Using Reo for Service Coordination

Alexander Lazovik* and Farhad Arbab

CWI, Amsterdam, Netherlands
{a.lazovik,farhad.arbab}@cwi.nl

Abstract. In this paper we address coordination of services in complex business processes. As the main coordination mechanism we rely on a channel-based exogenous coordination language, called Reo, and investigate its application to service-oriented architectures. Reo supports a specific notion of composition that enables coordination of individual services, as well as complex composite business processes. Accordingly, a coordinated business process consists of a set of web services whose collective behavior is coordinated by Reo.

1 Introduction

The current set of web service specifications defines protocols for web service interoperability. On the base of existing services, large distributed computational units can be built, by composing complex compound services out of simple atomic ones. In fact, composition and coordination go hand in hand. Coordinated composition of services is one of the most challenging areas in SOA. A number of existing standards offer techniques to compose services into a business process that achieves specific business goals, e.g., BPEL. While BPEL is a powerful standard for composition of services, it lacks support for actual coordination of services. Orchestration and choreography, which have recently received considerable attention in the web services community and for which new standards (e.g., WS-CDL) are being proposed, are simply different aspects of coordination. It is highly questionable whether approaches based on fragmented solutions for various aspects of coordination, e.g., incongruent models and standards for choreography and orchestration, can yield a satisfactory SOA. Most efforts up to now have been focused on statically defined coordination, expressed as compositions, e.g., BPEL. To the best of our knowledge the issues involved in dynamic coordination of web services with continuously changing requirements have not been seriously considered. The closest attempts consider automatic or semi-automatic service composition, service discovery, etc. However, all these approaches mainly concentrate on how to compose a service, and do not pay adequate attention to the coordination of existing services.

In this paper we address the issue of coordinated composition of services in a loosely-coupled environment. As the main coordination mechanism, we rely

* This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme.

B. Krämer, K.-J. Lin, and P. Narasimhan (Eds.): ICSOC 2007, LNCS 4749, pp. 398–403, 2007.

on the channel-based exogenous coordination language Reo, and investigate its application to SOA. Reo supports a specific notion of composition that enables coordinated composition of individual services as well as composed business processes. In our approach, it is easy to maintain loose couplings such that services know next to nothing about each other. It is claimed that BPEL-like languages maintain service independence. However, in practice they hard-wire services through the connections that they specify in the process itself. In contrast, Reo allows us to concentrate only on important protocol decisions and define only those restrictions that actually form the domain knowledge, leaving more freedom for process specification, choice of individual services, and their run-time execution. In a traditional scenario, it is very difficult and cost-ineffective to make any modification to the process, because it often has a complex structure, with complex relationships among its participants. We believe having a flexible coordination language like Reo is crucial for the success of service-oriented architectures.

The rest of the paper is organized as follows. In Section 2 we consider Reo as a modeling coordination language for services. A discussion of coordination issues, together with a demonstrating example and tool implementation discussion appears in Section 3. We conclude in Section 4, with a summary of the paper and a discussion of our further work.

2 The Reo Coordination Language

The Reo language was initially introduced in [1]. In this paper, we consider adaptation of general exogenous coordination techniques of Reo to service-oriented architecture. In our setting, Reo is used to coordinate services and service processes in an open service marketplace.

Reo is a coordination language, wherein so-called *connectors* are used to coordinate components. Reo is designed to be exogenous , i.e. it is not aware of the nature of the coordinated entities. Complex connectors are composed out of primitive ones with well-defined behavior, supplied by the domain experts. *Channels* are a typical example for primitive connectors in Reo. To build larger connectors, channels can be attached to nodes and, in this way, arranged in a circuit. Each channel type imposes its own rules for the data flow at its ends, namely synchronization or mutual exclusion. The ends of a channel can be either source ends or sink ends. While source ends can accept data, sink ends are used to produce data. While the behavior of channels is user-defined, nodes are fixed in their routing constraints. It it important to note, that the Reo connector is stateless (unless we have stateful channels introduced), and its execution is instantaneous in an all-or-none matter. That is, the data is transferred from the source nodes to sink nodes without ever being blocked in the middle, or not transferred at all. Formally, a Reo connector is defined as follows:

Definition 1 (Reo connector). *A* connector $\mathcal{C} = \langle \mathcal{N}, \mathcal{P}, E, node, prim, type \rangle$ *consists of a set* $\mathcal{N}$ *of* nodes*, a set* $\mathcal{P}$ *of* primitives*, a set* E *of* primitive ends *and functions:*

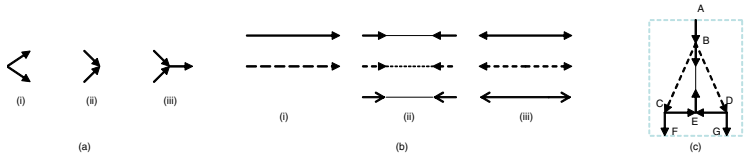

Fig. 1. Reo elements: (a)–nodes; (b)–primitive channels; (c)–XOR connector

- $prim : E \rightarrow \mathcal{P}$, *assigning a primitive to each primitive end,*
- $node : E \rightarrow \mathcal{N}$, *assigning a node to each primitive end,*
- $type : E \rightarrow \{src, snk\}$, *assigning a type to each primitive end.*

Definition 2 (Reo-coordinated system). $\mathcal{R} = \langle \mathcal{C}, \mathcal{S}, serv \rangle$, *where:*

- $\mathcal{C}$ *is a Reo connector;*
- $\mathcal{S}$ *is a set of coordinated services;*
- $serv : \mathcal{S} \rightarrow 2^E$ *attaches services to primitive ends* E *of the connector* $\mathcal{C}$.

Services represent web service operations in the context of Reo connectors. Services are black boxes, Reo does not know anything about their internal behavior except the required inputs and possible outputs that are modeled by the *serv* function. By this definition, services are attached to a Reo connector through primitive ends: typically to write data to source ends, and read from sink ends. Note that although we consider services as a part of a coordinated system, they are still external to Reo. Services are independent distributed entities that utilize Reo channels and connectors to communicate. The service implementation details remain fully internal to individual elements, while the behavior of the whole system is coordinated according to the Reo circuit.

Nodes are used as execution logical points, where execution over different primitives is synchronized. Data flow at a node occurs, iff (i) at least one of the attached sink ends provides data and (ii) all attached source ends are able to accept data. Channels represent a communication mechanism that connects nodes. A channel has two ends which typically correspond to in and out. The actual channel semantics depends on its type. Reo does not restrict the possible channels used as far as their semantics is provided. In this paper we consider the primitive channels shown in Figure 1-(b), with (i)–communication channels; (ii)–drain channels; and (iii)–spout channels. The top three channels represent synchronous communication. A channel is called *synchronous* if it delays the success of the appropriate pairs of operations on its two ends such that they can succeed only simultaneously. The bottom three channels (visually represented as dotted arrows) are *lossy* channel, that is, communication happens but the data can be lost if nobody accepts it. For a more comprehensive discussion of various channel types see [1]. It is important to note that channels can be composed into a connector that is then used disregarding its internal details. An example of such composed connector is a XOR element shown in Figure 1-(c). It is built out of five sync channels, two lossy sync channels, and one sync drain. The intuitive behavior of this connector is that data obtained as input through A is

delivered to one of the output nodes F or G. If both F and G is willing to accept data then node E non-deterministically selects which side of the connector will succeed in passing the data. The sync drain channel B-E and the two C-E, D-E channels ensure that data flows at only one of C and D, and hence F and G.

More details on the intuitive semantics of Reo is presented in [1] and in an extended version of this paper [5]. Various formal semantics for Reo are presented elsewhere, including one based on [2], which allows model checking over possible executions of Reo circuit, as described in [3].

3 Building Travel Package in Reo

To illustrate our ideas we use a simple example that is taken from the standard travel domain. We consider reserving a hotel and booking transportation (flight or train in our simplified setting). This process is simple, and works for most users. However, even typical scenarios are usually more complicated with more services involved. Our simple process may be additionally enriched with services that the average user may benefit from, e.g., restaurants, calendar, or museum services. However, it is difficult to put all services within the same process: different users require different services sharing only a few common services.

Traditionally, when a process designer defines a process specification, he must explicitly define all steps and services in their precise execution order. This basically means offering the same process and the same functionality to all users that potentially need to travel. This makes it difficult to add new services, since only a limited number of users are actually interested in the additional services. We first consider some particular user's travel expectations:

> *A trip to Vienna is planned for the time of a conference; a hotel is desired in the center or not far from it; in his spare time, the client wishes to visit some museums; he prefers to have a dinner at a restaurant of his choice on one of the first evenings.*

Hard-coded business process specifications cannot be used effectively for such a complex yet typical goal with a large number of loosely coupled services. The problem is that the number of potential additional services is enormous, and every concrete user may be interested in only a few of them. Having these considerations in mind, the business process is designed to contain only basic services with a number of external services (or other processes) that are not directly a part of the process, but a user may want them as an added value, e.g., museum and places to visit, or booking a restaurant.

One of the possible Reo representations is provided in Figure 2. Box A corresponds to the process with basic functionality. The client initiates the process by issuing a request to the hotel service. If there are no other constraints, the process non-deterministically either reserves a flight or a train and proceeds to payment. Note, that the hotel service is never blocked by the location synchronization channels (between the hotel and the XOR (see Figure 1-(c)) element) since they all are connected by lossy channels. In Figure 2 the flight service is

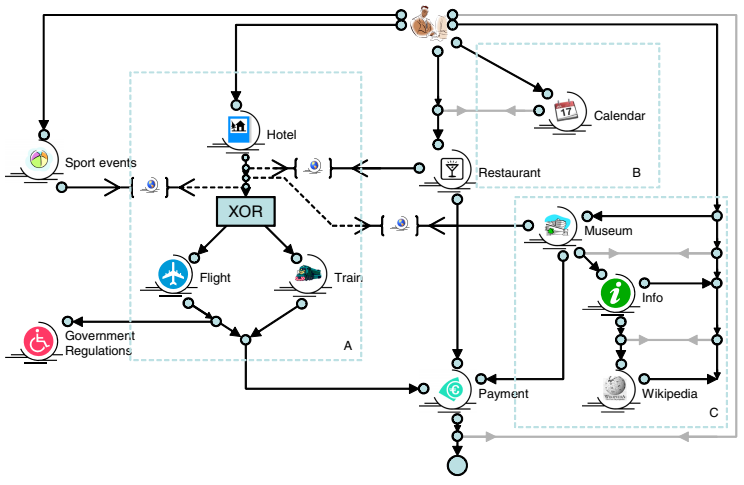

Fig. 2. A travel example in Reo

additionally monitored by a government service, that is, a flight booking is made only if the government service accepts the reservation.

Box *B* corresponds to the user request for visiting a restaurant located not far from the hotel. It is modeled as follows. The restaurant service itself is connected to the hotel using the location synchronization channel, that is, the restaurant service is invoked *only* if the hotel location is close. The location synchronization channel is a domain-specific example of a primitive channel supplied by the domain designers. It models a synchronization based on a physical location [5]. The synchronization is unidirectional: the hotel is reserved even if there are no restaurants around. We also use a calendar service to check if the requested time is free, and if it is, then the calendar service fires an event, that is, through the synchronization channel, enables the restaurant service.

Box *C* shows a possible interactive scenario for requesting a museum visit. If the user issues the corresponding request, the museum service is checked if it is close to the hotel. Then it may show additional information from the tourist office, or, if the user is interested, point to corresponding information from the Wikipedia service. User interaction is modeled via a set of synchronization channels, each of which defines whether the corresponding service is interesting to the user. Finally the payment service is used to order the requested travel package. In this example the payment service is used as many times as it has incoming events. For the real world application, it is practical to change the model to enable the user to pay once for everything.

Using our example, we have just shown how Reo can be used to coordinate different loosely-coupled services, and, thereby, extending the basic functionality of the original basic process. An advantage of Reo is that it allows modeling to reflect the way that users think of building a travel package: for each goal, we just have to add a couple of new services, add some constraints in terms of channels and synchronizations, and we have a new functionality available.

The Reo coordination tool [4] is developed to aid process designers who are interested in complex coordination scenarios. It is written in Java as a set of plug-ins on top of the Eclipse platform (`www.eclipse.org`). Currently the framework consists of the following parts: (i) graphical editors, supporting the most common service and communication channel types; (ii) a simulation plug-in, that generates Flash animated simulations on the fly; (iii) BPEL converter, that allows conversion of Reo connectors to BPEL and vice versa; (iv) java code generation plug-in, as an alternative to BPEL, represents a service coordination model as a set of java classes; (v) validation plug-in, that performs model checking of coordination models represented as constraint automata.

4 Conclusions and Future Work

In this paper we presented an approach for service coordination based on the exogenous coordination language Reo. It focuses on only the important protocol-related decisions and requires the definition of only those restrictions that actually form the domain knowledge. Compared to traditional approaches, this leaves much more freedom in process specification. Reo's distinctive feature is a very liberal notion of channels. New channels can be easily added as long as they comply with a set of non-restrictive Reo requirements. As a consequence of the compositional nature of Reo, we have convenient means for creating domain-specific language extensions. This way, the coordination language provides a unique combination of language mechanisms that makes it easy to smoothly add new language constructs by composing existing language elements.

In this paper we assumed that services support a simplified interaction model. While this is acceptable for simple information providers such as map or calendar services, this assumption is not true in general. We plan to investigate the possibility of using Reo in complex scenarios where services have extended lifecycle support. Reo is perfect in defining new domain-specific language extensions. However, we lack specific extensions to the coordination language that support various issues important to services, e.g., temporal constraints, preferences, and extended service descriptions.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Structures in CS 14(3), 329–366 (2004)
2. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. 61(2), 75–113 (2006)
3. Klueppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. In: FOCLASA'06 (2006)
4. Koehler, C., Lazovik, A., Arbab, F.: ReoService: coordination modeling tool. In: ICSOC-07, Demo Session (2007)
5. Lazovik, A., Arbab, F.: Using Reo for service coordination. Technical report, CWI (2007)