Compositional Construction of Web Services Using Reo

Nikolay Diakov, Farhad Arbab

Dutch National Research Institute for Mathematics and Computer Science (CWI),
P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

Abstract. A Web Service can represent a unit of business logic that an organization exposes to other organizations on the World Wide Web. The recent efforts of the industry to agree on a common definition for Web Services resulted in the Web Services (WS) standard that governs how one defines, advertises and uses Web Services. Composition of primitive Web Services into complex ones presents the next challenge for the industry. Existing proposals for languages for service composition (also called choreography of Web services) typically come from the business process modeling community and often lack foundations in theoretical computer science and possibilities to address composition from a more general perspective than business process applications only. In this paper we present our work-in-progress on compositional construction of Web Services using the Reo coordination language. The Reo language has a strong formal basis and promotes loose coupling, distribution, mobility, exogenous coordination, and dynamic reconfigurability. We carry out this work within the context of the Cybernetics Incident Management (CIM) project.

1 Introduction

The main purpose of e-business consists of the automation of business processes using software applications. Service-oriented computing focuses on describing the externally observable behavior of such software applications. The Web Services standard applies the ideas of service-oriented computing to the Web. A Web Service represents a reusable piece of business logic that an organization can expose to other organizations via the World Wide Web. Fig. 1 shows the organization of the main ingredients of the WS standard [10] in a layered fashion.

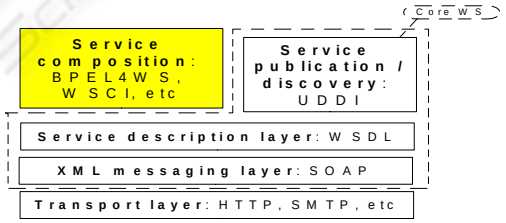

Fig. 1. WS layered architecture

Diakov N. and Arbab F. (2004).
Compositional Construction of Web Services Using Reo.
In *Proceedings of the 2nd International Workshop on Web Services: Modeling, Architecture and Infrastructure*, pages 49-58
DOI: 10.5220/0002666400490058

The service transport layer has the responsibility for transporting messages between a provider and a requestor of a Web Service. The XML messaging layer has the responsibility for the encoding of messages using a common XML format such as the SOAP protocol [14], so that both sides can have a common understanding of the structure of a message. The service description layer describes the public interface of a Web Service using the Web Services Description Language (WSDL) [15]. Service publication and discovery allows publishing and searching for Web Services using the Universal Description, Discovery and Integration (UDDI) [16]. The SOAP, WSDL, and UDDI specifications constitute the core of the WS standard. These specifications have reached a mature state wherein many major software vendors have committed to incorporating WS into the basic infrastructure of their products.

Automation of business processes across organizational boundaries has become a recent trend in e-business. This trend reflects the need for explicit modeling of long-running interactions and complex dependencies among different organizations. The WS composition layer (Fig. 1) facilitates this need by allowing one to build new Web Services out of simpler ones. At present, the industry has not agreed upon a common specification for service composition. Business Process Execution Language for Web Services (BPEL4WS) [17] and the Web Service Choreography Interface (WSCI) [18] constitute two examples of candidates for a service composition specification standard.

Current proposals for a Web Service composition standard suffer from two major types of problems [8]: (a) proposals often reflect some company interests in specific domain issues, which makes them inadequate for treating more general problems in service composition, and (b) proposals have operational semantic problems due to their lack of a strong formal basis. These two problems contribute to the insufficient expressive power of existing proposals in terms of a more theoretical perspective on component composition (one can generally view Web Services as components). Furthermore, it seems that current proposals for Web Service composition do not provide sufficient expressive power to support coordination of distributed activities, for example, as in transactions. We see this as the reason for the appearance of the WS-Coordination [13] as a standardization activity that tries to fill this gap separately from the composition of Web Services. The work described in this paper addresses the problems we mentioned so far altogether using the Reo coordination language.

2 The Reo Coordination Language

Reo presents a paradigm for composition of software components based on the notion of *mobile channels*. Reo enforces an exogenous channel-based coordination model that defines how designers can build complex coordinators, called *connectors*, out of simpler ones. Application designers can use Reo as a “glue code” language for compositional construction of connectors that orchestrate the cooperative behavior of component instances in a component-based system. One can find an introduction to Reo in [1], and a detailed elaboration on the language and its model in [2]. The Reo coordination language provides, among others, the following features:

- Loose coupling among components;
- Support for distribution and mobility of heterogeneous components;
- Exogenous coordination (i.e., by third parties);
- Dynamic reconfigurability;
- Formal semantics based on a coinductive calculus of flow [3] and (alternatively) on constraint automata [4].

5.2 Basic concepts

Reo does not say much about the application components, whose activities it coordinates. From the point of view of Reo, a system consists of a number of *component instances*, communicating through connectors that coordinate their activities. Reo assumes that a component instance contains one or more active entities (e.g., processes, agents, threads, actors, etc.), which communicate with entities outside of their component instance only through input/output operations that they perform on a (dynamic) set of channel ends connected to the component instance. Reo completely abstracts from the details of the communication within a component instance. Instead, Reo focuses on the inter-component-instance communication, which takes place exclusively through connectors.

Reo allows compositional construction of a connector out of simpler connectors, where channels represent the atomic connectors. A channel has precisely two channel ends. Reo introduces two types of channel ends: *sink* and *source*. A sink dispenses data out of its channel. A source accepts data into its channel.

Reo models a connector as a graph of nodes and edges, where zero or more channel ends may coincide on every node, every channel end coincides on exactly one node, and an edge exists between two nodes if and only if there exists a channel whose channel ends coincide on those nodes.

When a component instance *knows* a channel end, any active entity inside it can perform *Reo operations* on that channel end.

5.2 Reo operations

Reo defines the following operations, which relate to the manipulation of the connector topology: *create*, *forget*, *join*, *split*, and *hide*. The *create* operation creates a channel (of some defined type) and makes the channel ends available to the performer of the operation. With the *forget* operation a component instance tells Reo that it does not need a channel end anymore. The *join* operation allows joining of two nodes identified by two channel ends, each coincident with one of the nodes. The *split* operation allows for splitting a node into two nodes by specifying the channel ends that the performer requires to coincide on the new nodes. The *hide* operation allows the performer to protect the topology of a node designated by some coincident channel end, making subsequent *join* and *split* fail on channel ends coincident with that node.

Reo defines the following operations, which allow input/output of data: *connect*, *disconnect*, *wait*, *read*, *take*, *write*, and *move*. The *connect* operation connects the performer to a channel end by providing exclusive access to the node (and thus all

channel ends coincident with it) on which this channel end coincides. The *disconnect* operation releases a previously established connection. The *wait* operation allows the performer to wait for some condition on a channel end. The *read* operation allows the performer to non-destructively read data from a sink. The *take* operation does the same as read but it also removes the data from the sink. The *write* operation allows the performer to write data to a source. The *move* operation allows the performer to move a channel end to another location. Note that changing location does not change the topology of the connector or the connection status of the moved channel end.

5.3 A useful set of primitive channels

Reo assumes the availability of an arbitrary set of channel types, each with well-defined behavior. As an example, we present the following non-exhaustive set of channel types, each with some distinctive properties: *Sync*, *Filter*, *SyncDrain*, *SyncSpout*, *LossySync*. A Sync channel has a source and a sink. Writing a message succeeds on the *source* of a Sync channel if and only if taking of a message succeeds at the same time on its *sink*. The Filter channel behaves like the Sync except that it loses all data that do not match the specified pattern of the Filter. A SyncDrain has two sources. Writing a message succeeds on one of the sources of a SyncDrain channel if and only if writing a message succeeds on the other source. A SyncSpout has two sinks. A SyncSpout channel serves as an unbounded source of data that matches certain pattern on both of its sinks. A LossySync channel has a sink and a source. The source always accepts all data items. If the sink does not have a pending read or take operation, the LossySync loses the data item, otherwise the channel behaves as a Sync.

3 Composing Web Services Using Reo

Consider an example in which some organization wants to offer a “Holiday Reservation Service” (HRS) that allows customers to organize holiday travels (Fig.2).

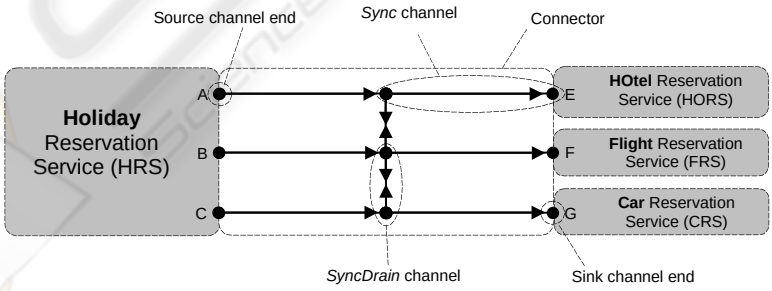

Fig. 2. Committing a transaction modeled with Reo

For simplicity, we assume that organizing a holiday requires making reservations for a hotel, for a flight and for a car. Some other organizations offer services each doing part of the job: hence the HORS, FRS and CRS services. Again for simplicity, we assume that the HRS service has already negotiated with a client a selection of proper flight, hotel and car. As the last thing before asking the customer to pay, the HRS service needs to "commit" a transaction containing each of the reservations. Now, a holiday reservation should only succeed when all other three reservations succeed.

We model the "commit" part of the behavior of the HRS service using a "*barrier synchronization*" connector in Reo consisting of six synchronous channels and two synchronous drain channels, organized together as in Fig.2.

The HRS service makes "commit" requests on channel ends A, B, and C in any order and at times it needs to. According to the semantics of the barrier synchronization connector, the three "commits" of the HRS system will succeed (the success condition for the holiday reservation) if and only if the three reservations at the HORS, FRS and CRS services, respectively, succeed at the same time.

This example illustrates the ease with which Reo allows construction of connectors that exhibit complex behavior such as transaction support. The simple composition rules of Reo yield a surprisingly expressive power that enables the construction of different types of communication infrastructures, such as peer-to-peer, shared data space, software bus, etc., out of a very small set of primitive channel types [2], [11].

4 Reo Coordination Middleware

In order to use Reo for efficient construction of software applications such as Web Services, we need to build a Reo coordination middleware.

A *middleware* coordinates the interactions among application components by providing functionality that bridges the gaps between software applications and the low-level hardware and software [5]. A *coordination middleware* consists of system software that provides a set of reusable common services and programming mechanisms for coordination of component behavior. Coordination middleware simplifies the development of distributed software applications by offering high-level abstractions for programming of component compositions, conceptually closer to an application model defined using a coordination language, than the low-level programming methods offered by most existing component middleware technologies.

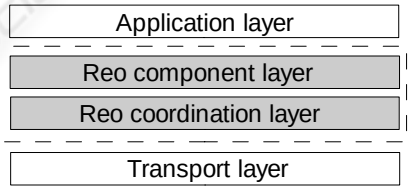

Fig. 3. Reo coordination middleware

We model a Reo coordination middleware as two separate layers between the transport layer and the application layer (Fig. 3). The Reo coordination layer provides the basic Reo functionality, such as basic channels and operations on their channel ends. The Reo component layer provides the minimum services and tools necessary for building, deploying and executing application components with Reo.

5 Reo-enabled Web Services

We consider Web Services as a class of software applications. In order to facilitate this class with a Reo coordination middleware, we need to provide an additional adaptation layer – the Reo WS layer (Fig.4). We do this in a layered fashion, because we wish to allow the mutual coexistence of both Web Services and "pure" (non-Web Services) Reo components.

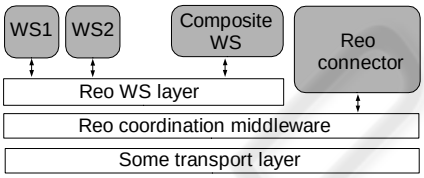

Fig. 4. Layered representation of a composite Web Service

The Reo WS layer effectively allows making a Reo component out of any Web Service. This way Web Services can interact with Reo components and vice versa. Further, we consider two ways to construct the Reo WS layer: (a) as an alternative transport layer for the Web Services technology, and (b) as a collection of adapters. We view these two approaches as complementary, because each has its own pros and cons.

5.1 Reo Transport for Web Services

The WS standard separates the definition of messages exchanged with a Web Service from the way the distributed environment communicates these messages to and from a Web Service. We use this feature of the WS standard to define a Reo Transport For Web Services – RT4WS (Fig. 5).

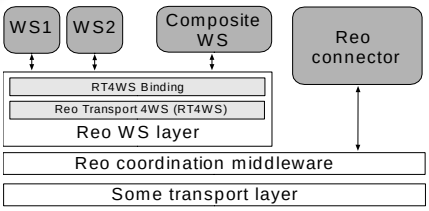

Fig. 5. Reo transport for Web Services

WS providers who choose the Reo coordination middleware need to select the RT4WS for use with their Web Service. In order to do this in a generic way, the provider needs to use a special binding for the Reo transport. The RT4WS binding defines how a provider sends messages with a particular transport (i.e., allows the provider to *bind* the service to the transport).

A Reo-enabled transport (potentially) allows for an efficient and transparent integration of the Reo middleware with an arbitrary existing Web Service. We regard the fact that the provider must actually install the new transport in his infrastructure in order to enable his Web Services for Reo as the main drawback of this approach.

5.2 WS Adapters for Reo

In the second approach we define two types of adapters (Fig. 6): Reo WS provider adapters and Reo WS requestor adapters.

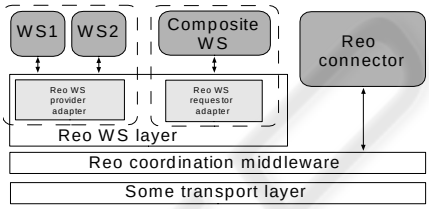

Fig. 6. Reo adapters for Web Services

The Reo WS provider adapter mediates all communication between the Reo coordination middleware and a Web Service. Effectively, the Reo WS provider adapter appears to a Web Service as a typical WS requestor using one of the transports originally supported by the Web Service to make requests to the Web Service. On the other hand, the Reo WS provider adapter appears as a special Reo component that offers the functionality of the Web Service to other Reo-enabled Web Services or "pure" Reo components. In a similar way, the Reo WS requestor can connect to a Reo WS provider adapter, and appears to a typical WS requestor as a normal Web Service. This allows a requestor (e.g., a composite Web Service) to use other Reo-enabled Web Services or "pure" Reo-components as if they behaved as "normal" (non-Reo-enabled) Web Services. Effectively, both adapters play the role of additional tiers between a service provider and a service requestor that enable their interaction with a Reo-enabled infrastructure provided by the Reo coordination middleware.

Providers of Web Services need not do anything to allow their Web Services to interact with Reo components, e.g., for the purpose of participating in a composition with other Web Services. We consider this as the main advantage of the adapter approach. In fact, the WS provider need not know that someone "adapts" its Web Services for use with the Reo infrastructure. We view possible performance degradation, if any, resulting from the multiple transport protocols used in every adapter, as a disadvantage of this approach. This problem often appears in multi-tier system.

6 Coordination Middleware based on MoCha

In this section we describe our high-level architecture of a coordination middleware using the MoCha middleware. This architecture uses Web Services as a Reo component model by following the transport approach we discussed earlier.

MoCha stands for **Mo**bile **Cha**nnels middleware [7], [9]. A mobile channels middleware provides channel communication with dynamic reconfiguration capabilities. Mobile channels provide to components transparent support for mobility within a distributed environment.

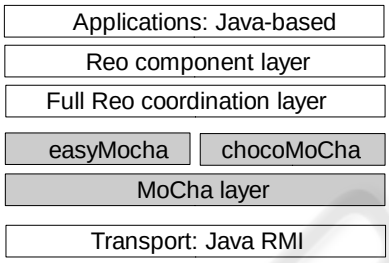

Fig. 7. The MoCha middleware

The MoCha middleware (Fig. 7) implements a subset of the Reo functionality, limited to direct component-to-component connections. The MoCha middleware constitutes our first step toward a full Reo coordination middleware implementation. MoCha uses Java RMI as a distributed processing environment. MoCha has three basic components: the MoCha layer, easyMocha layer, and chocoMocha layer. The MoCha layer provides the basic channel communication. The easyMoCha layer provides a more extensive and user friendly interface together with the ability of keeping the consistency of channel-end references for easier development of applications on top of the MoCha layer. The chocoMoCha layer provides operations that allow components to connect to and disconnect from MoCha channel ends. In order to become fully Reo compliant, MoCha needs an additional layer that adds the concept of *nodes* and operations such as *join* and *split*, among others, on Reo nodes. The MoCha layer does not provide a component model, therefore we need an additional Reo component layer to use MoCha in component-based applications. The work presented in [7] describes a first version of such a component layer.

We use the MoCha prototype to build a Reo transport for Web Services. Fig. 8 shows the resulting coordination middleware architecture for WebServices.

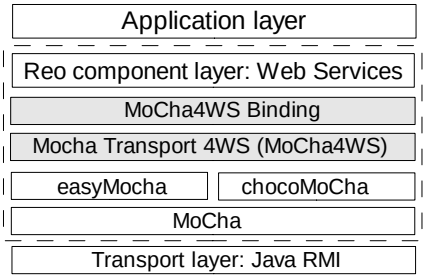

Fig. 8. Coordination middleware architecture for Web Services based on MoCha

The MoCha, easyMocha, and chocoMoCha layers constitute the bottom layer of the coordination middleware. In order to integrate MoCha with Web Services we make a MoCha4WS transport layer and a MoCha4WS Binding layer that allows the use of this new transport. This effectively makes Web Services appear as a Reo component layer, which designers can use to build applications imbued with the features that Reo provides.

7 Conclusions

In this paper we presented our ongoing work on compositional construction of Web Services. We use a coordination language, Reo, that promotes loose coupling, distribution, mobility, exogenous coordination, and dynamic reconfiguration. We propose Reo as a modeling language in which designers implement Web Service compositions. The strong formal basis of Reo guarantees possibilities for both model checking and verification, as well as well-defined execution semantics of a Web Service composition. As such, we consider Reo complementary to the existing languages (typically coming from the business process modeling community) for service composition.

We aim at producing a prototype coordination middleware that allows designers to build robust and reconfigurable composite Web Services.

8 Acknowledgements

The authors would like to thank Juan Guillen-Scholten for his help on the MoCha middleware. We carry out this work within the context of the Cybernetics Incident Management (CIM) project, SENTER project nr. TSIT2021, The Netherlands.

References

1. Arbab, F., Mavadatt, F. “Coordination through channel composition”. In “Coordination Languages and Models: Proc. Coordination 2002, volume 2315 of Lecture Notes in Computer Science, Springer-Verlag, pp. 21-38.
2. Arbab, F. “Reo: A Channel-based Coordination Model for Component Composition”. To appear in “Mathematical Structures in Computer Science”, 2004.
3. Arbab, F., Rutten, J.J.M.M. “A Coinductive Calculus of Component Connectors”. In the Proceedings of 16th International Workshop on Algebraic Development Techniques (WADT 2002), Lecture Notes in Computer Science 2755, Springer, 2003, pp. 35--56.
4. Arbab, F., Baier, C., Rutten, J., Sirjani, M. “Modeling Component Connectors in Reo by Constraint Automata”. To appear in Electronic Notes in Theoretical Computer Science, 2004.
5. Schantz, R. E., and Schmidt, D. C. “Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications,” Encyclopedia of Software Eng., Wiley & Sons, New York, 2001; also available at http://www.cs.wustl.edu/~schmidt/PDF/middleware-chapter.pdf.
7. Guillen-Scholten, J.V., Arbab, F., de Boer, F.S., and Bonsangue, M.M. “A Channel-based Coordination Model for Components”, in the Proceedings of 1st International Workshop on Foundations of Coordination Languages and Software Architectures, ENTCS 68.3, Elsevier Science, 2002.
8. Van der Aalst, W.M.P. “Don't go with the flow: Web services composition standards exposed”. Trends and Controversies, IEEE Intelligent Systems, issue Jan/Feb, 2003.
9. Arbab, F., F., de Boer, F.S., Guillen-Scholten, G.V., Bonsangue, M.M. “MoCha: A Middleware Based on Mobile Channels”. In the Proceedings of the 26th Annual International Computer Science Software and Applications Conference (COMPSAC'02).
10. Web Services site. http://www.w3c.org/2002/ws/
11. Arbab, F. “Abstract Behavior Types: A Foundation Model for Components and Their Composition”. In the Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002), LNCS 2852, pp.33-70, 2003, The Netherlands.
12. The CIM project official web page. http://www.almende.com/cim.
13. F. Cabrera et al., “Web Services Coordination (WS-Coordination)”, August 2002, http://www.ibm.com/developerworks/library/ws-coor/
14. XML Protocol Group, “SOAP 1.2”, W3C Recommendation, June 2003, http://www.w3c.org/2000/xp/Group/
15. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, “Web Services Description Language (WSDL) 2.0”, W3C, November 2001, www.w3.org/TR/wsdl20/
16. Universal Description, Discovery and Integration (UDDI) protocol 3.0, November 2004, http://www.uddi.org/
17. F. Curbera, Y. Goland, J. Klein, F. Leyman, D. Roller, S. Thatte, and S. Weerawarana, “Business Process Execution Language for Web Services (BPEL4WS) 1.1,” May 2003, http://www.ibm.com/developerworks/library/ws-bpel/
18. Web Services Choreography Interface 1.0 specification, December, 2003, http://wwws.sun.com/software/xml/developers/wsci/.