Event Exposure for Web Services: A Grey-box Approach to Compose and Evolve Web Services[⋆]

Chunyang Ye and Hans-Arno Jacobsen

University of Toronto

Abstract. The service-oriented architecture (SOA) is an emerging software engineering paradigm for developing distributed enterprise applications. In this paradigm, Web services are encapsulated and published as black-box components accessible to service consumers following the principles of component-based design. This however restricts the flexibility and adaptability of Web services to react to changing requirements, which are commonplace today, especially in the emerging smart Internet and smart interactions domain. In this chapter, we propose a grey-box approach to compose and evolve Web services to increase their flexibility and adaptability. By exposing the services' internal state changes at runtime as events, our approach allows services involved in service compositions to share and consume events from partner services, and make use of these events to evolve and adapt their behavior. This approach is illustrated in two case studies.

1 Introduction

The service-oriented architecture (SOA) is a widely adopted software engineering paradigm to manage the complexity of software development for distributed enterprise applications. In this paradigm, service providers develop reusable software components, publish them as Web services, and register them in service registries. By composing selected services from service registries, service consumers can quickly develop collaborative applications across distributed, heterogeneous and autonomous organizations.

Traditional service composition approaches inherit the principles of component-based design and treat Web services as black-box components. These principles hide the implementation details of services and encapsulate their functionality in service interfaces (e.g., WSDL and WSCI [28] etc.). In this way, the complexity of maintaining and interoperating services in service-oriented applications is reduced, and the business concerns of service providers are protected (i.e., service implementations are not revealed.)

However, services are different from components, in the sense that a service usually describes a partial behavior whereas a component describes a whole behavior [5]. With respect to an SOA environment, services are usually developed

⋆ This work is published as a chapter in the book "The Smart Internet", Springer LNCS, Volume 6400.

independently by different service providers to realize some partial enterprise application functionality. Also, often participating services are executed and maintained in different heterogeneous and autonomous organizations. Therefore, services in a service composition usually have little knowledge about their partner services except a restricted view via the partners' statically exposed service interfaces. All runtime interactions are based on these statically exposed and restricted interfaces, which may not be able to handle unexpected scenarios at runtime. As a result, these black-box solutions are inadequate for SOA applications to flexibly react and adapt to changing application domain requirements. Applications in emerging domains such as the newly developing smart Internet, smart interactions, and Web 2.0 spaces are especially prone to suffer from lack of flexibility and adaptability due to the ever changing standards and new specifications that drive these rapidly developing domains.

For example, in a healthcare application network composed of black-box services, the interactions among hospital services and pharmacy services strictly adhere to their statically exposed service interfaces (e.g., the hospital services send the electronic prescriptions to the pharmacy services). During their interactions, the hospital services have no idea of the internal execution state of the pharmacy services (e.g., what kinds of medication are in short supply). Similarly, the pharmacy services know nothing about the internal runtime state of the hospital services (e.g., the number of patients registered by the hospital services and what the kinds of medical conditions diagnosed are). The hospital services and the pharmacy services may collaborate to work well under normal operational conditions; that is without the knowledge of their partners' runtime execution state. However, such an application lacks the flexibility to quickly adapt to changing requirements. For instance, when a flu outbreaks, the number of patients would increase dramatically, but the pharmacy services may not expect such a scenario at design time and are unable to detect and handle such a scenario at runtime. As a result, the pharmacy services are unable to react more smartly by timely and pro-actively provisioning more drugs via their own suppliers.

To enable applications to react to changing requirements, service providers may need to evolve the services in a service composition. For instance, both hospital services and pharmacy services may be re-designed and re-encapsulated to extend their original interfaces with additional query functionalities. Then new interactions are developed based on these extended interfaces to cover the changing requirements (e.g., the pharmacy services can periodically retrieve the statistics on patients and update their inventory levels accordingly). However, such a solution may be less efficient to keep up with the frequently changing requirements because service providers may need to sit down and negotiate their new service interfaces before they start to evolve and redeploy their services, since they have little knowledge about their partner services. Moreover, this approach may also reduce the reusability of Web services because the extension of the interface for a service in one application may make the service unable to work in another application with differently changing requirements (which may need an altogether different service interface).

The difficulties for black-box services to efficiently handle changing requirements suggest to provide services more information about their partner services besides their statically exposed interfaces. On the other hand, the implementation details of services should be hidden to protect the business interests of service providers. These observations inspire us to go beyond black-box component-based composition approaches to explore additional runtime information from services without revealing their implementation details. In this chapter, we propose a novel grey-box approach to compose Web services and evolve them dynamically. Our approach encapsulates the internal state changes of services as *events*. For example, the hospital services may raise an event whenever the number of patients (for each type of disease) registered with the hospital services on a daily basis changes. Such events can be exposed to partner services such that these services can make use of the events to evolve and adapt their behavior to react to changing requirements. For instance, if the number of patients treated for certain diseases increases dramatically, the pharmacy services will be notified of these events. The pharmacy services will then react to the events and adapt to increase the inventory levels for specific medication used in the treatment of the disease in advance so that the response time to supply medication in response to the outbreak of the disease is shortened.

The advantage of this approach is its flexibility and ability to evolve and adapt services to react to changing requirements quickly. Services need not be re-encapsulated and re-deployed every time some domain requirement changes. Instead, services can keep their service interfaces unchanged and simply make use of events exposed from partner services to evolve and adapt their behavior. Such event-driven evolution makes the services and their interactions smarter, that is, our approach increases the flexibility and adaptability of services and reduces the services' response time as a result of changing application domain requirements. Moreover, a service can customize the encapsulation and exposure of events for different service compositions it participates in. In this way, the reusability of the service is honored, at least to a large extent.

The rest of this chapter is organized as follows: Section 2 introduces our solution methodology. Section 3 presents two case studies based on real-life examples to illustrate our approach. Section 4 analyzes the state of the art in this area. Section 5 summarizes the work, extrapolates the potential impact of this work, and presents further research questions.

2 A Grey-box Framework for Web Service Composition

2.1 Overview

In our model, we view an *event* as a change in state [18]. A *state* of a service is defined as a snapshot of its execution at runtime (e.g., the number of patients registered today). The execution of a service can be seen as a series of transitions

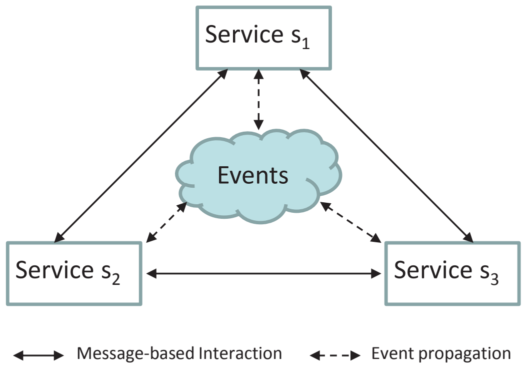

Fig. 1. Conceptual EDSOA model.

among its states. The transition from one state to another is denoted as a state change. For example, the hospital service updates its state –that is, the number of patients registered today– whenever a new patient is registered. Such states are usually invisible from outside the service, and thus referred to as *internal states*. In our approach, to allow collaborating partner services to become aware of each others' internal state changes, services convey internal state changes as events for different service compositions the services participate in and publish events to collaborating partners. By subscribing to events of interest, collaborating partner services may trigger corresponding adaptations (e.g., the pharmacy service increases its inventory level for certain medications) on being notified of such events. In this chapter, we refer to such adaptations as *event-driven adaptations*.

In this model, services in a service composition can not only interact with each other through the invocation of their exposed functions (via sending messages to their partners and receiving messages from their partners), but can also publish events to their partners and make adaptations on being notified of events from their partners. Note that events are different from messages. An event is raised to convey a service's internal state change. A service may publish events to other services, but it has neither an idea about who will subscribe to and therefore be notified of its events, nor how these events are handled by other services (e.g., whether they are discarded or trigger event-driven adaptations inside other services). In contrast, a message is a part of a communication protocol among services, which mandates and synchronizes the behavior of the service sending the message and the service receiving the message, regardless of a synchronous or asynchronous underlying mechanism to transfer messages. Messages must be received and handled in a certain order by a service. A service may be blocked to wait for a message from another service, but a service is not blocked to wait for an event since services are notified of events, which they may consume by triggering adaptations or simply discard.

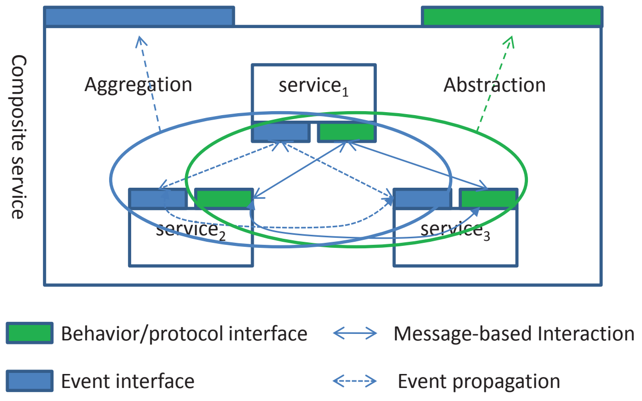

Fig. 2. Grey-Box composition of services.

To differentiate from traditional SOA service compositions that support only message-based interactions, we use the term *Event-Driven Service-Oriented Architecture* (EDSOA[1] for short) to describe the type of service compositions that support both message-based interaction and event propagation among services and corresponding event-driven adaptations. Fig. 1 illustrates the conceptual EDSOA model. In EDSOA, services are published, discovered and composed in the same way as in SOA. However, in addition to SOA, EDSOA services in a service composition may also publish events to and consume events from their partners.

The novel aspects of our work lie in that services are composed not only based on their statically exposed behavior interfaces, but also interact through events from their event interfaces. This additional dimension for service compositions increases the flexibility and adaptability of SOA applications. To a large extend, our approach also respects the reusability of services.

2.2 Grey-Box Composition of Services

One practical concern of EDSOA is that the well-established encapsulation and modular development principles of Web services should not be breached by the integration of event-driven techniques, since events may be propagated across the boundary of services in a service composition in arbitrary ways. Therefore, it is desirable to encapsulate internal state changes of Web services as events and abstract the exchange of events among services into service interfaces as well. In our model, as illustrated in Fig. 2, a service has two interfaces: a behavior interface and an event interface. The *behavior interface* is the traditional Web service interface (e.g., WSDL and WSCI [28]), which describes the functionality of the service and its communication protocols. The *event interface* of a service

[1] Note that the term "EDSOA" used in this chapter is different from the one proposed in SOA 2.0 [29]. The difference is explained in Section 4.

on the other hand describes the types of events the service wants to expose and the types of events the service consumes from its partners.

A service composition in EDSOA concerns not only the interactions of services through behavior interfaces, but also determines the event advertisements and event subscriptions among services (defined below). The service interactions based on behavior interfaces are the same as their counterparts in SOA applications. However, the event propagation among services requires additional mechanisms. In our work, we adopt the content-based pub/sub model for event propagation [6] (WS-Eventing [28] also adopts a pub/sub model). In this model, event producers (e.g., services or software modules) advertise the types of events they will generate (also referred to as *event advertisements*); event consumers (e.g., services or software modules) subscribe to the types of events they are interested in (also referred to as *event subscriptions*). Both, event advertisements and event subscriptions can be expressed as a set of event types in event interfaces. At runtime, each event is propagated from its producer to all subscribers whose subscriptions are matched by the event's content.

Services can be composed hierarchically and incrementally to form composite services. In EDSOA, a composite service also needs to provide an event interface in addition to its behavior interface, as illustrated in Fig. 2. The composite service aggregates events from the events exposed from the event interfaces of its involved services, and exposes the aggregated events in its event interface. Similarly, the composite service may subscribe to events from its partner services in its event interface. These subscribed events will be decomposed into different events and propagated to its involved services. How exposed events are aggregated from its involved services and how subscribed events are decomposed and propagated to its involved services are determined by the business logic of the composite service.

The advantage of this approach is that services need not propagate low-level events generated inside the services to all partner services. Instead, they can customize and aggregate the low-level events into more meaningful high-level events. This not only reduces the traffic due to event propagation, but also helps to hide the implementation details of a service.

2.3 Event-Driven Service Evolution

With the support of event exposure and event propagation, services can become aware of their partner services' runtime execution states. Services can make use of these events to evolve and adapt their behavior dynamically. Fig. 3 illustrates the implementation model of event-driven service evolution.

In the model, a service is equipped with queues to store subscribed events from its partner services. To evolve a service and adapt its behavior dynamically, event-condition-action (ECA [1]) rules can be added to the rule repository of a service dynamically. Whenever the service is notified of an event matching its subscriptions, the event is put into the tail of a queue. The service can then handle the events in its queues by invoking the matching ECA rules to handle the events. For example, the pharmacy services may evolve to become aware of

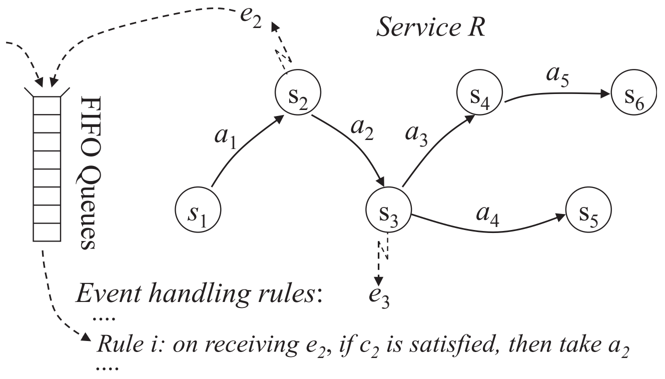

Fig. 3. Event-driven service evolution model.

the number of patients reported by the hospital services by subscribing to the hospital services' events, and add ECA rules to handle the events (i.e., increasing the inventory levels for specific medication whenever the number of patients increase dramatically).

The advantage of this approach is that it increases the flexibility and adaptability of service-oriented applications, because ECA rules can be added to the services' repository dynamically. Moreover, the reusability of services is respected since services need not be re-designed and re-deployed; their behavior interface remains unchanged.

2.4 Implementation

As a proof of concepts, we implemented a prototype of the proposed grey-box framework. In this prototype implementation, we extended BPEL 2.0 [21] to support event exposure and event-driven adaptation for business processes. To support event interfaces, each BPEL processes is attached separated XML documents defined by our event interface language called *Event Interface Description Language* (EIDL for short). Fig. 4 describes the schema of the event interface description language.

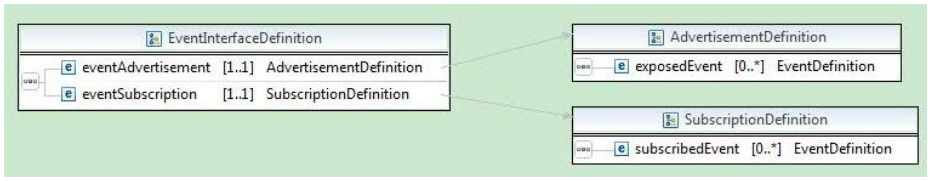
EventInterfaceDefinition
eventAdvertisement [1..1] AdvertisementDefinition
eventSubscription [1..1] SubscriptionDefinition
AdvertisementDefinition
exposedEvent [0..*] EventDefinition
SubscriptionDefinition
subscribedEvent [0..*] EventDefinition

Fig. 4. Event interface description language schema.

In BPEL, users can define event handlers to handle two types of events (receiving a message or time-out alarm). We can define event handlers in BPEL to handle events from partner services at runtime. However, this implementation requires redeploying a BPEL process whenever a new ECA rule is added to this process. Moreover, different instances of a BPEL process may need different rules at runtime. Therefore, event handlers in our prototype are defined in separate documents attached to a BPEL process and encapsulated as Web services too. In this way, whenever a new ECA rule is added for a particular instance of a BPEL process, only the corresponding new event handler is deployed as a Web service, and the BPEL process and its other instances are not affected. As a result, ECA rules can be added or removed dynamically.

Our prototype implementation is built based on the open-source project called Apache ODE (Orchestration Director Engine [2]). Apache ODE is a BPEL engine for orchestrating BPEL processes. In order to support event exposure for BPEL processes, we implemented an event listener and hooked it into the Apache ODE engine. The event listener is responsible for monitoring the events generated during the execution of BPEL processes, and exposing the events defined in the event interfaces and filtering out the others. To propagate events to the partner services, we designed a pub/sub interface that can be implemented by existing pub/sub systems (e.g., Padres [26]). When an event interface is deployed, the exposed events and subscribed events defined in the event interface are converted to corresponding advertisements and subscriptions. In this way, services can publish events to and consume events from their partner services through the underlying pub/sub system at runtime. To support event-driven adaptation, we designed a rule engine and an adapter for the underlying pub/sub system. This adapter is notified of events from partner services and the rule engine evaluates whether an ECA rule can be triggered for every incoming event. If a rule can be triggered, the rule engine will invoke the corresponding event handler, which is deployed as a Web service. The architecture of the prototype is illustrated in Fig. 5.

In the next section, we conduct two case studies to investigate and evaluate the advantage of our proposal and compare it to existing black-box solutions.

3 Case Studies

In this section, we present two case studies to illustrate our approach. In the case studies, we compare our solution to existing work to illustrate the flexibility of our approach.

3.1 Healthcare Application

This example is from a healthcare application, as illustrated in Fig. 6(a), where the public health center service (PHC for short) collaborates with hospital services to provide healthcare services for patients.

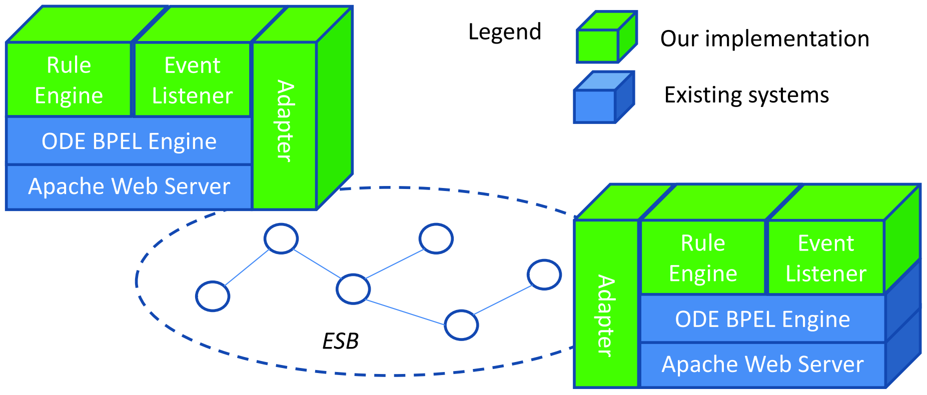

Fig. 5. Prototype architecture.

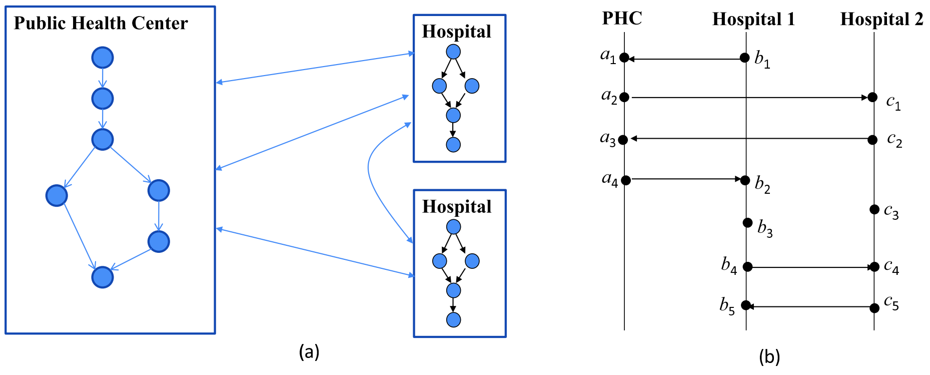

Fig. 6. Healthcare application scenario.

Let us consider a collaboration scenario in this application, as shown in Fig. 6(b). Hospital 1 sends a request message (represented as action b_1) to PHC to transfer a patient to another hospital with more adequate treatment procedures. On receiving the message (a_1), the PHC service finds that Hospital 2 satisfies the admission requirements. It then sends a patient transfer request message (a_2) to Hospital 2. If Hospital 2 agrees to receive this patient, it sends a transfer permit message to PHC (c_2), and prepares for the reception of the patient (c_3, e.g., schedules the ambulance). On receiving the permit forwarded from PHC (a_4), Hospital 1 does some preparation (b_3, e.g., physical checkup for the patient) and then sends a transfer confirmation to Hospital 2 (b_4). On receiving the transfer confirmation, Hospital 2 will arrange an ambulance to transfer the patient (c_5).

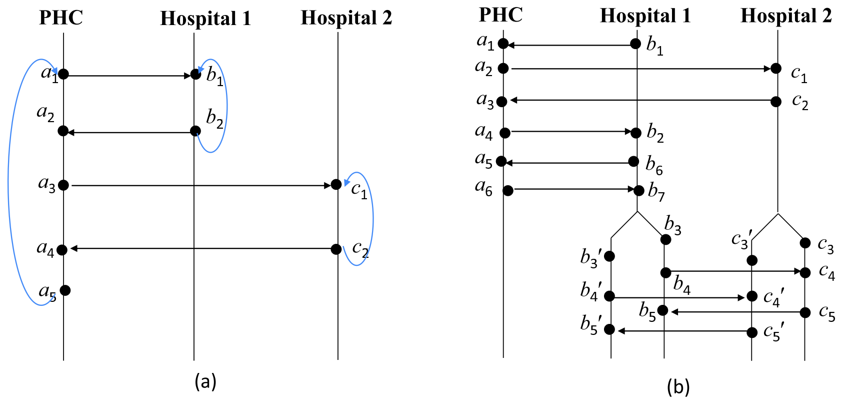

Fig. 7. Sample black-box solution.

Note that this service-oriented application works well for the given scenario. To provide more qualified healthcare services for patients, new application requirements are imposed by the public health regulator, mandating healthcare services to react to emergency situation quickly (based on a specified service level). For example, in case of a flu outbreak, the hospital services should react quickly to quarantine the patients to prevent them from being cross-contaminated. With respect to above patient transferring scenario, if a flu outbreak is reported in the area of Hospital 2, Hospital 1 should not transfer the patient to Hospital 2. Instead, Hospital 1 may request remote therapy support from Hospital 2 to avoid the patient being cross-contaminated.

To address these new requirements, the services in this application need to be re-designed, re-encapsulated and re-deployed using traditional approaches. For example, Fig. 7 illustrates an implementation of these services to cover these new requirements. In order to detect the outbreak of a flu, the public health center service will monitor and periodically query the number of patients and the corresponding types of disease in the hospital services. Therefore, these services have to be re-designed and re-encapsulated to provide new operators (i.e., b_1, b_2, c_1 and c_2) in the new interfaces to support such query functionality, as illustrated in Fig. 7(a). Similarly, when Hospital 1 wants to transfer a patient to Hospital 2, new operators (i.e., a_5 and a_6) are needed in the public health center service for Hospital 1 to query whether it is risky (i.e., whether there is a flu outbreak in Hospital 2) to transfer the patient. If so, both hospital services use the re-designed functionality to provide remote therapy support for the patient, as illustrated in Fig. 7(b).

From the above sample implementation, we can observe that traditional black-box service composition solutions lack the flexibility and adaptability to handle new application requirements, because the services need to be re-designed, re-encapsulated and re-deployed. In the rapidly evolving smart Internet environments of today, the requirements of an SOA application may change frequently.

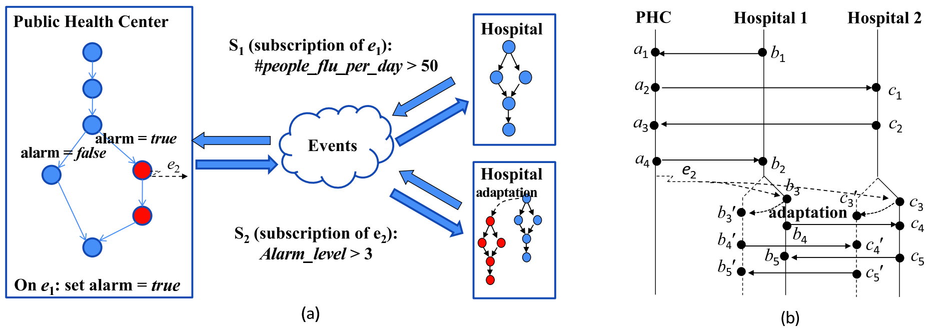

Fig. 8. Our solution.

Black-box solutions are unable to handle the changing requirement quickly. As a result, the quality of the services (e.g., availability of the services) is compromised. Moreover, since the service behavior interfaces also need to be re-designed, the reusability of the services is also compromised.

In contrast, our grey-box solution can address these changing requirements in a flexible way. In our solution, services expose some of their internal state changes as events and publish these events to their partner services. For example, as illustrated in Fig. 8(a), whenever the number of flu patients registered per day changes, the hospital service will publish an event to its partners, specifying the current number of flu patients registered per day. By subscribing to such events, if the PHC service is notified of an event indicating more than 50 flu patients per day (the number 50 is determined from historic data about flu outbreaks) are registered, the PHC service changes its flu alarm state from safe (i.e., $Alarm\_level \leq 3$) to risky (i.e., $Alarm\_level > 3$). Such an internal state change of the PHC service is defined as another event, a flu alarm event, and published to hospital services. On being notified of the flu alarm event, hospitals may adapt their regular flu treatment procedure (e.g., separate flu patients from others). For example, as illustrated in Fig. 8, during the preparation for patient transfer (b_3), if Hospital 1 is notified of an event e_2 from the PHC indicating that a serious flu outbreak is detected in Hospital 2, Hospital 1 will adapt its behavior (i.e., from b_3 to b'_3) to request support from Hospital 2 (b'_4) instead of transferring the patient to Hospital 2 to protect the patient from being cross-contaminated. Similarly, Hospital 2 also adapts its behavior to provide remote therapy services for Hospital 1 on being notified of event e_2.

Compared to traditional black-box solutions, the following advantages are observed in our grey-box solution: First, it increases the flexibility of the application, because the services need not be re-designed and re-deployed. Instead, only some events are defined and exposed to their partner services. In this way, event-driven adaptation rules (ECA rules) can be easily added to the application to handle new requirements. If the requirements change, services may only

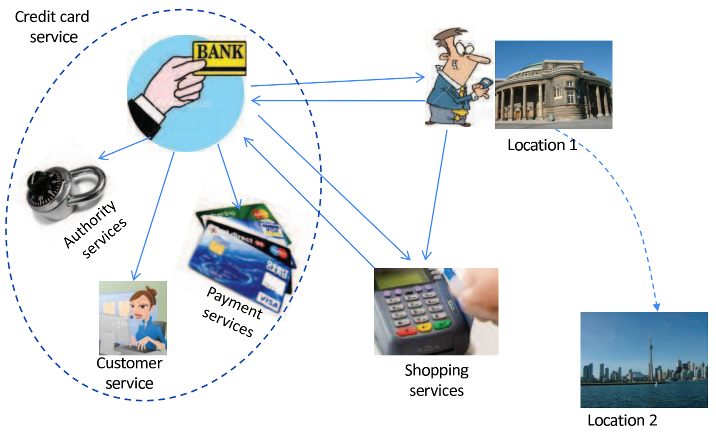

Fig. 9. E-Commerce application scenario.

need to expose different events and change the ECA rules. Next, the reusability of services are respected, because the behavior interface of some services (e.g., public health center service) remains unchanged.

3.2 E-Commerce Application

This example is from an e-commerce application, where a customer with a mobile device travels to different places. The customer uses shopping services provided by merchants and pays by credit card. Fig. 9 illustrates a typical scenario. The customer uses a credit card to pay the bill from the merchants, who then transfer the payment request to the bank. On receiving the request, the bank will check the authenticity of the card holder and approve the payment if it is an authorized transaction. Sometimes, the bank may call the customer to confirm the transaction before approving the payment if the amount paid in the transaction is larger than a specified threshold.

To provide better services for customers in such a scenario, the bank may want to improve the security of its credit card services on the one hand, and provide more flexible and smarter interactions with customers on the other hand. Let us image a scenario where the customer travels to another country, and shops with a credit card. Since the payment is from another country, to improve the security of the credit card services and protect the interests of card holders, the bank may call the customer to confirm the transaction before approving the payment. The customer whose cellphone is operating under expensive roaming charges may decide not to take the call and, thus, the transaction is declined, with all the negative consequences such a situation entails. A more flexible and smarter interactions are required to resolve this dilemma. For example, if the customer travels to another country, the bank may choose to send a short message to the customer instead of dispatching a phone call. Similarly, if security concerns about the credit card are raised during the weekend or a holiday, the bank may contact the customer via his cellphone instead of his office phone.

As explained in Section 3.1, traditional black-box approaches to enrich original services with such new functionality require refactoring or re-designing the services and re-deploying them. Such solutions lack the flexibility since whenever a new functionality is added, the services need to be re-factored or re-designed and re-deployed. This also compromises the availability of the services during the service evolution.

In the rest of this section, we illustrate our solution based on event exposure and event-driven adaptation. We show that our solution increases the flexibility and adaptability of the services for such scenarios. To implement the above described new functionality, the following events are exposed by these services:

1. $e_{location}$. In order to increase the security of the credit card services, the customer may agree to expose and publish this location change event to credit card services whenever he travels to another country. This event indicates the current location of the customer. This can be done via the GPS functionality running on the customer's mobile device.
2. e_{amount} and $e_{merchantLocation}$. These events are raised from the payment service that handles the payment transaction. These events are exposed and propagated to the card holder authenticity checking services which can make use of these events to estimate the risk of the transaction.
3. e_{alarm}. This event is raised from the authenticity services that evaluate the risk of a payment transaction. Whenever the services detect some unusual events, this event is raised and propagated to the credit card service, whose customer services will contact the card holder to confirm the transaction before approving it. Note that this event is a composite event, in the sense that it is composed of other events. For example, a payment from a location different from the current (or home) location of the card holder with an amount larger than a threshold may indicate a potential risk that the card is stolen. Therefore, this event can be defined as $e_{amount}.value > threshold \wedge e_{merchantLocation} \neq currentlocation_{customer}$. Other abnormal scenarios can be defined in a similar way.

To propagate these events, the following event interfaces are defined for the corresponding services:

1. $EI_{customer}$. This event interface is for the customer, whose mobile device service exposes event $e_{location}$ through the event interface. Other possible events may be exposed in the event interface in the future (e.g., the current state of customer: in meeting, traveling, sleeping etc.)
2. $EI_{creditcardservice}$. This is the event interface for the composite service: credit card service. The event interface subscribes to the event $e_{location}$ from customers.
3. $EI_{payment}$. This event interface is for the payment services, which expose these two events e_{amount} and $e_{merchantLocation}$.

4. $EI_{authority}$. This event interface is for the authentication services. In this event interface, two events (i.e., e_{amount} and $e_{merchantLocation}$) are subscribed to from partner services (e.g., payment services). The authority services also expose the event e_{alarm} in this event interface.
5. $EI_{customerservice}$. The customer service consumes the event e_{alarm} from the authentication services through this event interface.

Based on these exposed events and event interfaces, event-driven adaption rules can be defined to implement the new functionality, that is, increase the security and interact with customers in a flexible and smart way. Some sample rules are illustrated as follows:

1. **ECA-Rule**: *On being notified of* $e_{location}$*, set* $currentlocation_{customer} = e_{location}.value$.
   This rule is defined for the credit card service. Its intuitive meaning is to update the current location of customers.

2. **ECA-Rule**: *On being notified of* e_{alarm}*, if* $currentlocation_{customer} = location_{register}$*, call customer; else, send an SMS message.*
   This rule is defined for the customer service. The intuitive meaning is to provide a flexible and smarter way to interact with customers, that is, if the customer is traveling, then send an SMS message instead of calling him.

3. **ECA-Rule**: *On being notified of* e_{amount} *and* $e_{merchantLocation}$*, such that* $e_{amount}.value > threshold \wedge e_{merchantLocation} \neq currentlocation_{customer}$*, then raise event* e_{alarm}.
   This rule is defined for the authentication services to detect abnormal situations. Other rules can be defined to detect abnormal situations in a similar way.

In this case study, we observed the following advantages of our approach compared to existing black-box solutions: First, by making use of exposed events from partner services, it is easy to define ECA rules to support flexible and smart interactions between customers and the credit card service at low cost. Second, our solution is easy to maintain and evolve for new application requirements. For example, if a new security requirement is imposed, the services may only need to expose additional events and define new ECA rules. This exempts services from the burden to be re-designed and re-deployed. It also simplifies the design and composition of services by providing simple behavior interfaces to capture the basic business logic, leaving the frequently changing parts to be realized by event-driven adaptation rules.

3.3 Summary and Lessons Learned

In this section, we illustrate our proposal using two case studies. In the studies, new requirements are added to existing service-oriented applications. We observed that existing black-box solutions lack the flexibility and adaptability in both studies. This is because services need to be re-designed and re-deployed whenever the requirements change. Such solutions also compromise the availability and reusability of services. In contrast to black-box solutions, our solution provides an additional dimension to compose and evolve services through event exposure and corresponding event-driven adaptation. Such an additional dimension simplifies the maintenance and evolution of services, because the frequently changing behavior introduced by new requirements can be implemented by this additional dimension. As a result, services need not be re-designed and re-deployed whenever the requirement changes. This increases the flexibility and adaptability of service-oriented applications. Moreover, a service can customize different event interfaces to different service compositions and keep its behavior interface unchanged. In this way, the reusability of services is also respected.

In the study, we also learned some lessons from our solution. The additional dimension for service composition also imposes new challenges for application development. The increased flexibility and adaptability may introduce more inconsistency among services in a service composition. Services in a service composition may consume events from their partner services and make inconsistent behavior adaptations. Such inconsistent behavior adaptations may cause serious deadlock problems for a service composition at runtime which may not be expected and easily discovered.

For example, in the first case study, as illustrated in Fig. 8, if Hospital 2 does not react to event e_2 (or due to out of order event delivery), then the inconsistent adaptation between Hospital 1 and Hospital 2 will cause a deadlock, because Hospital 2 is still waiting for the transfer confirmation (i.e., c_4) from Hospital 1, whereas Hospital 1 has adapted to request a remote therapy (i.e., b'_4) from Hospital 2. As a result, both services are blocked waiting for responses from each other and thus form a deadlock.

We observed that such inconsistency issue caused by event exposure and corresponding event-driven adaptation has not been addressed in existing work. The reason is that existing work exposes only the syntax of events in the event interfaces, whereas the causality relationships among events and the effects of corresponding event-driven adaptation are missing. For example, in the industry standard Service Component Architecture (SCA) [22], event interfaces are proposed to specify event propagation among service components. W3C also proposed the WS-Eventing [28] protocol to standardize the propagation of events among services. However, the event interfaces in these standards expose only the format and syntax of events and the underlying event propagation mechanisms, ignoring how the causality relationships among events and event-driven adaptations impact the behavior of services in a service composition. With respect to the example in Fig. 8, the event interfaces generated by these approaches describe only the syntax of events e_1 and e_2. However, the event-driven adaptations in services Hospital 1 and Hospital 2 are invisible to each other. As a result, the aforementioned inconsistency issue cannot be detected based on these event interfaces.

These lessons suggest that not only the syntax of events but also their causality and effects of corresponding event-driven adaptations among services should be exposed in event interfaces. How to define and how to derive such information for event interfaces require further research efforts.

4 Related Work

In this section, we review work related to the scope of our research falling in the areas of reflection, service evolution, event-driven development, and event processing.

Software Reflection. Reflection [25] is an approach to inspect the internal state of a system and adapt its behavior dynamically. By providing programmatic access to the meta-level information about the system, the instructions of the system can be revised at runtime. Services can be implemented with reflection-oriented programming to adapt their behavior dynamically. However, this is impractical for a composite service composed of services from different organizations, because the exposure of the services' meta-level information may reveal proprietary implementation details. Our approach only needs to expose the internal runtime states of services as events and no meta-level information or other details about the implementation of services is required and revealed. The kinds of state changes revealed by events are left under the control of the service designer.

Service Evolution. The issue of service evolution has been addressed by many researchers. Casati *et al.* [7] proposed approaches to modify process descriptions and manage ongoing process instances whose description has been modified. Aalst and Basten [27] proposed to evolve services based on inheritance of workflows. A similar approach was proposed by Schrefl *et al.* [24] to evolve a class from its supertype class. All these approaches evolve services by extending or revising their behavior interfaces. Our work complements this work by evolving services from another dimension leaving the behavior interfaces unchanged.

Event-driven SOA and Event-driven BPM. The concept of "Event-driven SOA", also known as SOA 2.0, has been proposed in the community [29]. However, the meaning of this concept is different from the term "EDSOA" proposed in our work. The "Event-driven SOA" concept is proposed from the perspective of event-based business process execution and collaboration. The purpose is to define and trigger business applications based on event-driven rules instead of describing the business logic in a procedural manner. The advantage is that business applications can be executed dynamically and can react quickly to changing requirements. However, the approach proposed in our work is to provide an additional dimension to procedure-based business collaboration based on event exposure and event-driven adaptation. This additional dimension allows services to be composed and evolved in a grey-box manner. The purpose is to increase the flexibility and adaptability of SOA applications.

On the other hand, event-driven business process management has been widely adopted in enterprise applications due to the needs for increased flexibility and adaptability of business processes [8,15,17,18,19,20,23,30]. This requires effectively integrating business logic with the generation, exposure, propagation, detection and handling of events in business applications. Frei *et al.* [13] proposed to use aspect-oriented program (AOP for short) techniques to extract and expose events from legacy enterprise applications. Developers can make use of these events for refactoring the legacy applications. In addition, industry standards like BPEL [21] also support two kinds of events, namely a timeout alarm and the receiving of a message, which however are local to a BPEL process and are not propagated to other partners. The notification mechanism in BPEL is similar to event notification, but it is based on messages. BPEL processes interact through messages only. In our work, we differentiate message exchange and event propagation. More general events are supported and they can be propagated among services to support dynamic service evolution and behavior adaptation. In the SCA specification [22], events can be propagated among components, where event propagation is governed by the WS-Eventing protocol [28]. However, the causal relations between events and the effects of event-driven adaptation inside a service are not exposed in these interfaces. As a result, these event interfaces cannot be used to check the compatibility property of services. In addition, Rapide [18] applies a top-down approach to design event-driven applications. However, the methodology to compose and evolve services based on events is not addressed in these approaches.

Interfaces for services and components. In component-based development, interfaces are used to abstract the functionality of components and encapsulate their access points. The composition of components should conform to the requirements specified in their interfaces. Different interfaces may specify different requirements. For example, Beyer *et al.* [4] proposed to specify three kinds of constraints in Web service interfaces, namely signature constraints, consistency constraints, and protocol constraints. These constraints define the correctness requirements for syntax level, functionality level and collaboration level of a service. Alfaro and Henzinger [9] proposed to describe interfaces as automata to capture temporal aspects of constraints. Their approach provides a type system to check the compatibility of interface modules based on game theory. Scalability is a major concern for compatibility checking based on interface automata. To solve this problem, Emmi *et al.* [10] proposed a modular verification approach based on assume-guarantee rules. However, the effectiveness of this approach depends on how to derive the appropriate assumption for each model. The event interface proposed in our work is difference from existing service interfaces. The difference is that message-styled communication is used to connect service interfaces based on message names, whereas the asynchronous content-based pub/sub event propagation model is adopted for event interfaces in our approach. In addition, interfaces of components in reactive systems [14] usually include only the input and output signals of the components, and the effects of the signal handling are left to the behavior description of the component. This is not feasible in Web services because the implementation of services is usually invisible to others. On the other hand, many equivalence relationships between two processes are defined in process algebras (e.g., CSP) [3]. These equivalence relationships can be used to derive interfaces from service implementations under different equivalence semantics. However, these relationships cannot be applied to derive event interfaces for Web services because they are based on synchronous message-based communication.

Event Processing Techniques. Currently, many middleware systems have been proposed to support event processing. Representative ones include content-based pub/sub systems [6,11,26], which can be used to store and deliver events from where they are produced to where they are consumed. In addition, some of these systems may support the processing of complex events (e.g., composite subscriptions [16]). Fiege *et al.* [12] further proposed to scope pub/sub applications to increase the flexibility. Our work supports event aggregation and decomposition for a composite service. More complex event processing can be supported by integrating new services implementing such complex event processing.

5 Conclusions and Future Work

In this chapter, we proposed a grey-box approach to compose and evolve Web services. Our approach is based on events specified in event interfaces which complement the traditional behavior interfaces of services as an additional dimension. Events capture the internal state change of a service. The advantage is that a service can make use of this information to better adapt its behavior to the internal state changes of its partner services. This increases the flexibility and adaptability of service-oriented applications to react faster to changing application requirements. This also honors the reusability of services.

However, the additional dimension for service compositions also imposes new challenges for application development. The increased flexibility and adaptability may introduce more inconsistency among services of a composition. Services in a service composition may consume events from their partner services and make inconsistent behavior adaptations. Such inconsistent behavior adaptations may cause serious deadlock problems for a service composition at runtime which may not be expected and easily discovered. In addition, a subsequent evolution with new behavior adaption inside a legacy service may also introduce deadlocks into an originally deadlock-free service composition. Therefore, the detection of deadlock and the guarantee of deadlock-free evolution of services augmented with events is an important research question. In particular, the exposure of events in event interfaces and the kind of information required for this exposure are also important concerns. We plan to study how to derive some equivalence relationships to guide the evolution of a service without violating its original properties.

Another research question pertains to the aggregation and decomposition of events inside a composite service, especially for a top-down design of a service composition. This should be reflected in a service composition specification. A methodology to consider both the behavioral aspects and event aggregation is needed. Moreover, new approaches are needed to discover and select services based on the additional dimension enabled through event exposure and event propagation. Other research questions such as a service quality-of-service, service maintainability, and service transactions may need new attention and require rethinking in sight of event exposure.

In addition, we also plan to utilize the additional information provided by event exposure to test and debug service compositions. Services in a service composition are usually tested by using black-box testing approaches due to the unavailable implementation details of services. Our approach sheds light on testing service compositions in a grey-box manner, that is, test cases may be designed to cover the service implementation in a more accurate way with the help of exposed events in the service's event interface. Moreover, these exposed events may also be helpful to diagnose faults in a service composition. We plan to study how to expose events and how to derive test cases to test service composition based on event exposure.

This work has the potential to impact the way services are developed and composed. The work opens a door for service developers and service consumers to rethink services (which have been viewed as black-box components for a long time) and the way services are composed (usually viewed as nothing new but a special case of component composition). In our work, a service is no longer a simple and "dead" black-box component with a statically exposed interface, but a "live" component that not only exposes static interfaces but also consumes and exposes dynamic information and dynamically adapts its behavior. The exposure of additional information as events will be helpful to SOA applications in many ways, such as contribute to their flexibility, adaptability, evolution, testing, and debugging and so on. The solutions to the aforementioned research issues may also encourage some current industry standards (such as WSDL and SCA) to further integrate events and event-related information into their specifications.

Acknowledgments

This research was in part supported by IBM's Center for Advanced Studies, an IBM Faculty Award, a Discovery Accelerator Supplement and a Discovery grant from the Natural Sciences and Engineering Research Council of Canada, an Ontario Early Researcher Award, and a grant from the Mathematics of Information Technology and Complex Systems research network.

References

1. C. Act-Net Consortium. The active database management system manifesto: a rulebase of ADBMS features. *SIGMOD Rec.*, 25(3):40–49, 1996.
2. Apache. Apache orchestration director engine. http://ode.apache.org/index.html.
3. J. A. Bergstra. *Handbook of Process Algebra.* Elsevier Science Inc., New York, NY, USA, 2001.
4. D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *WWW '05*, pages 148–159, New York, NY, USA, 2005. ACM.

5. M. Broy, I. H. Krüger, and M. Meisinger. A formal model of services. *ACM TOSEM*, 16(1):5, 2007.
6. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM TOCS*, 19(3):332–383, 2001.
7. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. In *ER'96*, pages 438–455, London, UK, 1996. Springer-Verlag.
8. T. Chau, V. Muthusamy, H.-A. Jacobsen, E. Litani, A. Chan, and P. Coulthard. Automating sla modeling. In *CASCON '08*, pages 126–143, New York, NY, USA, 2008. ACM.
9. L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9*, pages 109–120, New York, NY, USA, 2001. ACM.
10. M. Emmi, D. Giannakopoulou, and C. S. Păsăreanu. Assume-guarantee verification for interface automata. In *FM '08*, pages 116–131, Berlin, Heidelberg, 2008. Springer-Verlag.
11. F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD'01*, pages 115–126, 2001.
12. L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering event-based systems with scopes. In *ECOOP '02*, pages 309–333, London, UK, 2002. Springer-Verlag.
13. A. Frei, A. Popovici, and G. Alonso. Eventizing applications in an adaptive middleware platform. *IEEE DSO*, 6(4):1, 2005.
14. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and a. Shtul-Trauring. Statemate: a working environment for the development of complex reactive systems. In *ICSE '88*, pages 396–406, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
15. S. Hu, V. Muthusamy, G. Li, and H.-A. Jacobsen. Distributed automatic service composition in large-scale systems. In *DEBS '08*, pages 233–244, New York, NY, USA, 2008. ACM.
16. G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware'05*, pages 249–269, New York, NY, USA, 2005. Springer-Verlag New York, Inc.
17. G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Trans. Web*, 4(1):1–33, 2010.
18. D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley, Boston, MA, USA, 2001.
19. V. Muthusamy and H.-A. Jacobsen. BPM in cloud architectures: Business process management with SLAs and events. In *BPM '10*, pages 5–10, Hoboken, New Jersey, USA, 2010. Springer-Verlag.
20. V. Muthusamy, H.-A. Jacobsen, P. Coulthard, A. Chan, J. Waterhouse, and E. Litani. Sla-driven business process management in soa. In *CASCON '07*, pages 264–267, New York, NY, USA, 2007. ACM.
21. OASIS. BPEL 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.
22. OSOA. SCA event processing. http://www.osoa.org/.
23. M. P. Papazoglou and W.-J. Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.
24. M. Schrefl and M. Stumptner. Behavior-consistent specialization of object life cycles. *ACM TOSEM*, 11(1):92–148, 2002.
25. B. C. Smith. *Procedural Reflection in Programming Languages.* PhD thesis, MIT, 1982.

26. M. systems research group. Padres. http://research.msrg.utoronto.ca/Padres/.
27. W. M. P. van der Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *TCS*, 270(1-2):125–203, 2002.
28. W3C. WSCI, WSDL, WS-Eventing. http://www.w3.org.
29. Wiki. SOA 2.0. http://en.wikipedia.org/wiki/Event-driven_SOA.
30. W. Yan, S. Hu, V. Muthusamy, H.-A. Jacobsen, and L. Zha. Efficient event-based resource discovery. In *DEBS '09*, pages 1–12, New York, NY, USA, 2009. ACM.