lifecycle of service composition, are fully discussed below.

2.1. Structure of composite service

The building blocks of a composite service include three main parts: composition schema, service proxy and component service (see Fig. 1.)

- Composition schema describes the control flow and data flow between the component services in the composition, and can also include other factors such as exception handling, transaction and compensation.
- Component Services are the services synthesized in the composition. A composite service should contain more than one component service. A composite service can be as a component service of another composite service.
- Service proxy plays the roles of invoking and bridging component services. Composite service is simply a client of component services, and is not physically incorporated with the component services. Therefore, the composite service should explicitly or implicitly contain several service proxies to invoke component services. Additionally, each service proxy can bind various concrete services as long as they have the same service interface based on the loose coupling characteristic of service-orientation [10]. Accordingly, a “binding” relationship between service proxy and component service is added.
2.2. Service composition lifecycle

In addition to the static structure of a composite service, the service composition lifecycle is also devised to show phases of service composition. A full composition process includes the following phases:

- **Composite Service Definition**: to build the composite service schema or an executable composite service.
- **Service Deployment**: to deploy executable composite services in a service composition engine.
- **Service Execution**: to perform tasks of a composite service in the service execution engine that can host and run composite services.

It is noted that composition-related activities are insufficient on their own. Service discovery activities should also be integrated.

2.2.1. Static service composition

Static composition is the style in which services to be composed are determined at design time [9]. If only static composition is considered, then service discovery is only required in the “service definition” phase (see Fig. 2) to query for component services and produce a composite service. Interface matching (matching by inputs and outputs) and semantic or conditional matching (matching by non-functional conditions, such as cost and performance) are adopted according to the developer’s
needs. Static service composition enables developers to produce an executable composite service.

2.2.2. Dynamic service composition

Dynamic composition means that services to be composed are decided at run-time [9]. Dynamically discovering the best available services to fit the needs of a specific customer requires dynamic composition technology to be ready. All three composition phases (see Fig. 2) must rely on service discovery when applying dynamic composition:

- Composite Service Definition: the abstract composite service flow (i.e. the composition schema) needs to be produced before invoking the composite service. To construct the flow, interface matching is required to choose the interfaces of the component services linked in this composition.
- Service Deployment: all required service bindings for all service nodes must be ready in order to deploy the composition. Interface matching is required to locate interface-compliant component services. Semantic matching is also required to select appropriate concrete component services as default binding services. Ideally, at least one concrete component should be assigned to each service node.
- Service Execution: when the user approaches the composite service, a flow instance should be instantiated. Binding service can be determined based on semantic matching for this instance according to the user’s requirements. Besides, if some component service leaves or malfunctions during execution of service flow instance, semantic matching or interface matching can be re-performed.

3. Availability of Service Composition

To improve the availability of composite service, appropriate measures must be developed to compute numeric availability values to evaluate the proposed approach. Two complementary measures, namely service node availability and flow pattern availability, are defined to derive the availability of composite services.

3.1. Availability of service nodes in service composition

In general, component service availability is defined as the percentage that a web service is available and functioning within its operational requirements [21]. The availability of each component service should be regulated based on the service contract or service level agreement (SLA).

A service node can be deemed as the connection point with a service proxy to bind concrete component services in a composite service. Due to the flexible binding nature of service-orientation, the availability of a service node should be aggregated from availability of all candidate concrete services since they can serve as replacements to one another to make the service node endurable, upon which measures of the availability of a service node can thus be developed as follows:
All possible service rankings for a service node should be considered, and all possible replacement sequences should also be identified. For example, if two concrete component services $S_a$ and $S_b$ are qualified to be bound to a service node and availability are $AS_a$ and $AS_b$, then its two possible replacement sequences are $\{S_a, S_b\}$ and $\{S_b, S_a\}$. Sequence $\{S_a, S_b\}$ means that if service $S_a$ fails or is missing, then $S_b$ can take over.

The aggregated availability can be calculated according to all replacement sequences. The availability of a service node in the case of replacement sequence $\{S_a, S_b\}$ is given by $AS_a + (1 - AS_a) \times AS_b = AS_a + AS_b - AS_a \times AS_b$. Notably, different replacement sequences may derive the same aggregated service availability. For instance, the service availability of sequence $\{S_a, S_b\}$ is the same as that of sequence $\{S_b, S_a\}$: $AS_a + AS_b = AS_a + AS_b - AS_a \times AS_b$.

Hence, the formulation of service node availability measurement can be defined as follows:

**Definition 1.** Assume $N$ candidate services (with the same service interface): $S_{a1}, S_{a2}, S_{a3}, \ldots, S_{an}$, are discovered for a service node $S_a$ in a composite service, and their corresponding availabilities are $AS_{a1}, AS_{a2}, AS_{a3}, \ldots, AS_{an}$, respectively. The availability of service node $S_{node}$ is defined as:

$$S_{node} = (AS_{a1} + AS_{a2} + AS_{a3} + \cdots + AS_{an})$$

$$- (AS_{a1} \times AS_{a2} + AS_{a1} \times AS_{a3} + \cdots + AS_{a1} \times AS_{an})$$

$$+ AS_{a2} \times AS_{a3} + AS_{a2} \times AS_{a4} + \cdots + AS_{an-1} \times AS_{an})$$

$$+ (AS_{a1} \times AS_{a2} \times AS_{a3} + AS_{a1} \times AS_{a2} \times AS_{a4} + \cdots$$

$$+ AS_{an-2} \times AS_{an-1} \times AS_{an}) - \cdots$$

$$+ (-1)^{N+1} \times AS_{a1} \times AS_{a2} \times AS_{a3} \times \cdots AS_{an}$$

### 3.2. Availability derived from flow patterns in service composition

To measure the availability of service compositions, the composition service flow is reduced by eliminating unnecessary information in order to reveal the relationship among component services. All composition service flows can be linked to three types of flow patterns, and can then be mapped into constructs in BPEL [2]. Measures, derived from the properties of the flow patterns, are used to define the availability of composite services. Figure 3 shows the flow patterns and their corresponding availability measures. Each component service is represented with a rounded rectangle and connected by directed arrows to indicate the execution sequence. These flow patterns and their availability measures are represented as follows:

- **Sequence:** this pattern can be mapped into the `<sequence>` construct in BPEL. The `<sequence>` construct can be adopted to define a collection of activities to be performed sequentially in lexical order [2]. In a reduced service flow that only
concentrates on relationships among component services, the sequence pattern is used to describe the case where several component services are invoked sequentially. Since all services participating in this flow pattern are required and should be invoked, the availability measurement should treat all component services as identically important factors. See “Sequence” part of Fig. 3 for illustration.

**Definition 2.** Assume component services $S_1, S_2, \ldots, S_n$ are in Sequence pattern, and their corresponding availabilities are $AS_1, AS_2, \ldots, AS_n$, respectively. The availability of this service flow pattern $A_{\text{sequence}}$ is defined as:

$$A_{\text{sequence}} = AS_1 \times AS_2 \times \cdots \times AS_n$$

- Parallel: this pattern can be mapped into the `<flow>` construct in BPEL. The primary semantic effect of grouping a set of activities in a flow is to enable concurrency [2]. Parallel pattern means that multiple component services are invoked in parallel. All services that participate in this pattern is also necessary. See “Parallel” part of Fig. 3 for illustration.

**Definition 3.** Assume component services $S_1, S_2, \ldots, S_n$ are in Parallel pattern, and their corresponding availabilities are $AS_1, AS_2, \ldots, AS_n$, respectively. The availability of this service flow pattern $A_{\text{parallel}}$ is defined as:

$$A_{\text{parallel}} = \frac{AS_1 + AS_2 + \cdots + AS_n}{n}$$

- Switch: this pattern can be mapped into the `<switch>` construct in BPEL. The `<switch>` can be applied to select one branch of activity from a set of choices [2]. In a reduced service flow, component services in each branch may be invoked with
The probability ranged from 0 to 1. The summation of probabilities of all branches should be 1. For simplicity, all branches should be assumed to have identical execution probabilities. See “Parallel” part of Fig. 3 for illustration.

**Definition 4.** Assume component services $S_1, S_2, \ldots, S_n$ are in different branches separately within *Parallel* pattern, and their corresponding availabilities are $AS_1, AS_2, \ldots, AS_n$, respectively. The availability of this service flow pattern $AS_{\text{switch}}$ is defined as:

$$AS_{\text{switch}} = \frac{AS_1 + AS_2 + \cdots + AS_n}{n}.$$ 

Although the *loop* pattern can be mapped into the `<while>` construct in BPEL, for the sake of performance, this pattern will not be considered in a composite service since it may cause a performance bottleneck in service execution, particularly when the termination condition is not well defined. Therefore, the proposed service flow patterns do not include *loop*.

Comparing with Menasce’s proposal [16] that presented five types of web service composition scenarios, our work focuses more on flow patterns that can be represented by the constructs in BPEL. Moreover, the aim of our work is to measure service availability, rather than cost or execution time, as in [16].

To measure the availability of a composite service, both service nodes and service flow patterns should all be considered. Refer to Sec. 5.5 for an illustrative example.

### 4. Dynamic Service Composition Pattern

This section first identifies essential issues of improving the availability of composite services. A new composite pattern, namely, Dynamic Service Composition (DSC) pattern, is then formulated to realize these requirements.

#### 4.1. Requirements for improving availability of service composition

Two ways can be applied to improve service availability: reduce the frequency of failures or reduce the time to recover from them [17]. According to Brewer’s viewpoint [5], it is generally easier to reduce the “time to recovery” in large and evolving systems, which is also a viewpoint shared in our work to improve service availability.

Accordingly, in order to improve availability of composite services, four important factors that affect “time to recovery” identified by Menasce and Almeida [17] are analyzed to find issues in the proposed service composition framework. Solving these issues is required to minimize “time to recovery” and then increase the availability of composite services.

- Detect the failure: the first step to recover from failures is detecting the failure. For service composition, the underlying infrastructure or middleware, such as servers, networks, disk arrays, database and message bus, are assumed to be
highly available since techniques exist for building highly available back-end systems [1]. However, service compositions must handle issues related to service dynamics, i.e. component service departure or malfunctioning. Hence “detecting failures of component services” is considered as an important issue to be addressed.

- Diagnose the cause of the failure: Detailed diagnosis of the cause is considered as the duty of the component service provider; and therefore, no issue is derived with respect to this factor.
- Determine possible solutions to the problem: the most effective method after detecting failures caused by component services is to replace the failed component service. The issue considered here is “discovering and selecting suitable component services”.
- Correct the problem: after discovering and selecting services, a composite service should bind suitable component services, and perform original incomplete tasks. The issue is thus how to “bind or substitute services” seamlessly.

To further explore these identified issues, five requirements are then derived as the criteria for designing the service composition framework, including notification mechanism, service registration mechanism, capability of replacing matching algorithms, binding management, and service proxy mechanism (see Fig. 4).

- Detecting Failures of Component Services: if any bound component service or candidate one is malfunctioning or missing, making it unregistered, then the
service requester (i.e. the composite service in this context) should be notified for further processing. Hence, the “notification mechanism” must be integrated into the service composition framework.

- Discovering/Selecting Component Services: tackling this issue involves satisfying two system requirements: a “service registration mechanism” and the “capability of replacing matching algorithms”. The registration mechanism provides point of contact for publishing and locating services, and is crucial for performing service discovery. The capability of replacing a matching algorithm allows the service matchmaker to adjust the matching mode according to the context or intentions of service requesters and service providers. For instance, some service requesters have to search for services according to the signature of service interface, while others want to search services based on properties or attributes of services.

- Binding/Substituting Component Services: a binding of a service into a composition can be defined as the reference used to choose the service to be invoked as part of the composition [22]. “Service bindings should be managed” in the various phases of the lifecycle of service composition to improve service selection or substitution. Additionally, to bind a component service to a composite service, a “service proxy” component may be prepared and designated as the bridge between the application and the service (see Sec. 2.1).

4.2. Patterns applied for improving availability of service composition

To realize these requirements, design patterns are adopted to design the service composition framework. Design patterns provide a set of reusable ideas that can speed up the design process since these ideas can be utilized as frameworks [23]. Monday [18] proposed fifteen web services patterns to satisfy the aims of learning about web services, adapting to web services, determining when changes occur, refining the structure of web services, and creating flexibility in web services. Although Monday’s patterns are currently the most acceptable ones in the SOA domain, there is still room for improvement, since the requirements of service availability in service composition are not properly addressed in his work.

Additionally, some recurring problems identified in the reign of service-orientation can still be solved using object-oriented design patterns (such as GoF patterns [11]) and can be used as supplements to web services patterns. Moreover, a composite design pattern can be used as a platform to gain benefits from both extant object-oriented design patterns and web services patterns. As Riehle claims in [24]: “a composite design pattern is a pattern that is best described as the composition of further patterns, the integration of which shows a synergy that makes the composition more than just the sum of its part”. Therefore, this work postulates that pattern composition could bring an elegant design philosophy, streamline various design considerations and provide a solution to enhance service availability, rather than applying individual patterns separately.
The following describes the making of a composite pattern by identifying and selecting patterns for each aforementioned requirement (see Fig. 4).

- **Notification Mechanism**: the *Observer* [11] pattern is selected to realize this requirement. The *Observer* pattern can notify and update relevant objects when the target object changes its state. Hence, composite services can be informed about the state of the related component services.

- **Service Registration Mechanism**: the *Service Directory* [18] pattern is chosen to realize this requirement. Service directories are data structures with deep associated structures to represent the data that they store about a service. This service directory is the primary point of contact for publishing, locating, registering for notifications, and maintaining directory entries. The Service Directory is thus a qualified service registry.

- **Capability of Replacing Matching Algorithms**: only one pattern, namely the *Strategy* [11] pattern, can implement this requirement. The *Strategy* pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Thus service matching algorithms can be replaced at either deployment or runtime.

- **Binding Management**: no ready pattern exists to manage service bindings. Therefore, the notion of service bindings will be integrated into the composite pattern.

- **Service Proxy Mechanism**: the *Proxy* [11] pattern is selected for addressing this requirement. The *Proxy* pattern provides a surrogate or placeholder for another object to control access to it, and can form a bridge between the service requester and the service.

![Fig. 5. Overview of pattern composition.](image)

Figure 5 illustrates the concept of pattern composition adopted herein. We argue that the *Service Directory* Pattern, which is a simple service registry, can be enhanced as a newly-defined *Service Matching* Pattern by combining *Observer* Pattern, *Strategy* Pattern, and *Proxy* Pattern (presented by Gamma *et al.* [11]), with the features of event notification, replacement of matching algorithms and
service proxy for binding services. The Service Matching Pattern is then further refined as Dynamic Service Composition (DSC) Pattern by incorporating semantics of relationships between the composite service and component services based on the concept of service binding. The DSC pattern is adopted to design a flexible architecture of service composition. The following sections introduce the two newly-defined patterns: Service Matching and Dynamic Service Composition.

### 4.3. Service matching pattern

![Structure of service matching pattern](image)

Figure 6 depicts the structure part of the Service Matching pattern. The following improvements are made to this new pattern:

1. *Publisher* is renamed as *ServiceProvider* and *Application* as *ServiceRequester*, which correspond to terms used in SOA, to avoid name conflicts. The name *Publisher* is confusing because it is equivalent to *Subject* in Observer pattern.

2. The *Directory* class becomes *ServiceMatchmaker* class, and adopts a *MatchingAlgorithm* class (a variety of *ConcreteStrategy* class in Strategy pattern) to complete the tasks of matching service requests and service descriptions, and return a service list.

3. The *ServiceMatchmaker* class inherits from the *SM_Subject* class (equals to the *Subject* class in Observer pattern), and the *ServiceRequester* class inherits from
SM_Observer class at the same time. The original Service Directory pattern implicitly includes the functionality of the Observer pattern; this feature is simply emphasized by explicitly integrating with the Observer pattern.

(4) ServiceInterface and ServiceProxy classes equivalent to Subject and Proxy classes in the Proxy pattern are added. Both ServiceImplementation and ServiceProxy are subclasses of ServiceInterface that specifies standard interfaces. ServiceRequester could consume services provided by ServiceImplementation via utilizing ServiceProxy that carries the same interface as ServiceImplementation.

4.4. Dynamic service composition pattern

As described in Sec. 2.2.2, dynamic composition means that services to be composed are determined at run-time [9]. The Service Matching pattern can be applied to improve the design for composing services dynamically, and to construct a new pattern, called Dynamic Service Composition (DSC) pattern. Figure 7 shows the structure of this pattern with the concept of “service binding” (shown as Rectangle in dashed line in Fig. 7):

(1) ServiceRequester is renamed as a CompositeService, indicating that the composite service acts as a service requester in SOA.

![Fig. 7. Structure of dynamic service composition pattern.](image-url)
A new \textit{ServiceBinding} class is added to maintain a memory for recording endpoints of candidate services and the current binding service for this instance of the service flow.

The DSC pattern can be applied to help tackle the requirements to increase availability of composite services based on the notions of dynamic service discovery and flexible service binding. However, in order to make the proposed service composition framework complete, it is necessary to address the activities involved in service composition, service discovery and service invocation, by modeling the complex interactions among the three participants in DSC pattern: ServiceMatchmaker, CompositeService, and ServiceImplementation, through an extension of Contract Net Protocol, called Extended Contract Net Protocol (ECNP).

5. ECNP: Extended Contract Net Protocol

The Contract Net Protocol (CNP) \cite{25} specifies problem-solving communication and control for nodes in a distributed problem solver. A contract net has agents that are totally cooperative, and selects contractors based on suitability, adjacency, processing capability and current agent load.

This work presents an Extended Contract Net Protocol (ECNP) to realize the interactions among the three participants in DSC pattern: ServiceMatchmaker, CompositeService, and ServiceImplementation, by Service Matchmaking Engine, Service Composition Engine and Service Provision Engine, respectively. There are four main scenarios, depicted using sequence diagram of UML 2.0 \cite{19}, in ECNP: making contracts, awarding tasks, normal decommiting and abnormal decommiting.

Moreover, this work adopts our previous work, Possibilistic-Petri-Nets-Based (PPN-Based) Service Matchmaker \cite{12}, to register, discover and select services. Standard service matchmaking is based on keyword searching. The PPN-based matchmaker is devised to address partial matching. It is applied to perform not only I/O matching for component services, but also conditional matching to locate services based on domain-dependent or domain-independent properties such as cost, response time, type and quality level of service.

In order to illustrate four scenarios of ECNP, an example of a traveling assistant service using our proposed methodology is given. For the sake of clarity, this example is simplified in order to explain how our methodology can be utilized. This example involves a \textit{Traveling Assistant Composite Service} consisting of three main component services, \textit{Hotel Reservation Service}, \textit{Route Planning Service}, and \textit{E-Mail Service}. Figure 8 displays the control flows between services in this composition. This composite service has the following expected behaviors: (1) it will find a hotel near a scenic spot input by the user, and reserve a room through the Hotel Reservation Service on behalf of the user; (2) it will generate a map that shows the route from the hotel to the spot; and (3) it will transmit an email message to the user listing all the service results, including the reservation information and
the generated map. To complete the above activities, another component service, called a glue service or wrapper service, has to be prepared to bridge the interface mismatch between E-Mail Service and the others, called Mail Content Generation Service. For simplicity, this component service can be neglected when considering service discovery or availability measurement.

The Service Matchmaking Engine is assumed to have multiple registered services (see Table 1) in this example. Every service provider must assert values of service availability it registers. The availability value of every component service is regulated using web service management mechanisms [20].

Note that the Service Matchmaking Engine contains only one E-Mail Service during the deployment time of this composite service. However, a new E-Mail service appears after this composite service is deployed.

Table 1. Component services, service providers, and service availabilities.

<table>
<thead>
<tr>
<th>Service Type</th>
<th>Service</th>
<th>Provider</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotel Reservation Service</td>
<td>hr₁</td>
<td>sp₁</td>
<td>0.99</td>
</tr>
<tr>
<td>Hotel Reservation Service</td>
<td>hr₂</td>
<td>sp₂</td>
<td>0.9</td>
</tr>
<tr>
<td>Hotel Reservation Service</td>
<td>hr₃</td>
<td>sp₃</td>
<td>0.999</td>
</tr>
<tr>
<td>Hotel Reservation Service</td>
<td>hr₄</td>
<td>sp₄</td>
<td>0.9</td>
</tr>
<tr>
<td>Route Planning Service</td>
<td>rp₁</td>
<td>sp₁</td>
<td>0.99</td>
</tr>
<tr>
<td>Route Planning Service</td>
<td>rp₂</td>
<td>sp₅</td>
<td>0.9</td>
</tr>
<tr>
<td>E-mail Service</td>
<td>es₁</td>
<td>sp₆</td>
<td>0.999</td>
</tr>
</tbody>
</table>
5.1. “Making contracts” scenario

In this scenario of ECNP, Service Composition Engine (SCE) makes contracts with component services by mediation of Service Matchmaking Engine (SME) to ensure the support for use of those component services hosted on Service Provision Engine (SPE) in the future. Steps in this scenario (see Fig. 9) are described as follows:

1. SPE advertises web services to SME.
2. SCE requests for services that match the proposed service interfaces which mainly comprise inputs and outputs.
3. SME replies with the matched services to SCE.
4. SCE requests SME for proposals of the matched web services.
5. SME replies bids to SCE if the matched web services agree to make contracts.
6. SCE makes the contracts and transmits the contracts to SME.
7. SCE advertises the composite service to SME.

Figure 9 includes two instances of the “loop” operator [19]. The outer loop describes SCE requests for candidate services for each activity that needs to invoke a specific type of service (that means it must be an implementation of a specific service interface). The inner loop describes SCE signing a contract with each candidate service, clearly defining factors such as the usage fee, usage duration and penalty for breaking the contract.

Consider the traveling assistant example, if the composite service provider completes service deployment in SCE, then SCE begins to request services that match the proposed I/O interfaces, namely the service interfaces of the Hotel Reservation Service, Route Planning Service and E-Mail Service. SCE then signs contracts with...
SME for legal use of these discovered services. It is assumed that SCE accepts all assertions, such as usage duration and cost. Finally, SCE advertises this composite service in SME.

5.2. “Awarding tasks” scenario

SCE that receives a request (by the service requester) starts to award tasks to component services. The key step in this scenario is that SCE requests SME for appropriate services again in a semantic manner before awarding tasks. Steps in this scenario (see Fig. 10) are described as follows:

1. SME forwards requests for the composite service to SCE.
2. SCE searches for the appropriate services by querying SME after receiving the service request.
3. SME returns the appropriate services to SCE.
4. SCE chooses binding services according to a service selection policy.
5. SCE awards the tasks to the chosen web services.
6. The matched web services hosted on SPE begin to perform the task and return the service results to SCE.

Figure 10 contains two instances of the “loop” operator, each adopted in a different way. The first loop describes how SCE decides the binding service for each “invoke” activity (or service node) in this instance of the service flow. The second loop describes the message exchanges between SCE and each designate binding service.

In the example, SCE will decompose the request into several sub-requests in order to locate a set of binding services according to the user’s request while SCE
receives a request by a user. This discovery activity ensures that semantics or properties of services satisfy user’s individual needs. SCE then starts to award tasks to binding services. An important recap can be noted:

- Different requests can induce different service combinations. If the first user issues a request of \{NearScenicSpots = spot_1, RoomType = single, MapDetailDegree = low\} (by filling out the request form [14]), then he obtains a service combination, \{hr_1, rp_2, es_1\} (determined by the PPN-based matchmaker). However, if the second user issues a request, \{NearScenicSpots = spot_2, RoomType = twin, MapDetailDegree = high\}, then he obtains a service combination, \{hr_2, rp_1, es_1\}. Each user can obtain and consume a special service combination that fits his respective requirements.

5.3. “Normal decommiting” scenario

![Diagram of ECNP: Normal decommiting](image)

If the provider of some component service wishes to repudiate the contract, then SCE can take the actions described in the scenario of ECNP (see Fig. 11). The scenario is described in detail as follows:

1. SPE requests SME for un-advertising service.
2. SME cancels the contract to SCE.
3. SCE cancels the contract and replies with agreements to SME if it agrees to the decommitment.
(4) If contractor services are still available, then the “Awarding Task” scenario is performed again (to bind a different component service).

(5) If no contractor services are available, then the “Making Contract” scenario can be performed again to ensure the availability of component services, and the Awarding Task scenario is performed later.

Figure 10 applies an interaction fragment called “InteractionUse” in UML. The InteractionUse is a shorthand for copying the contents of the referred Interaction containing the InteractionUse. This notation represents the reuse of defined sequence diagrams such as “making contracts” and “awarding tasks”.

5.4. “Abnormal decommiting” scenario

A component service may decommit the contracts without un-advertising itself. If SCE obtains error messages when awarding tasks, then it can recognize that contract decommitting occurs. SCE can then take actions based on this scenario (see Fig. 12). Steps in this scenario are described as follows:

(1) SCE awards tasks to the web service.
(2) SPE returns an error (wrapped as a SOAP fault message usually) to SCE if it fails to provide services.
(3) SCE requests SME for decommiting.
(4) SME returns the agreement to SCE if it agrees to cancel the contract.
(5) If contractor services are still available, then the “Awarding Task” scenario is performed again (as in the “Normal Decommiting” Scenario).
(6) If no contractor services are available, then the “Making Contract” scenario can be performed again, and the Awarding Task scenario is performed afterwards (as in the “Normal Decommiting” Scenario).

In the example, two typical cases may arise during service execution (corresponding to “Normal Decommiting” and “Abnormal Decommiting” scenario in ECNP):

- **Case 1:** If Route Planning Service $rp_1$ is unregistered (SCE will be notified by SME) or crashes (SCE will recognize it if SCE gets SOAP fault messages after retrying invocation for several times), then this component service can be substituted by $rp_2$, because $rp_2$ has become a contractor service early (in the phase of composite service deployment).

- **Case 2:** If E-Mail Service $es_1$ is unregistered or crashes, and there is no available contractor services, then SCE can attempt to re-sign a new service. This can be achieved in either of two ways: (1) if a new E-Mail service $es_2$ can be discovered, then SCE can sign a contract for using $es_2$, and continue the suspended tasks by awarding tasks to $es_2$; and (2) if no interface-compliant services can be found, then the system asks the user whether to perform partial service execution. In this case, the “mail” activity is skipped, and SCE simply generates information comprising the hotel reservation result and the map with planning route, and delivers it through web pages for browsing (not by email).

### 5.5. Evaluating service availability

The illustrative example can also be employed to compare the service availability of the proposed and conventional methods. With regard to conventional fixed service composition solutions (as in [27]), the availability of a composite service depends on the availability of fixedly binding component services. For instance, the availability $A'$ of a combination of $\{hr_1, rp_1, es_1\}$ (component services are pre-assigned, and are not substituted during the lifetime of the composite service) can be calculated as $A' = A_{hr_1} \times A_{rp_1} \times A_{es_1} = 0.99 \times 0.99 \times 0.999 = 0.9791199$, which is probably too low to be dependable.

However, service availability can be improved by applying the proposed approach (refer to Sec. 3 for the proposed availability measures):

- **Availability of Hotel Reservation Service Node:**
  
  $A_{hr} = A_{hr_1} + A_{hr_2} + A_{hr_3} + A_{hr_4} - (A_{hr_1} \times A_{hr_2} + A_{hr_1} \times A_{hr_3} + A_{hr_1} \times A_{hr_4} + A_{hr_2} \times A_{hr_3} + A_{hr_2} \times A_{hr_4} + A_{hr_3} \times A_{hr_4}) + (A_{hr_1} \times A_{hr_2} \times A_{hr_3} + A_{hr_1} \times A_{hr_2} + A_{hr_3} \times A_{hr_4}) - A_{hr_1} \times A_{hr_2} \times A_{hr_3} \times A_{hr_4} = 0.9999999$

- **Availability of Route Planning Service Node:**
  
  $A_{rp} = A_{rp_1} + A_{rp_2} - A_{hr_1} \times A_{hr_2} = 0.99 + 0.9 - 0.99 \times 0.9 = 0.999$

- **Availability of E-Mail Service Node:**
  
  $A_{es} = A_{es_1} = 0.999$
• Availability of Traveling Assistant Service can be calculated:
  \[ A = A_{hr} \times A_{rp} \times A_{es} = 0.9999999 \times 0.999 \times 0.999 = 0.9980009 \]

Consequently, although 99.80009% is not always acceptable for a highly-available system (they may need to meet the “five 9’s” criteria, i.e. up to 99.999%), the proposed approach still indicates a higher availability of web service compositions than conventional fixed service compositions.

In summary, the Extended Contract Net Protocol has the following advantages:

• Fault tolerance of Composite Service: due to notification by ECNP, the composite service can substitute binding services immediately when some of the binding services have been unable to perform tasks.
• Better availability: the failover mechanism of ECNP enables the composite service to be available even when some candidate services are missing or malfunctional.
• Increased usability: the composite service can bind appropriate services to fit the end user’s needs by interface and semantic service matching by PPN-based matchmaker. Each user can obtain his own service combination, even when consuming the same composite service as another user.

6. Related Work

In what follows we list some related works with respect to web service pattern and service availability.

Benatallan et al. [3] presented a suite of patterns for architecting and managing composite web services. They considered a life cycle of activities related to composite services, including wrapping native services, service advertisement/discovery, setting outsourcing agreements, assembling composite services, executing composite services, monitoring composite service executions and evolving services. Although patterns proposed in this scheme cover almost all research topics in service composition, they seem to be too high-level, and lack structures and behaviors, which are significant aspects of design patterns. In contrast, the proposed pattern-based approach solves a specific problem, namely service availability, and provides a detailed solution with the composite DSC pattern and the ECNP.

Buhler et al. [6] developed a composite design pattern for “stubless” web service invocation, based on the Bridge and Factory Method patterns. This composite pattern provides an agile software design for decoupling the invocation services that are found and bound at run-time. Our work is similar to theirs, but concentrates on dynamic service composition with discovery to improve availability, not just on the implementation-level binding problem.

Birman et al. [4] proposed extensions to the web services model that can support monitoring the health of the system, self-diagnosis of faults, self-repair of applications and event reporting. Their approach normally ensures “server-level” availability. Vilas et al. [26] proposed virtualization, in which new virtual web services are created instead of actual ones, and real web services are internally managed in a cluster in the back of the virtual service. Abraham et al. [1] introduced a central
hub that can manage invocation from several requesters and determine the best response. Paspallis et al. [21] presented an approach called smart-stubs and smart skeletons, which allows automatic rerouting of SOAP messages in order to ensure availability in the event of faults, and to block and adapt SOAP messages to enable dynamic upgrades of software. Dahlem et al.[8] presented a mediation model for improving the availability of composed services. The model masks failures in a service composition by transparently selecting and running an “alternative composition” at runtime.

Comparing with our proposed approach, most of these works do not take the availability of service composition, the effect of service discovery, as well as how and when to select failover services into account, which is the main focus of our work.

7. Conclusion

This work presents a discovery-based service composition framework to improve the static and dynamic integration of component services, with the following key features:

- devise the notion of service availability especially for composition;
- develop dynamic service composition (DSC) pattern to address the issues of service availability; and
- extend the Contract Net Protocol (ECNP) to coordinate service discovery, composition and invocation based on the composite pattern.

The main benefit of the proposed approach is to obtain better service availability by attaching multiple candidate services for future binding, and notifying users of changes to service status in real time (such as unregistered or interface changes). A composite service will still be available even when candidate services are either missing or malfunctional, since it can detach the failed component service, and re-bind a new one from candidate services list.

Our future research plan will further investigate the issues of Qualities of Web Services (QoWS) in service composition.

Acknowledgments

This research was partially sponsored by Ministry of Economic Affairs in Taiwan under the grant 96-EC-17-A-02-S1-029, and partially sponsored by National Science Council under the grant NSC 96-2422-H-008-001.

References


11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).


23. C. Peltz, Applying design issues and patterns in web services, 2005, 