Context Aware Differentiated Services Development
With Configurable Business Processes

Aries Tao Tao & Jian Yang
Department of Computing, Macquarie University, Sydney, NSW 2109, Australia
{tao, jian}@ics.mq.edu.au

Abstract

In order to make business service more flexible and reusable, it is desirable to provide users or applications the same service but with different service quality, different interaction paths, or with different outcomes. We call this design principle as Service Differentiation. This paper describes a fully functional mechanism where variability is externalized as usage context and context aware policies so that the core business process(es) need not to be altered for any anticipated changes. Service differentiation is realized by configured business processes and interfaces, and the dynamic 'binding' between user/application with a specific interface is determined by usage context during service invocation time.

1 Introduction

Supported by Web Service technology, Service Oriented Computing (SOC) allows resources on a network to be made available as services that can be accessed without the knowledge of their underlying platform implementation[4]. A service is a business concept that should be specified with an application or the user of the service in mind[13]. In order to understand how services are designed and developed, we need to take a close look into the relationships between service, service interface, and business process:

- A service is an abstract business concept that indicates the functionalities of a business.
- Service interface is a specification for user (client program) to interact with the service. It is supported by underlying business process(es), and consisted of operations which are derived from the corresponding business process(es).
- The business process(es) implements the functionalities of a service. The business process consists of activities that perform service functions. There are two types of activities in business process:
  - User invisible activities that are hidden from the service interface in order to hide internal business logics and certain implementation details from the users.
  - User visible activities that are presented in service interface and used for user (client application) interactions.

In e-Business world, the main aim of a business is to provide better, and more flexible services to its customers. Quite often we find that different users (client programs) may need to interact with the same service differently, or may expect different outcomes from the same service. These differences can be either hidden from the user, for example, the outcomes are dependant on the user profile; or presented to the user so that different interfaces for the same service should be provided. Therefore from business process design point of view, two types of supports are necessary:

- operational support - business process variability needs to be provided for different users and usages. Take Online Pharmacy Service (OPS) as an example, OPS provides different discount rates depending on customers’ profiles:
  - offers 10% discount to loyal customers.
  - offers no discount for normal customers.
- interface support - users with different behaviour semantics need to access the same service through different interfaces. Continuing with the same OPS example, different approval procedures are required for different types of medicines customers intend to purchase:
  - requires doctor’s prescription from people who purchase prescribed medicine.
Figure 1. Current & Proposed Service Design Approach
does not require any approval from people who purchase un-prescribed medicine such as Panadol.

Given the fact that different users or applications often have different (but overlapping) requirements, and may require the same service at different quality levels, with different interaction patterns, and may acquire different outcomes, it is a good idea to provide a single service with variations rather than several unrelated services. We refer to this design principle as Service Differentiation. The way we want to realize service differentiation, i.e., single service with different interaction patterns for different users/applications presents an interesting analogy to object orientation in terms of overloading and polymorphism. The determining rules for ‘binding’ a specific interface and service outcome with an application or an interaction pattern are governed by business policies, i.e., for ‘loyal customer’ 10% discount will be offered. Here in this paper, we refer user profile, location, interaction pattern, purpose of interaction as usage context, on which business policies are applied and linked to the different service invocation.

Now let us have a close look of current service design approaches in relation to internal business process and service interface design and analyse their deficiency in supporting service differentiation. As illustrated in Figure 1, business policies or rules are hard coded in the business process as different activity execution conditions. As a result, service is supported by a fixed business process which provides the same functionalities to all the users from the same service interface. On the other hand, differentiated service requires (1) business policies are separated from business processes; (2) usage contexts are separately maintained and explicitly associated with interfaces so that different users/applications can be ‘treated’ differently by the same service in terms of the different interfaces presented and different outcomes generated.

The basic design philosophy of ours, and one that distinguishes us from others, is that one core business process is required for all users in all circumstances with different context configured business processes generated for different users and different interaction patterns. This requires a new service design approach that separates the generic business activities that are applicable to all the circumstances from those only applicable to particular group of people or under specific conditions. Thus we propose a new service design approach that supports single service with multiple business processes which supports service differentiation. Our design is based on the following ideas:

- Use Abstract Business Process (ABP) to support the functionality that are generic and required by all the users.
- Use Policy to describe how the service should be differentiated for different users.

- Multiple Context Configured Business Processes (CCBPs) can be derived based on ABPs and Policies to support different users/applications.
- Context-aware service interfaces are derived from CCBPs.

Ultimately, our priorities and the focus of the paper are:

- to separately maintain usage context and policies,
- to dynamically generate business processes if necessary,
- to present different service interfaces to the users/applications based on different usage contexts, and
- to have minimal impact on business service when the policies undergo modifications.

This paper is organized as follows: Section 2 discusses the related work. The case study is introduced in section 3. Service design and related algorithms are specified in Section 4 and 5. We finally concludes our work in section 6.

2 Related Work

In this section we will discuss related work from five aspects. Firstly we review the research work been done on differentiated service. Secondly we shall take a general look into the research work in context. Thirdly we review the current state of arts of research in usage context in SOC area. Fourthly we review the current service describing techniques, arguing that they can not effectively describe differentiated services. Finally we analyse the related Web Service[25] Standards which can be used for our work.

The idea of service differentiation (DiffServ) [22] was firstly proposed in the area of to manage traffic streams in networking applications. For example, some traffic is treated better than the rest (faster handling, more average bandwidth, and lower average loss rate). Richard Veryard in [15] argued that the differentiated services should be used as a design pattern in SOC area. However, no methodology has been provided to realize service differentiation.

**Context** is widely applied in computing area: Want et al [17] has introduced an Active Badge system forwards phone calls according to the user context such as location. Abowd et al [1] applied context in implementing a tourist guide system called Cyberguide. Cheverst et al [5] also made use of context in their tour guide application called "the GUIDE project". By using an object-oriented approach, the ContextToolkit[8] provided a framework and a number of
reusable components to support rapid prototyping of sensor-based context-aware applications.

The importance of applying contexts to SOC has been addressed in the following literatures: Baldauf, Dustdar and Rosenberg discussed the close relationships between context and services in their survey[3]. Maar et al[11] clearly stated the needs of contexts in order to derive personalized service. They also classified the context into several catalogues in terms of their effect to the service. Hong, Chiu and Shen[10] proposed a model to analyse the effect of contexts on service. Gu, Pung and Zhang[9] proposed a formal context model based on ontology using Web Ontology Language to address issues including semantic representation, context reasoning, context classification and dependency. However, no methodology has been provided to specify how to configure business process based on usage contexts. Our work complements their work by providing a working mechanism to derive context configured business processes for a business service, based on which differentiated service can be provided through context aware service interfaces.

Work has been done in the area of specifying public interface for business processes. Chiu et al [7] presented a meta-model for Service Interface as Workflow Views, which provided a novel approach to derive Service Interface (as Workflow View) from a Service (as workflow). By abstracting Service Interface as certain subset of Service, it allows internal information to be hidden from external users. However, the work only focused on abstract a single service interface which did not take usage context into consideration. To support different user groups, Zhao, Liu and Yang[19] proposed the concept of relative workflow view by explicitly extracting visibility constraints (Invisible, Traceable, Contactable) on activities of workflow. Based on different visibility constraint for different users over the same workflow, multiple relative workflow views could be derived for different users with different relationship with the service. However, the service provider has to manually set constraint every time for each new service user, the system does not scale well. Using a completely different strategy, our work allows service interface generated from configured business processes, which are derived automatically based on usage contexts. As a result, the maintenance effort is shifted from managing business processes to managing usage contexts. Now let us have a close look at current relevant web service standards. WSCI[26] allows the service to be described as a sequence of Web Service calls binding to WSDL[27]. In advance, BPEL[20] allows the service interface to be described as an abstract business process, which is a subset of BPEL process. BPEL allows several abstract business processes to be derived for one service. However, BPEL specification does not provide any mechanism or standard to generate multiple service interfaces, which is often required by business as illustrated in our example. Several Semantic Web Service Description standards such as OWL-S[21], WSMO[29] has been proposed. Comparing with WSDL and BPEL, Semantic Web Service provides better support in common understanding of context semantics and reasoning on complex relations between contextual concepts[12]. However, service is designed to have only one Service Process Model. Such design limits the service flexibility to support different user interactions. Our work can be used to extend current Semantic Web Services by allowing multiple service Process Model to be derived for a single service, and hence allow users to access the same service in different ways.

In order to relate the policy with Web Services, Web Services Policy Framework WS-Policy[33] was proposed by the World Wide Web Consortium (W3C). It is a general framework for specifying various Web service properties in a way that complements WSDL and BPEL. On the other hand, Web Services Policy Language WSPL[30] was proposed by the Organization for the Advancement of Structured Information Standards (OASIS). It is suitable for specifying a wide rage of policies, e.g., acceptable and supported encryption algorithms or privacy guarantees. Both WS-Policy and WSPL focus on supporting security domain only. Vladimir et al[31] extends the WS-Policy for monitoring and adaptation of Web services and their composition. On the other hand, WS-Context[32] and WSDL-S[28] provide a base to allow contexts to be described under Web Services execution environment. However, their work also does not support service differentiation. Our work complements their work by providing a service design which allows context based service differentiation.

3 A Motivating Example

In order to understand the issues involved in differentiated service development, we conduct a case study in Online Pharmacy Service (OPS). Due to space limit, we only take a look of the Checkout process that is depicted in Figure 1. The Checkout business process consists of the following activities, some of which may not available for all users:

- Login - receives the user name and password in order to identify the user.
- Display Goods Selection - displays a list of goods selected by the user.
- Display Total Price - displays the total prices the customer needs to pay for the goods.
- Display discounted price depending on the customer's profile.
– For normal customer - there is no discount available.
– For loyal customer - 10% discount is offered.
– For VIP customer - 15% discount is offered.

• Approval - extra approval may be required depending on the type of purchased medicine:
  – For prescribed medicine - prescription from a doctor is required.
  – For un-prescribed medicine - no approval is required.

• Receive Payment - receives payment from user and hence return invoice.

• Delay Payment is only available for customer with usage context as "VIP customer".

• Product Delivery - contacts delivery company to deliver the product to customers.

As we can see, the Checkout process actually performs different tasks for different types of customers or for different types of purchases. Two types of usage contexts can be identified: customer profile, and type of medicine purchased.

Our goal is to develop a differentiated Checkout service that provides context aware interfaces. These interfaces provide explicit service usage information to their users in terms of conditions, restrictions, benefits, etc. Therefore users can choose different service interfaces based on their profiles or usage contexts. With the different discount rates information associated with the service interface description, customers can be encouraged to join the loyal or VIP membership instead of the normal membership. In the following sections we will show how this goal can be achieved based on the above example.

4 Differentiated Service Specifications

Instead of supporting a service with one monolithic business process and single service interface, our design philosophy is to separate the generic functionalities that are available for all users from the specific functionalities with certain context dependant requirements. The main issue is to find a way of identifying and mapping between process logic, usage context, inputs and outputs. As illustrated in Figure 2, the design consists of four components:

• Abstract Business Process (ABP) that implements the generic functionalities that are provided to all the users.

• Business policies that provides the mapping between usage contexts and functionalities. In the context of service differentiation, business policies can support different users with different functionalities or different outcomes as follows:
  – Policies can provide mappings between usage profile that are derived from interactions with specific functions for producing different outcomes. For example, in Checkout process, the "customer profile" as a usage context can be retrieved from the "login" activity. By applying "discount policy", "VIP customer" will get further 15% discount by invoking the "applying 15% discount" function.
  – Depends on the usage context values, the policy decides to plug different business processes into the ABP, and generate Context Configured Business Processes (CCBPs) which perform different business tasks. For example, in case the user wants to purchase prescribed medicine, then additional approval process is required, i.e., requiring user to present prescription from its doctor.

• Context Configured Business Process that supports users with different functionalities, and thus realizes differentiated services based on usage contexts. For example, in Checkout process, the loyal customer and normal customer will be supported by different Context Configured Business Processes.

• Service Interfaces that act as interaction protocol. Each Service Interface is supported by one Context Configured Business Process. It allows users to interact with the service differently based on their usage contexts.

4.1 Abstract Business Process

Definition 1 \( \text{ABP} = (\mathcal{A}, \mathcal{E}) \). Abstract Business Process is labeled as ABP where \( \mathcal{A} \) is a set of activities and \( \mathcal{E} \subseteq \mathcal{A} \times \mathcal{A} \) is a set of directed sequences that connects \( \mathcal{A} \). Each activity \( a \) has an attribute "ref".

\[ \forall \text{ activity } a \in \mathcal{A}, a \text{ could either be one generic activity or policy specific activity:} \]

• If \( a \) only needs to perform a task for all users regardless contexts (e.g. the login activity in Checkout process), then we regard \( a \) as generic activity and \( a.\text{ref} \) refers to a task only.

• If \( a \) needs to perform different tasks or even execute different processes depending on contexts (e.g. the ap-
Figure 2. Service Interface Derivation Algorithm
activity in Checkout process is required to execute different approving processes, then we regard \( a \) as policy specific activity. In this case, \( a_{\text{ref}} \) refers to a template in the policy (will be discussed in section 4.2) which defines how contexts aware functions should be performed.

Based on the definition provided above, we can model the Abstract Business Process of the Checkout process as depicted on top-left side of Figure 2:

**Example 1** \( ABP_{\text{checkout}} = (A_{\text{checkout}}, E_{\text{checkout}}) \)

\[
A_{\text{checkout}} = \left\{ \text{login, display-goods, display-total, discount, approval, receive-payment, delay-payment, delivery} \right\}
\]

where \( a_{\text{login}}, a_{\text{ref}} = t_{\text{login}}, a_{\text{display-goods}}, a_{\text{ref}} = t_{\text{display-goods}}, a_{\text{discount}}, a_{\text{ref}} = t_{\text{discount}}, a_{\text{approval}}, a_{\text{ref}} = t_{\text{approval}}, a_{\text{receive-payment}}, a_{\text{ref}} = t_{\text{receive-payment}}, a_{\text{delay-payment}}, a_{\text{ref}} = t_{\text{delay-payment}}, a_{\text{delivery}}, a_{\text{ref}} = t_{\text{delivery}} \)

Note that \( a_{\text{discount}}, a_{\text{approval}} \) and \( a_{\text{delay-payment}} \) are policy specific activities that refers to the templates in \( P_{\text{ Checkout}} \) which will be discussed in section 4.2. The rest activities refer to tasks only.

\[
E_{\text{checkout}} = \left\{ (a_{\text{login}}, a_{\text{display-goods}}), (a_{\text{display-goods}}, a_{\text{discount}}), (a_{\text{discount}}, a_{\text{approval}}), (a_{\text{approval}}, a_{\text{receive-payment}}), (a_{\text{approval}}, a_{\text{delay-payment}}), (a_{\text{receive-payment}}, a_{\text{delivery}}), (a_{\text{delay-payment}}, a_{\text{delivery}}) \right\}
\]

4.2 Context Aware Policy

**Definition 2** Policy \( P \) is a set of templates. Each template \( \text{temp} \) is referred by a policy specific activity in the Abstract Business Process. As discussed before, policy in the context of service differentiation is responsible for providing different functionalities according to different usage contexts.

**Definition 3** \( \text{temp} = (\text{COND}, \text{ABP}) \), \( \text{temp} \) provides mappings between context conditions and Abstract Business Processes. Since an ABP only can be associated with one context condition, different functions will be provided by ABPs based on different context conditions \( \text{cond} \in \text{COND} \).

**Definition 4** \( \text{cond} \rightarrow \text{context comparison}_{\text{op}} \text{ simple var} | \text{cond logical}_{\text{op}} \text{ cond} \)

- \( \text{The cond returns a boolean value by comparing the context against a simple variable (e.g. string, int, float) via comparison}_{\text{op}} \) (e.g. \(<, >, ==, \leq, \geq \)).

- Complex conditions are formed by linking conditions with logical \( \text{op} \) (e.g. \( \land, \lor \)).

Take the template \( \text{temp}_{\text{discount}} \) as an example, (as showed in Policy in Figure 2), it defines different discount rates provided to different customers. The discount rates is determined by the value of the usage context \( c_{\text{loyalty}} \), and the value of \( c_{\text{loyalty}} \) can be \( d_{\text{normal}}, d_{\text{loyal}}, d_{\text{vip}} \) which represents three loyalty levels: normal customer, loyal customer and VIP customer. As a result, the template \( \text{temp}_{\text{discount}} \) defines three relations that: (1) \( a_{\text{discount}} \) connects the business process "10% discount" for customers’ with \( d_{\text{loyal}} \); (2) \( a_{\text{discount}} \) connects the business process "15% discount" for customers with \( d_{\text{vip}} \); and (3) \( a_{\text{discount}} \) connects the business process "nil" which provides no discount for customers’ with \( d_{\text{normal}} \).

As illustrated in Figure 2, the policy of OPS contains three templates which correspond to the three policy specific activities in the main Abstract Business Process - \( a_{\text{discount}}, a_{\text{approval}}, \) and \( a_{\text{delay-payment}} \).

**Example 2** \( C_{\text{Checkout}} = \{c_{\text{loyalty}}, c_{\text{medicine}}\} \) There are two usage contexts that cause the Checkout process to perform different tasks:

- \( c_{\text{loyalty}} \) represents the customer loyalty context. The value of \( c_{\text{loyalty}} \) has been explained in the above.

- \( c_{\text{medicine}} \) represents the type of the purchased medicine. The value of \( c_{\text{medicine}} \) can be:
  - \( d_{\text{daily}} \) to represent the purchased medicine is for daily use that does not require doctor’s prescription.
  - \( d_{\text{prescribed}} \) to represent the purchased medicine which needs to be prescribed by a doctor.

**Example 3** \( \text{P}_{\text{Checkout}} = \{\text{temp}_{\text{discount}}, \text{temp}_{\text{approval}}, \text{temp}_{\text{delay-payment}}\} \), The policy \( P \) for the Checkout process consists of three templates. As defined before, templates provide mappings between conditions and abstract business processes. However in this paper for simple illustration purpose, we only provide mapping between conditions with single tasks instead of abstract business processes. The detailed example of context policies is depicted in Figure 2:

- As showed at the top of Policy in Figure 2, \( \text{temp}_{\text{discount}} \) represents the policy to perform different discount rates for customers with different loyalty level in the activity \( a_{\text{discount}} \). It consists of the following three tuples:
  - \( (c_{\text{loyalty}} == \text{d}_{\text{normal}}, \text{t}_{\text{nil}}) \) - provides no discount to normal customer. Notice that \( \text{t}_{\text{nil}} \) means no task is performed in the business process.
- \( \text{cloyalty} = d_{\text{loyal}} \times t_{10\% - \text{discount}} \) - provides 10% discount to loyal customer.
- \( \text{cloyalty} = d_{\text{vip}} \times t_{15\% - \text{discount}} \) - provides 15% discount for VIP customer.

• As showed at the middle of Policy in Figure 2, \( t_{\text{approval}} \) represents the policy to perform different approving procedures for customers who purchasing different categories of medicine in the activity \( a_{\text{approval}} \). It consists of the following two tuples:

- \( \text{cmedicine} = d_{\text{daily, not}} \) - for customers who purchase daily medicine only, no approval is required.
- \( \text{cmedicine} = d_{\text{prescribed, confirm - doctor}} \) - for customers who purchase prescribed medicine, confirmation from doctor is required.

• As showed at the bottom of Policy in Figure 2, \( t_{\text{delay - payment}} \) represents the policy to perform different payment procedures for customers with different loyalty level in the activity \( a_{\text{delay - payment}} \). It consists of the following two tuples:

- \( \text{cloyalty} = d_{\text{normal, loyal}} \lor \text{cloyalty} = d_{\text{loyal, not available}} \) - for normal customer or loyal customer, delay payment is not available.
- \( \text{cloyalty} = d_{\text{vip, delay - payment}} \) - for VIP customer, delay payment is available.

4.3 Context Configured Business Process

After the introduction of Abstract Business Process, Context, and Policy in the previous sections, we can now provide a clear picture of how a 'concrete' business process - context configured business process, is generated. As discussed before, an abstract business process is a process that has policy specific activities, which refer to policy templates. A template consists of a set of mappings between context conditions and business processes (or tasks in simple situation). The number of tuples (mappings) in a template determines the edges that a policy specific activity can be associated with when the activity is replaced by the business processes (or tasks) that those template conditions point to. The relationships between Abstract Business Process, Context, Policy, template, and Context Configured Business Process is summarised in Figure 3.

**Definition 5**  
\[
\text{CCBP} = (T, S).
\]

- \( T \) is the set of tasks that are provided to a group of users.
- \( S \) is the set of tuples \((s, \text{cond})\)

\[ s \in T \times T \text{ is a directed edge that connects two tasks in } T. \]

\( \text{cond} \) is the usage context value condition that attached to the edge \( s \), which indicates that the \( \text{cond} \) must be satisfied in order to successfully access the edge \( s \).

This design has two characteristics. Firstly, a business service is supported by multiple CCBPs that are generated based on the context conditions. For Checkout service, since the context \( \text{cloyalty} \) can have three values, and \( \text{cmedicine} \) can have two different values, there are six possible context value combinations as conditions for policy templates. Therefore six possible CCBPs can be generated accordingly. For space limitation, we only demonstrate two CCBPs in Figure 2: the \( \text{CCBP}_1 \) supports normal customers who purchase daily medicines; the \( \text{CCBP}_6 \) supports VIP customers who purchase prescribed medicines.

Secondly, as depicted in Figure 2, \( \text{cond} \) is attached on the edges of these generated CCBPs, which indicates the 'path' s the procedure of configuration, from ABPs to CCBPs, has followed. The configuration from ABP to CCBP is a recursive procedure by unfolding policy specific activities in ABPs until all of them are replaced by concrete tasks and all the edges between tasks are labelled with context conditions.

4.4 Service Interface

A Context Aware Service Interface corresponds to a CCBP. It allows a group of users who share the common context values to interact with the service in a particular way, and therefore enjoy the differentiated service provided to them.

**Definition 6**  
\( SI = (\text{userCond}, VT, VS) \).

- \( \text{userCond} \) is a set of \( \text{cond} \) that extracts from the edges of the corresponding CCBP. It represents the contexts the user must hold in order to access this specific interface.
- \( VT \subseteq T \). For security and privacy concerns, the service interface hides invisible tasks from users. In other word, the service interface only inherits visible tasks from its corresponding CCBP.
- \( VS \subseteq VT \times VT \) is the set of directed edges that connect two visible tasks in \( VT \).

As showed in Figure 2, we can see that multiple service interfaces can be automatically derived from CCBPs. Every service interface supports a group of users with common usage contexts. This service interface design benefits the users from the following:
Figure 3. Proposed Service Design Approach
It allows users to check if their contexts satisfy the condition as an extra step in process matching before the users can establish interaction with the service.

During service invocation, it provides guideline for proper user interactions, and prevents exceptions caused by unexpected context conditions encountered.

5 Differentiated Service Design

The steps in differentiated service design are as follows:

- **Usage Context Identification**: usage contexts need to be identified, e.g., location, user profile, purpose, etc. The possible values for the identified contexts also need to be identified so that conditions can be established and mappings between the conditions and abstract business processes for policy templates can be developed.

- **Context Configured Business Process Generation**: generates CCBP based on ABPs and Policies by recursively replacing policy specific activities with the ABPs that the policy templates refer to till all the activities refers to the concrete tasks and all the edges are labelled by conditions.

- **Context Aware Service Interface Derivation**: derives interfaces from CCBPs.

5.1 Usage Context Identification

This is the process for storing contexts, conditions, templates, and policies that consists of mappings between context conditions and ABPs. Business policies can change frequently, therefore modifications are expected and should be supported. We plan to implement contexts, conditions, and policies on a database so that they can be queried, searched, and modified.

5.2 Context Configured Business Process (CCBP) Determination

CCBPs are generated based on ABPs, Policy templates identified in the last phase. The algorithm for generating CCBPs consists of two steps: Find Available Graph and Find Executable Business Process:

- **Find Available Graph** traverses the ABP, retrieve different business processes from templates based on context values, plug those business processes into the corresponding policy specific activities, and finally derives available graph as ag. ag represents the functionality that is available to users groups. ag is in the

```algebra
Algorithm 1 FindAvailableGraph
Require: INPUT: ABP, cc (context conditions), t_{i-1} \in T, a_i \in A, and ag
1: if a_i = nil then
2: return
3: end if
4: if a_i is a task then
5: bp_i = a_i.ref
6: else
7: bp_i = DeriveServiceInterface(a_i, cc)
8: end if
9: add bp_i into ag.T
10: add bp_i related edges and context conditions into ag.S
11: next_activity_set = get_next_activity_set (a_i, ABP)
12: if bp_i = t_{not_available} then
13: return
14: else if bp_i = t_{nil} then
15: for all a_{i+1} in next_activity_set do
16: FindAvailableGraph( ABP, context_conditions, bp_{i-1}, a_{i+1}, ag)
17: end for
18: else
19: for all a_{i+1} in next_activity_set do
20: FindAvailableGraph(ABP, context_conditions, bp_i, a_{i+1}, ag)
21: end for
22: end if
```

```algebra
Algorithm 2 FindCCBP
Require: INPUT: CCBP, ag, t_i, t_{i+1} \in T
1: if t_i = nil then
2: return
3: else if t_i = t_{not_available} then
4: return
5: else
6: add t_i and it’s related transitions into CCBP.T
7: end if
8: prev_task_set = get_prev_task_set (t_i, ag.S)
9: for all t_{i-1} in prev_task_set do
10: FindCCBP (CCBP, ag, t_i, t_{i-1})
11: end for
```

Authorized licensed use limited to: MACQUARIE UNIV. Downloaded on February 5, 2009 at 22:32 from IEEE Xplore. Restrictions apply.
same structure as CCBP which contains two components: (1) $T$ - the tasks that can be accessed by the specific contexts; (2) $S$ contains the corresponding edges to link the tasks.

- **Find CCBP.** Some branches of $ag$ may contain the task $t_{not\_available}$, those branches are consider as broken branches because they do not lead to a proper end of business process. In order to avoid exceptions, the function $FindCCBP$ delete all the broken branches from $available\_graph$, then derives CCBPs.

As showed in algorithm 1, in order to derive available graph from ABP, the function $FindAvailableGraph$ starts from the root activity with $available\_graph$ (ag) as an empty graph, and traverse each activity of the ABP as follows:

- If the activity is a generic activity, then add its referred task and related transitions into the available graph; else if the activity is policy specific activity, derive the corresponding business process from Policy as $bp_i$, and plug $bp_i$ into the $ag$ (Line 4-10).
- This function will recursively call itself to traverse every activity unless:
  - it reaches the end of the ABP (Line 1-2).
  - it finds a broken branch caused by $t_{not\_available}$, then stop further traversing on the branch (Line 12-13).

As showed in algorithm 2, in order to derive CCBP from available graph, the function $FindCCBP$ works as follows:

- It starts from the last task in available graph (ag), and reversely traverses the available_graph (Line 8-10) until the root is reached (Line 1-2).
- The broken branch which includes $t_{not\_available}$ would be ignored (Line 3-6).

### 5.3 Service Interface Derivation

Service interfaces are derived from CCBP by hiding invisible tasks from users. Each Service Interface (si) enables a group of users who share common usage contexts to interact with the service in certain ways. As showed in algorithm 3, $FindServiceInterface$ works as follows:

- It recursively traverse CCBP from the root task (Line 12-20) until reach the end of the CCBP (Line 1-2).
- Extracts the context conditions (Line 8).
- The invisible tasks and related transition are hidden from users (Line 12-15).

#### Algorithm 3 FindServiceInterface

**Require:** INPUT CCBP, si(service interface), $t_{i-1} \in T$, $t_i \in A$

1: if $t_i = nil$ then
2:     return
3: else if $t_i$.visibility = visible then
4:     add $t_i$ into si.VT
5:     let $t_i$ related transitions to be trs
6:     for all $tr \in trs$ do
7:         add $tr.s$ into si.VS
8:         add $tr.cond$ into si.userCond
9:     end for
10: end if
11: next_task_set = get_next_task_set ($t_i$, CCBP))
12: if $t_i$.visibility = invisible then
13:     for all $t_{i+1}$ in next_task_set do
14:         $FindServiceInterface$ (CCBP, si, $t_{i-1}$, $t_{i+1}$)
15:     end for
16: else
17:     for all $t_{i+1}$ in next_activity_set do
18:         $FindServiceInterface$(CCBP, si, $t_i$, $t_{i+1}$)
19:     end for
20: end if

### 6 Conclusion

Service users or applications often have different interaction or service outcome requirement from a business service. In this paper, we proposed and argued the need for service differentiation. We believe multiple interfaces supported by different underlying business processes should be provided for the same service. The design is developed around the concept of usage context, based on which concrete business processes and interfaces are generated and labeled with the relevant context conditions.

Future work will be carried out in the following areas:

- **service description:** we are currently investigating a way to incorporate our approach into OWL-S so that differentiated service can be described properly.
- **service interface binding:** we also focus on the way to bind an interface with application context during service invocation time.

### References


[23] Simple Object Access Protocol (SOAP) 1.1. www.w3.org/TR/soap/


[27] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsd1


