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 coordinated composition 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 a Reo expression.
In this approach, it is easy to maintain a loosely coupled environment with services knowing nothing about each other. Although it is claimed that BPEL-like languages maintain service independence, 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 and choice of individual services compared to traditional approaches.

1 Introduction

Service-oriented architecture (SOA) is an emerging paradigm for distributed computing and e-business processing that has evolved from object-oriented and component-based computing, to enable building agile networks of collaborating business applications, distributed within and across organizational boundaries. A number of challenging issues need to be addressed before the service-oriented computing paradigm becomes reality. These challenges include service modeling and design methodologies, architectural approaches, service development, deployment and composition, programming and evolution of services as well as their supporting technologies and infrastructure.

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 [5].

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

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 (compositions), as in 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 a main coordination mechanism, we rely 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. Traditionally, when a process designer produces a process specification, he must explicitly define all steps and services in precise execution order. Often, the result is an over-specification of what the actual process is really about, which is made necessary only by the attempt to express the specification using the available (essentially sequential, imperative, and process oriented) languages and their supporting tools. This makes it difficult to adapt such over-specified processes to accommodate the needs of all users who require the same functionality, but perhaps with some non-essential deviations its implementation. By placing interaction and its coordination at the center of attention, Reo lifts the level of abstraction for the specification of composed processes.

This paper is organized as follows. In Section 1.1 we provide an overview of related work. A discussion of coordination issues, together with a demonstrating example appear in Section 2. In Section 3 we consider Reo as a modeling coordination language for services. We describe a reference implementation for the Reo

framework and its architectural issues in Section 4. We conclude in Section 5, with a summary of the paper and a discussion of our further work.

1.1 Related work

Service coordination is managing the interactions among different business processes and the atomic services that they may entail. Several channel-based languages and frameworks have been developed for composition and coordination of (distributed) entities, e.g., Petri nets, π-calculus, coordination based on mobile channels [16], stream calculus [15]. However, these general frameworks do not particularly cater to certain issues in service-oriented computing, e.g., nondeterministic nature of services or late service implementation binding.

A number of approaches to service coordination are based on rich service semantic descriptions, e.g., knowledge-based semantic web service composition [7], service discovery and composition based on semantic matching [14], and semiautomatic composition of web services based on semantic descriptions [17]. However, the assumption of rich service semantic descriptions is still too strong in practice today.

The declarative nature of business rules and objectives has spawned several frameworks using rule-based languages. The WS-Policy framework provides a general purpose model for describing a broad range of service requirements, preferences, and capabilities. Typically it is used when a provider describes the set of conditions that a requester should satisfy before invoking the service. RuleML [9] is a powerful technique for expressing business rules over semantically annotated services. Extensions of RuleML for distributed service-oriented environments are proposed in [13]. However, general frameworks based on business rules are not well suited to deal with process coordination needs because they lack visual representation of coordination; are often not composable; and are not easy to extend to define new domain-specific coordination languages.

One of the approaches to complex coordination is automatic service composition. A configurable approach to service composition is proposed in [6]. To support dynamic composition of web services from existing web services and dynamic integration of their data, [18] proposes a data integration technique. In [4] services are modeled as execution trees represented by deterministic finite state machines. In [12] the Golog planner is used to automatically compose semantically described services. The *HTN* planner SHOP2 was applied for web service composition in [19]. In [11] an XML Service Request Language was introduced in order to express complex user goals and preferences over partially composed business processes. Up to now most efforts in service composition have been concentrated on how to combine atomic services. In this paper we consider an alternative approach to service coordination that allows us not only to compose atomic as well as composed services, but also to alter existing compositions by adding new elements.

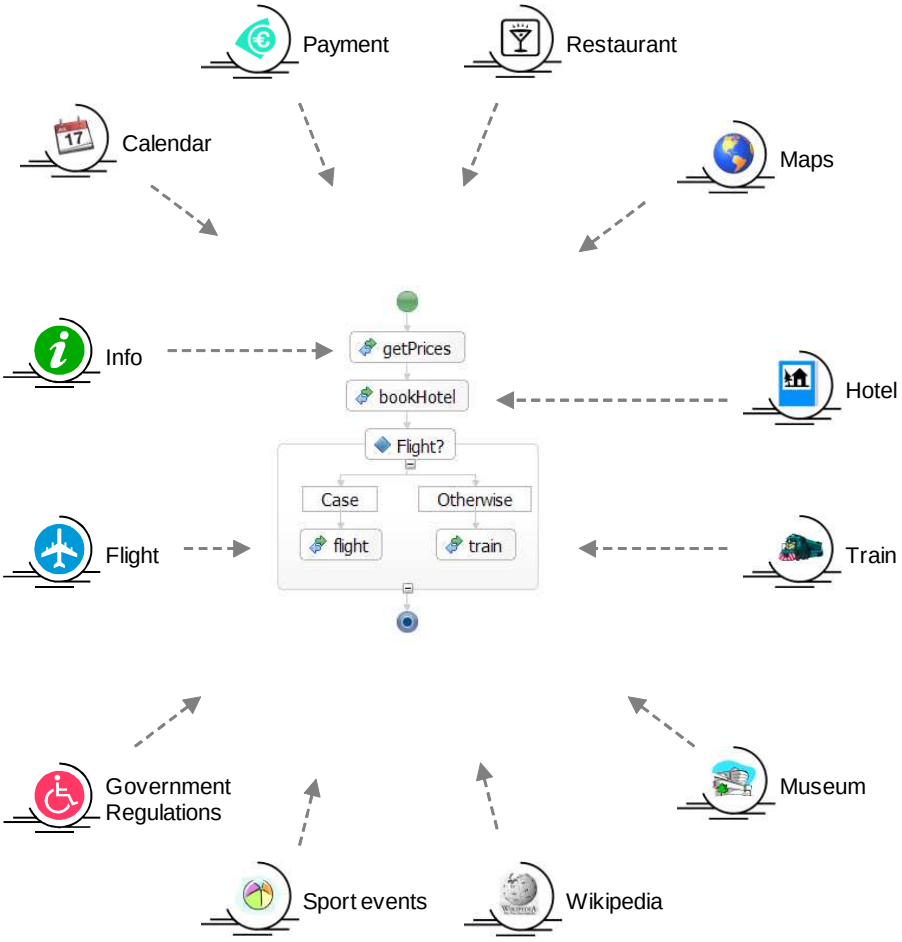

Fig. 1. Sample process with additional services

2 Need for coordination

To illustrate our ideas through the paper 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). A possible BPEL code for building a travel package is shown in the center in Figure 1. The process works as follows. First, it requests an information provider for a hotel and prices for flight and train. Then it tries to book the hotel, and if successful, it reserves the flight or the train, depending on some conditions, e.g., user's choice.

The provided 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, museums, etc.; see Figure 1 for more such 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 precise execution order. That basically means to offer the same process and the same functionality to all users who potentially need to travel. This makes it difficult to add new services, since only a limited number of users are actually interested in additional services. Moreover, even with basic services, as booking flights and hotels, it is difficult to say which service has to be put in the process first. For an average trip, reserving a flight implies looking for a hotel, since taking a flight ticket is slightly more "risky". But if you consider Christmas time and overloaded hotels at ski resorts, you may probably want to book the hotel first, and then, if successful, start looking for a suitable means of transportation to get to the reserved hotel.

Before discussing possible service coordination for such complex scenarios, let us first consider some particular user 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 (as we did in Figure 1) 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., museums and places to visit, or booking a restaurant.

We assume to have a marketplace of properly described corresponding services. The likely reasoning is then done as follows. First, we try to reserve a hotel for specific dates. We check the price, give it to the user, and, with his approval, reserve a room. The additional requirement of the hotel being in the center, checks only hotels in the center. This may be checked by a separate service like

Google Maps (assuming it is described as a separate service). When the hotel is about to be reserved, the flight is fetched and booked in a similar way. To schedule museum visits, we may also use a scheduler service (ex., Google Calendar) to arrange for proper timing. If the user provides his own calendar, the system may also take it into account. Additionally, the tourist package service may have a relation with the description of some museums – say, from Wikipedia. The restaurant service is used in the same way, while we may additionally ensure its location is close to the hotel.

Having this scenario working would require a language that in a fast and convenient way coordinates behavior of services, exposing the possibility to use new services with minimum strict requirements. In this way, the language can be seen as a high-level description of possible scenarios with easier integration of new services to coordinated "networks of services".

To accomplish this, we use the coordination language Reo that has the following important properties: it is (i) declarative; (ii) dynamic; (iii) compositional. *Declarative* language describes what the desired protocol is, rather than how to create it. This enables us to specify service coordination in a high-level abstract way, describing what the protocol should contain: services, communication channels, business objectives, but not how to actually execute services in any particular order, as it is done, for example, in BPEL. The declarative nature of the language allows not to over-impose constraints where they are not needed, leaving more freedom for each particular process execution to satisfy even the most complicated client preferences.

The *dynamic* nature of the language enables fast inclusion of new services, communication channels, synchronization points, and monitoring mechanisms. This is crucially important for open service-oriented environments, with new services becoming available, new business opportunities arising, or new government regulations arriving that must be taken into account.

The coordination language is *compositional* from two different perspectives. First, easy and fast composition of existing services and processes. Second, the composition of the elements of the Reo language themselves. For example, consider a synchronization point between two services: hotel and restaurant reservations. We must ensure that the two services are executed only if restaurant is close to the hotel. Such a scenario can be repeated for different pairs of services: hotel and museum, local transportation, etc. Instead of re-developing it every time, we may infer a new language expression called, say, location synchronization. It hides the details of how it synchronizes services, allowing the process coordinator to think at the domain level. With such internal details hidden, we have a convenient mechanism for creating domain-specific language extensions. This way, the coordination language provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs by composing existing language elements, channels, and services.

3 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:*

- $prim : E \rightarrow \mathcal{P}$*, 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

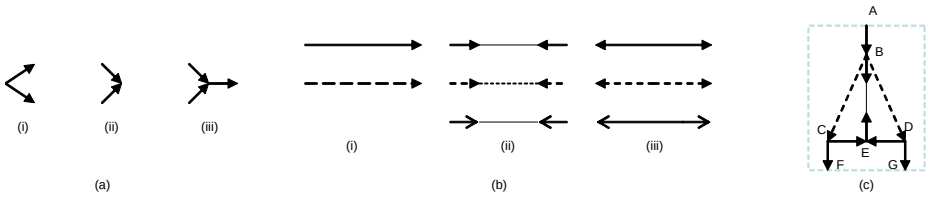

Fig. 2. Reo elements: (a)–nodes; (b)–primitive channels; (c)–XOR connector.

attached sink ends provides data and (ii) all attached source ends are able to accept data. A node does a destructive read at one of its sink end and replicates the data item to all its source ends.

A channel is the most typical example of a primitive. A channel represents 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 2-(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].

In the context of coordination of service compositions using Reo, services may perform the following operations on channels connected to them:

- `take`. The `take` operation succeeds when an entity can successfully read from the sink side of a channel and there is a suitable data available in that channel. The `take` operation is destructive, that is, the data item is removed from the channel. The `read` operation is a non-destructive version of `take`. After performing `read` the corresponding data item is copied, but not removed from the channel;
- `write`. The `write` operation on a channel source end succeeds when channel is ready to accept the data. Depending on the channel type, there are additional requirements for a `write` operation to succeed, e.g., a synchronous channel requires a corresponding `take` operation to be performed at the sink end of the channel.

In the general coordination framework there are also other types of operations: `connect`, `disconnect`, `move` [1], that are less interesting for our purposes.

It is important to note that channels can be composed into a more complex connector that is then used disregarding its internal details. An example of such composed connector is a XOR element shown in Figure 2-(c). It is built out of five sync channels, two lossy sync channels, and one sync drain. The intuitive

Channels	source-sink	2 sinks (drain)	2 sources (spout)
synchronous `offer(c)`:	`offer(src(c))`	-	$\wedge$ `offer(src(c))`
`accept(c)`:	`accept(snk(c))`	$\wedge$ `accept(snk(c))`	-
lossy `offer(c)`:	`offer(src(c))`	-	$\vee$ `offer(src(c))`
`accept(c)`:	`true`	$\vee$ `accept(snk(c))`	-

Table 1. Channel semantics.

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.

We now define the semantics of nodes and channels more precisely. As a basis for our description we use two boolean functions, `offers` and `accepts`, that basically denotes whether a node or a channel is ready for corresponding `write` and `take` operations. In general, control flow in Reo is performed from source to sink nodes through channels and mixed nodes. A transition is performed whenever `offers`=`accepts`=`true`. We now show how values of `accept` and `offer` are propagated along the execution:

- The node offers data if at least one incoming channel is ready to offer data: `offer(n)` = $\vee$ `offer`$(snk(n))$. If several channels offer data, then all of them are processed one by one until data is accepted. For source nodes `offers(n)` is true whenever a `write` operation is performed on the source node by the connected service;
- The node accepts the data to be written if all its outgoing channels are ready to accept data: accepts(n) = $\wedge$ `accepts`$(src(n))$. The data item then is replicated to all outgoing channels. For sink nodes `accepts(n)` is true whenever a `take` operation is performed on the node. In terms of SOA, `accepts(n)` is true if the connected service is ready to accept a message.

Note that a `read` operation does not affect the `accept` function. However, if `offer` = `true` a `read` operation can be used to "look-ahead" on the possible `write` data to reason on whether the corresponding component, channel or node is willing to accept the data with a `take`.

The behavior of our channels is shown in Table 1. A synchronous channel with source and sink ends transfers data only if both ends are ready. The lossy sync channel always accepts data, even if the node at its sink end is not accepting, in which case the data is lost. A synchronous drain may be seen as synchronization between two nodes: the data item is accepted only simultaneously from both nodes.

More details on the intuitive semantics of Reo is presented in [1, 2]. Various formal semantics for Reo are presented elsewhere, including one based on [3], which allows model checking over possible executions of Reo circuit, as described in [10]. In [8] Reo semantics is defined in terms of connector coloring.

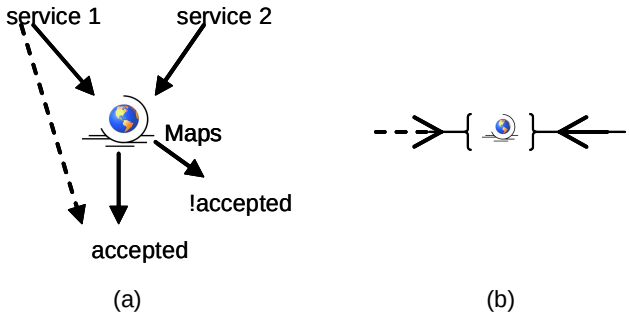

Fig. 3. Synchronization channel based on location.

3.1 Building the travel package in Reo

One of the main features of Reo is composability of its primitive elements into more complex connectors. One of the examples is a XOR element that is shown in Figure 2-(c). We know consider a more interesting, domain-specific example of a synchronization based on physical location. In the example introduced in Section 2 a restaurant and/or a museum close to the hotel are requested to be reserved. This suggests an idea of a channel that synchronizes a restaurant/museum and a hotel only if they are close to each other. The synchronization must be unidirectional: the hotel is reserved even if there are no restaurants around. The protocol for such a channel is shown in Figure 3-(a), where the `service1` represents the hotel service, and the `service2` represents the location-dependent service, e.g., a restaurant. The channel uses a `Map` service to calculate the distance between service locations. As a result, the `Map` service sets the channel `accept` function to `true` if `service2` is close, and to `false` otherwise. There is also a lossy channel from `service1` to the `accepted` state that is used if there is no close `service2`: it allows us not to block `service1`, e.g., hotel reservation, even if there are no restaurants nearby. The visual representation of the location synchronization channel is shown in Figure 3-(b). The provided example is intentionally simplified to illustrate how Reo may be used to build new domain-specific extensions: we assumed that a service can inform about its location, ignored possible operation semantic gaps, etc.

We now return to the example introduced in Section 2. In this example, Reo coordinates services represented by interfaces, but not actual service implementations. Since service discovery is out of the scope of this paper, we assume we have a smart service registry that provides us with proper service implementations whenever needed.

One of the possible Reo representations is provided in Figure 4. 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 synchro-

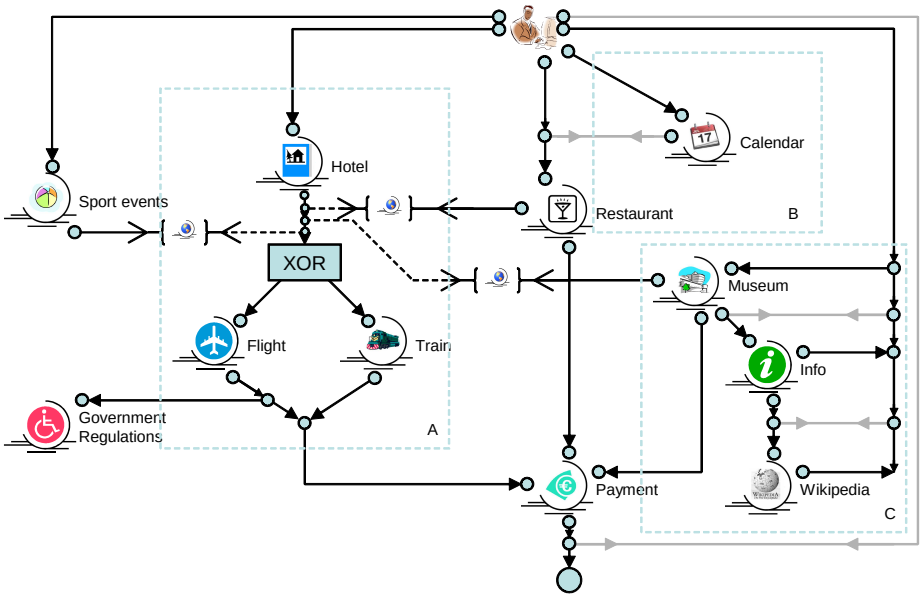

Fig. 4. A travel example in Reo

nization channels (between the hotel and the XOR (see Figure 2-(c)) element) since they all are connected by lossy channels. In Figure 4 the flight service is 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 3. 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

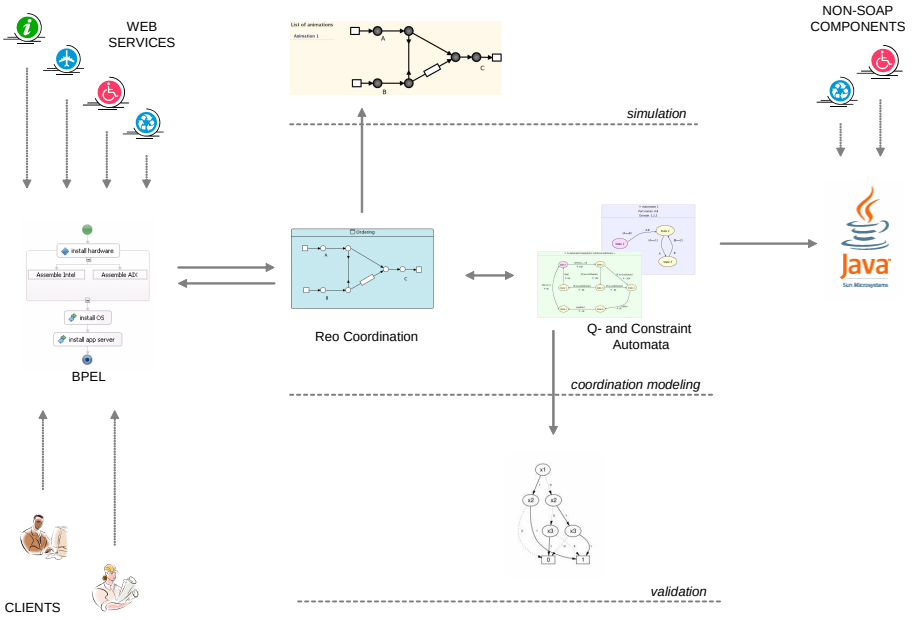

Fig. 5. Reo coordination framework for services.

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.

4 Implementation

In Section 3, we introduced our Reo framework for service coordination. Here we consider its implementation. The Reo coordination tool is developed to aid the process designers who is 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 over coordinations represented as constraint automata.

More detailed view of the Reo framework architecture is shown in Figure 5. The central part of the framework is a visual editor for Reo connectors. It represents the actual coordination model with services and communication channels. The developed tool also allows us to represent Reo in terms of constraint automata, an alternative behavioral model. This is useful if additional validation based on model checking techniques [10] is required. Q-Automata representation is used if QoS aspects of communication channels are important. Along with editing, Reo editor maintains simultaneous conversion to BPEL. To test the coordination model, one may run an animated simulation using the animation plug-in. In some situations it is desirable to use a designed coordination model in a non-web service scenario: in this case generation of java code is used. In the generated code distributed non-SOAP components are represented by java thread wrappers.

5 Conclusions and future work

In this paper we presented an approach to service coordination based on the exogenous coordination language Reo. This approach places the focus 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 the non-restrictive Reo requirements. As a result of compositional nature of Reo, we have convenient means for creating domain-specific language extensions. By that, the coordination language provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs by composing existing language elements, channels, and services.

In the paper we assumed that services support a simplified interaction model. While it is acceptable for simple information providers as map or calendar services, this assumption is not true in general. We plan to investigate the possibility of using Reo in complex scenarios with services having extended lifecycle support. Reo is perfect for defining new domain-specific language extensions. However, we lack specific extensions to the coordination language to support various issues important to services, e.g., temporal constraints, preferences, service extended descriptions. It is important to note, that coordination constraints introduced in this paper are not limited to user or business requirements from service providers: they may come as well from non-functional requirements, such as security, transactional consistency/compensation, etc. In future we plan to address specific non-functional requirements including QoS and service level agreements. As a separate line of research we plan to investigate the possibility to give better control to users over actual process execution, e.g., using preferences.

References

1. F. Arbab. Reo: a channel-based coordination model for component composition. *Math. Structures in CS*, 14(3):329–366, 2004.
2. Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. *Sci. Comput. Program.*, 55(1-3):3–52, 2005.
3. C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, 2006.
4. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *ICSOC-03*, 2003.
5. BPEL4WS. *Business Process Execution Language for Web Services*, May 2003.
6. F. Casati, M. Sayal, and M. Shan. Developing e-services for composing e-services. In *13th Int. Conf. on Advanced Information Systems Engineering (CAiSE)*, 2001.
7. L. Chen, N.R. Shadbolt, C. Goble, F. Tao, S.J. Cox, C. Puleston, and P. Smart. Towards a knowledge-based approach to semantic service composition. In *2nd Int. Semantic Web Conf. (ISWC2003)*, LNCS 2870. Springer, 2003.
8. D. Clarke, D. Costa, and F. Arbab. Connector colouring i: Synchronisation and context dependency. *Electr. Notes Theor. Comput. Sci.*, 154(1):101–119, 2006.
9. B. Grosof. Representing e-commerce rules via situated courteous logic programs in ruleml*1. *Electronic Commerce: Research and Applications*, 3(1):2–20, 2004.
10. S. Klueppelholz and C. Baier. Symbolic model checking for channel-based component connectors. In *FOCLASA'06*, 2006.
11. A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. *Journal on Digital Libraries*, 2005.
12. S. McIlraith and T. C. Son. Adapting Golog for composition of semantic web-services. In *Conf. on principles of Knowledge Representation (KR)*, 2002.
13. C. Nagl, F. Rosenberg, and S. Dustdar. Vidre - a distributed service-oriented business rule engine based on ruleml. In *EDOC'06*, 2006.
14. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Int. Semantic Web Conf. (ISWC)*, pages 333–347, 2002.
15. J. Rutten. Elements of stream calculus (an extensive exercise in coinduction). *Electr. Notes Theor. Comput. Sci.*, 45, 2001.
16. J. Scholten, F. Arbab, F. de Boer, and M. Bonsangue. A component coordination model based on mobile channels. *Fundam. Inform.*, 73(4):561–582, 2006.
17. E. Sirin, B. Parsia, and J. Hendler. Filtering and selecting semantic web services with interactive composition techniques. *IEEE Intelligent Systems*, 19(4), 2004.
18. S. Thakkar, J. Ambite, C. Knoblock, and C. Shahabi. Dynamically composing web services from on-line sources. In *AAAI Workshop Intelligent Service Integr.*, 2002.
19. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating daml-s web services composition using shop2. In *ISWC-03*, 2003.