Automatic Code Generation for the Orchestration of Web Services with Reo

Sung-Shik T.Q. Jongmans[1], Francesco Santini[1], Mahdi Sargolzaei[2], Farhad Arbab[1], and Hamideh Afsarmanesh[2]

[1] Centrum Wiskunde & Informatica, Amsterdam, Netherlands
{S.S.T.Q.Jongmans,F.Santini,Farhad.Arbab}@cwi.nl
[2] Universiteit van Amsterdam, Amsterdam, Netherlands
{H.Afsarmanesh,M.Sargolzaei}@uva.nl

Abstract. We present a compositional construction of Web Services, using Reo and Constraint Automata as the main "glue" ingredients. Reo is a graphical and exogenous coordination language based on channels. We propose a framework that, taking as input the behavioral description of services (as Constraint Automata), their WSDL interfaces, and the description of their interaction in Reo, generates all the necessary Java code to orchestrate the services in practice. For each Web Service, we automatically generate a proxy that manages the communication between this service and the Reo circuit. Although we focus on Web Services, we can compose different kinds of service-oriented and component technologies at the same time (e.g., CORBA, RPC, WCF), by generating different proxies and connecting them to the same coordinator.

1 Introduction and Motivations

A *Web Service* (WS) can be very generally described as a software system designed to support interoperable machine-to-machine interaction over a network. The standards at the basis of WSs are the *Web Services Description Language* (WSDL) [18], which describes the interface in a machine-processable format, and *Simple Object Access Protocol* (SOAP) [17], which is used to format the exchanged messages, typically conveyed using HTTP with an XML serialization.

Web Services are strongly loosely-coupled by definition, and therefore, two fundamental combination paradigms have emerged in the literature, permitting complex combinations of WSs: *orchestration* and *choreography* [16]. Nowadays, there exist many workflow-description-based languages, defined to orchestrate or to choreograph WSs, including BPEL4WS [15] and WS-CDL [20] (see Sec. 2). However, such proposals remain at the description level, without providing any kind of formal reasoning mechanisms or tool support based on the proposed notation for checking the compatibility of WSs [11]. Despite all the efforts, composition of WSs is still a critical problem.

In this paper, we orchestrate WSs using the graphical language Reo [1]. Several (rather theoretical) studies on service orchestration using Reo already exist, including [11,12,13]. We build atop ideas presented in those papers, approaching

F. De Paoli, E. Pimentel, and G. Zavattaro (Eds.): ESOCC 2012, LNCS 7592, pp. 1–16, 2012.
© Springer-Verlag Berlin Heidelberg 2012

them from a more practical perspective: we present a tool that enables employing Reo for orchestrating *real* WSs, deployed and running on different servers.

The Reo language has a strong formal basis and promotes loose coupling, distribution, mobility, exogenous coordination, and dynamic reconfigurability. *Constraint Automata* (CA) [2] provide compositional formal semantics for Reo. The formal basis of Reo guarantees possibilities for both model checking and verification [10], as well as well-defined execution semantics of a Web Service composition [11]. Exogenous coordination of components in Reo by channels makes it suitable for modeling orchestration. In this modeling, WSs play the role of components and the orchestrator is the Reo circuit that coordinates them. In other words, Reo as a modeling language for service composition can provide service connectivity, composition correctness, automatic composition, and composition scalability, which are vital and valuable for modeling WSs.

In the rest of the paper, we present how to generate all the necessary Java code in an automated way, starting from the description of the orchestration (given as a Reo circuit), the description of the WSs interfaces (given as WSDL files), and the description of the WSs behavior (given as automata). For each WS, we generate a proxy application that acts as an intermediary relaying messages between its WS and the Reo orchestrator, i.e., between the "WS world" and the "Reo world." All the output code necessary to manage the orchestration in practice is generated automatically, in a manner completely transparent to both client and WSs developers, whose programmers do not have to be concerned with this middleware at all. Although we focus on WSs in this paper, the same framework can be used to compose different kinds of service-oriented and component technologies at the same time (e.g., CORBA, RPC, WCF), by generating different proxies and connecting them to the same Reo circuit. Therefore, we make Reo a complete language for the verification and (with this work) implementation of WS orchestration.

The paper is organized as follows: in Sec. 2 we describe the related work and further motivate this paper with respect to the literature. In Sec. 3, we summarize the necessary background notions about Reo. Section 4 forms the core of the paper, since it details the architecture of our Reo-based orchestration platform and how we implemented it. In Sec. 5 we present two case studies of WS combination that can be automatically generated with our tool, and in Sec. 6 we draw the final conclusions and describe our future work.

2 Related Work

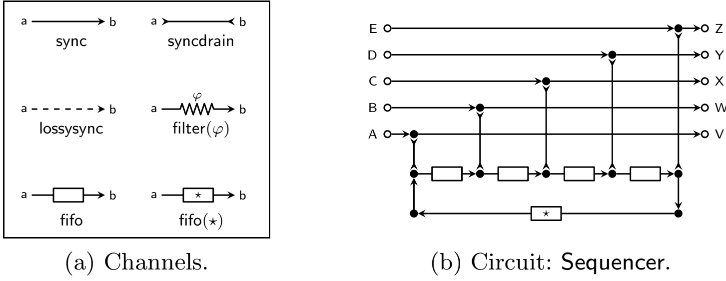
In literature we can find two main coordination paradigms to combine Web Services (WS): either through *orchestration* or *choreography* [16] languages. In orchestration, the involved WSs are under the control of a single endpoint central process. This process coordinates the execution of different operations on the WSs participating in the process. An invoked WS neither knows nor needs to know that it is involved in a composition process and that it is playing a role in a business process definition. Choreography, in contrast, does not depend on a central

orchestrator. Each WS that participates in the choreography has to know exactly when to become active and with whom to interoperate: they must be conscious of the business process, operations to execute, messages to exchange, as well as the timing of message exchanges. However, in real-world scenarios, corporate entities are sometimes unwilling to delegate control of their business processes to their integration partners. Therefore, in this paper we focus on the orchestration paradigm, although Reo can be used to describe choreographies [11] as well.

Many languages have emerged and been proposed in academia and industry for composition and execution of Web Services, according to the choreography or orchestration principles: some examples are BPEL4WS [15], WS-CDL [20], BPML and WSCI [19], and BPMN [6] and BPEL4CHOR [6]. The *Business Process Execution Language for Web Services* (BPEL4WS) is an orchestration language with an XML-based syntax, supporting specification of processes that involve operations provided by one or several WSs. Furthermore, BPEL4WS draws upon concepts developed in the area of workflow management. When compared to languages supported by existing workflow systems and to related standards (for example, WSCI), it relatively appears to be more expressive. BPEL4CHOR is a choreography-oriented version of WS-BPEL instead. The *W3C Web Service Choreography Description Languages* (WS-CDL) is a W3C candidate recommendation in the area of service composition. Like WSCI, the intent of WS-CDL is to define a language for describing multiparty interaction scenarios (or choreographies), not necessarily for the purpose of executing them using a central scheduler but rather with the purpose of monitoring them and being able to detect deviations with respect to a given specification. The *Business Process Modeling Notation* (BPMN) offers a rich set of graphical notations for control flow constructs and includes the notion of interacting processes where sequence flow (within an organization) and message flow (between organizations) are distinguished [6]. Several formal proposals have been made for representing WSs using, for example, Labeled Transition System, Process Algebra, Petri nets, and Reo itself [7,3,21,11].

Considering the existing implementations, we can find service-oriented workflow research tools, as *BliteC* [4] and *JOLIE* [14], and commercial offers, as *IBM WebSphere*, *BEA WebLogic Integrator*, *Microsoft Web Services Support*, and *WF*. These systems provide a design tool and an execution engine for business processes in workflow specification languages. For example part of the *BizTalk* suite (another Microsoft product) is the *BizTalk Orchestration Engine*, which implements XLANG (a precursor of BPEL4WS). *Windows Workflow Foundation* (*WF*) is a Microsoft technology that provides an API, an in-process workflow engine, and a rehostable designer to implement long-running processes as workflows within .NET applications. *BliteC* [4] is a software tool that translates service orchestrations written in *Blite*, into readily executable WS-BPEL programs. *JOLIE* [14] is a Java-based interpreter and engine for orchestration programs, with a mathematical underlying model.

Comparing our solution with the related work presented in this section, none of the XML-based languages in the proposed standards, e.g., BPEL4WS [15] or WS-CDL [20], comes with tools for a direct formal verification and model checking



(a) Channels. (b) Circuit: Sequencer.

Fig. 1. Graphical syntax of common channels and a circuit

Name	Behavior
sync	Atomically fetches an item on its source end a and dispatches it on its sink end b.
syncdrain	Atomically fetches (and loses) items on both of its source ends a and b.
lossysync	Atomically fetches an item on its source end a and, non-deterministically, either dispatches it on its sink end b or loses it.
filter(φ)	Atomically fetches an item on its source end a and dispatches it on its sink end b if this item satisfies the filter constraint φ; loses the item otherwise.
fifo	Atomically fetches an item on its source end a and stores it in its buffer.
fifo($\star$)	Atomically dispatches the item $\star$ on its sink end b and clears its buffer.

Fig. 2. Channel behavior

of programs or specifications in that language; therefore, verification of specifications in these languages requires a translation to a higher level of abstraction, in contrast to other formal techniques, such as Process Algebra [3] and Petri nets [21]. Moreover, with Reo a user is able to compose two orchestrators such that global synchronicity emerges from the synchronous behavior of the individual orchestrators [1]. This can be useful when different coordination protocols, designed for different services, need to be merged together in order to integrate all of them in the same single protocol. This advantage is granted by the formal definition of the *join* operator on two circuits [1]. Furthermore, the Reo language yields more declarative to directly specify an interaction, while with Process Algebra one has to define a sequence of actions to achieve the same interaction.

3 Reo

As its main feature, Reo facilitates compositional construction of *circuits*: communication mediums that coordinate interacting parties (in this paper, Web Services), each built from a number of simple *channels*. Every channel in Reo

has exactly two *ends*, and each such end has exactly one of two types: a channel end either accepts data items—a *source end*—or it offers data items—a *sink end*.[1] Figure 1a shows six different channels at the disposal of Reo users; Figure 2 describes their behavior. Interestingly, Reo does not fix which particular channels one may use to construct circuits with. Instead, Reo supports an *open-ended set of channels*, each of which exhibits a unique behavior. This feature enables users of Reo to define their own channels, tailored to their specific needs.

We call the act of "gluing" channel ends together to build circuits *composition*. One can think of composite circuits as digraphs with nodes and edges (channels) and compare their behavior to plumbing systems. In such systems, "fluids" flow through "pipes and tubes" past "fittings and valves." Similarly, in Reo circuits, "data items" flow through "channels" (along edges) past "nodes." Usually, the interacting parties themselves supply the data items that flow through the circuits they communicate through. To this end, every circuit defines an interface. Such an interface consists of the *boundary nodes* of a circuit: parties write and take data items only to and from boundary nodes.

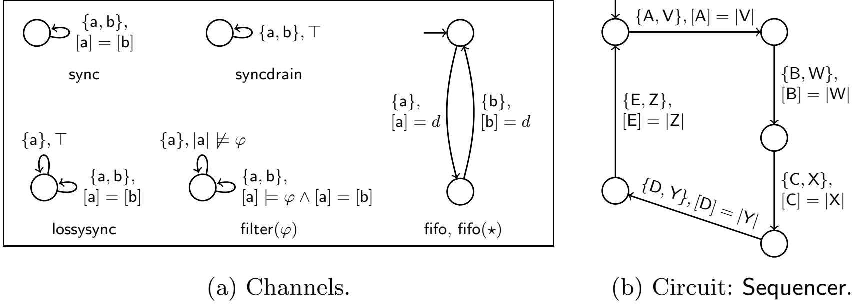
Figure 1b shows a circuit, named Sequencer, that one can construct from the channels in Fig. 1a. This circuit imposes an order on when parties can write and take data items to and from its boundary nodes (shown as open circles): first A and V, second B and W, ... , fifth E and Z, subsequently A and V again, etc. In general, one derives the behavior of a circuit from the behavior of the channels and nodes that it consists of—circuits exhibit *compositionality*. We skip the details here, however, for brevity and because they do not matter for the rest of this paper—see [1] for details.

Importantly, there exist various *semantic models* to formally describe the behavior of circuits. These semantic models,[2] among other applications, enable one to reason about the correctness of service orchestrations. For example, Kokash et al. employ the `mCRL2` toolkit to verify the correctness of Reo translations of business process models [10]. In this paper, however, we use formal models of circuits for two other purposes. First, we employ *Constraint Automata* (CA) [2] to automatically compile Reo circuits—i.e., orchestrators—to Java code. Second, we formalize the behavior of WSs in terms of CA (see Sec. 4).

Constraint Automata resemble classical finite state machines in the sense that they consist of finite sets of states and transitions. States represent the internal configurations of a circuit, while transitions describe its atomic coordination steps. Formally, we represent a transition as a tuple of four elements: a source state, a *synchronization constraint*, a *data constraint*, and a target state. A synchronization constraint specifies which nodes synchronize—i.e., through which nodes a data item flows—in some coordination step; a data constraint specifies which particular data items flow in such a step. Figure 3 shows the CA of the channels and circuits in Fig. 1.

[1] However, channels do not necessarily have *both* a source end *and* a sink end: they can also have two source ends or two sink ends.

[2] See [9] for a recent survey on the various semantic formalisms for Reo.



(a) Channels. (b) Circuit: Sequencer.

Fig. 3. Constraint Automata of common channels and a circuit

4 Orchestrating Web Services with Reo

Conceptually, orchestrating Web Services (WS) using Reo proceeds in three steps: (i) design an orchestrator circuit, (ii) deploy and run this circuit, and (iii) connect some WSs to it. The *Extensible Coordination Tools* (ECT),[3] a collection of Eclipse plug-ins constituting the default IDE for Reo, perfectly supports step (i): it allows users of Reo to design circuits using a drag-and-drop interface. But unfortunately, steps (ii) and (iii) involve less straightforward activities. How can we go from a circuit diagram to executable code? And how can we connect WSs oblivious to Reo to this executable circuit? In this section, we present two tools that address these questions. We call these tools the *Reo Compiler* and the *Proxy Generator*:

- The Reo Compiler compiles circuit diagrams to Java, addressing step (ii).
- The Proxy Generator generates *proxies* for WSs. Postponing the details until Sec. 4.2, a proxy serves as an intermediary between a circuit and a WS. Essentially, it relays data items from a circuit to a WS and vice versa, bridging the gap between them, addressing step (iii).

Figure 4 shows the architecture of our two tools and the intended workflow for using them. We elaborate on this figure in the next three subsections.

4.1 Reo Compiler: From Circuit Diagrams to Java

The Reo Compiler works as follows. Suppose a user of Reo has drawn a circuit diagram using the ECT and wishes to compile it to Java. Internally, the ECT stores this diagram as an XML document, which subsequently serves as input to the Reo Compiler; the box labeled "Reo Circuit" in Fig. 4 represents such an XML document. On input of an XML document `conn.xml` describing some circuit C, the Reo Compiler first parses this file; the component labeled "Reo Parser" represents the component involved. Subsequently, the Reo Compiler computes

[3] `http://reo.project.cwi.nl`

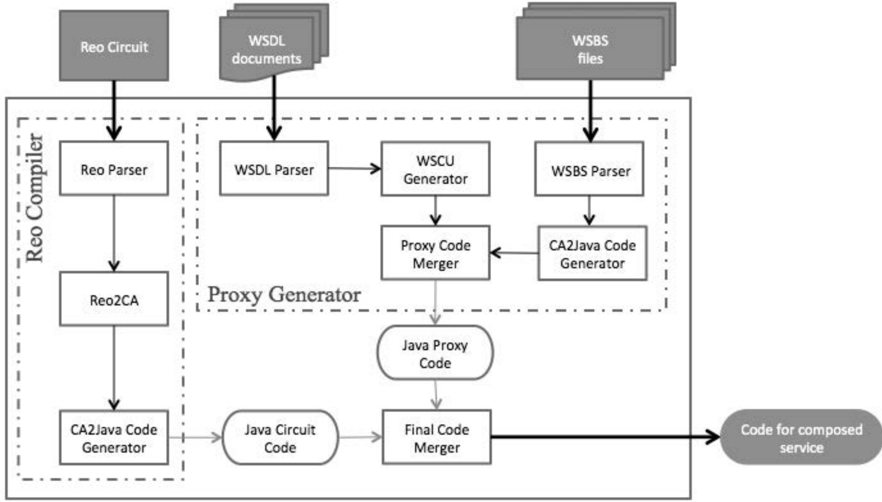


Fig. 4. Architecture of our code generation framework

the Constraint Automaton (CA) that models the behavior of C. For this purpose, it uses functionality that the ECT already ships with; the box labeled "Reo2CA" in Fig. 4 represents the component involved. This computation has a complexity exponential in the number of buffers—fifo channels—in C. Finally, based on the CA just computed, the Reo Compiler generates a Java class; the boxes labeled "CA2Java" and "Java Circuit Code" in Fig. 4, respectively, represent the component involved and the generated class. This computation has a complexity linear in the size of the CA. The generated class extends the `Thread` class, overriding the default `run()` method. In particular, `run()` now executes a state machine that simulates the CA computed previously, as follows.

Suppose a Java class `Conn` generated as described above for some circuit C. At runtime, an instance `conn` of `Conn` awaits requests for writing or taking data items at particular nodes. We call the concurrent data structures that register such requests *synchronization points*: for each node that a circuit allows interaction on—its boundary nodes—we have a synchronization point in the implementation. To make a transition, `conn` first checks the synchronization points of the nodes involved in that transition, i.e., the synchronization constraint in the corresponding transition of the CA of C. This check has a complexity linear in the size—the number of nodes—of the synchronization constraint. If appropriate requests exist, `conn` invokes a simple solver to check if the data items involved in those requests satisfy certain conditions, i.e., the data constraint in the corresponding transition of the CA of C. If they do, the transition fires: the requests resolve and data items flow according to the conditions checked before.

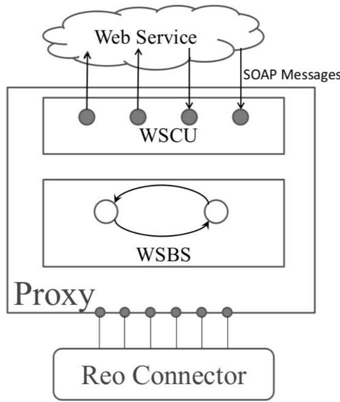


Fig. 5. Architecture of a proxy

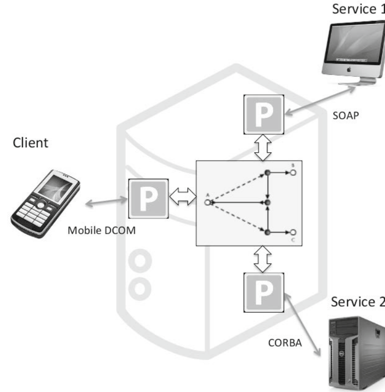


Fig. 6. Architecture of a deployed orchestration

Importantly, interacting parties can communicate with compiled circuits at runtime only through synchronization points. As detailed in Section 4.2, this causes issues if we want to connect WSs to a circuit compiled as described.

4.2 Proxy Generator: Connecting Web Services to Circuits

First, we discuss in more detail why we need proxies and how they work. Thereafter, we describe the process of generating them.

Background on Proxies: Suppose a WS deployed remotely—from the viewpoint of the machine on which the orchestrator runs—that we wish to include in a service composition orchestrated by a circuit. Ideally, we would simply send this WS the synchronization points of the boundary nodes it should connect to. Subsequently, this WS would perform I/O operations directly on the synchronization points it received. Unfortunately, however, this approach does not work: no WS supports direct communication through circuits—to adopt this as a prerequisite for service orchestration with Reo would constitute an unacceptable limitation. Instead, we take a different approach for including WSs in service compositions orchestrated with Reo (including existing ones): *service proxying.*

In service proxying, one creates a proxy for each WS in a service composition. These proxies then serve as intermediaries between a circuit and the orchestrated WSs.[4] To a WS, however, its proxy looks just like any other client. To explain service proxying in more detail, Fig. 5 shows the architecture of a proxy P, together with a circuit C and a WS S. Essentially, P consists of two *sides*:

[4] Note that the concept of service proxies works also for other orchestration languages.

a *circuit side* and a *service side*. On the circuit side, P has access to a number of synchronization points. Thus, this side allows P to write and take data items directly to and from C. On the service side, P has access to the network infrastructure that connects P with S. Thus, this side allows P to directly send and receive messages to and from S. Essentially, P has two tasks:

- Take data items from C on its circuit side; encode these data items into messages that it can send to S on its service side; send these messages.
- Receive messages from S on its service side; encode these messages into data items that it can write to C on its circuit side; write these data items.

Circuit Side. The circuit side of a proxy contains synchronization points connected to a Java implementation of a Constraint Automaton (CA). This CA implementation resembles the implementation discussed in Sec. 4.1. The box labeled "Simulation Automaton" in Fig. 5 represents this executable CA, which *simulates* the behavior of the WS involved. In particular, it simulates the *external view* [11] of this WS: every state represents an externally observable internal configuration of the WS, while every transition represents the exchange of one or more messages.[5] More precisely, the synchronization points that the simulation automaton inside a proxy has access to represent the *names* of the messages exchanged by a WS; the data items passing through these synchronization points constitute the actual payloads. (Recall from Section 4.1 that a CA implementation has a synchronization point for each boundary node appearing in one of its transitions.) For instance, one may have a synchronization point for a message named `addIntegersRequest` and a (serialized) data item "`x=1;y=2`" (the actual payload of `addIntegersRequest`).

Proxies require simulation automata for the following reason. First, we observe that *stateful* WSs—WSs with more than one internal configuration—may permit the exchange of different messages in different such configurations. The correct functioning of proxies depends crucially on this information: proxies must know which messages their WSs can exchange in each state to decide which synchronization points to allow interaction on. To illustrate this, suppose a WS that at some point permits only the receipt of a `multIntegersRequest` message. In that case, it makes no sense for its proxy to take data items from the synchronization point for the `addIntegersRequest` message: the proxy may try to relay data items taken from this synchronization point, but it will certainly fail. After all, the WS does not permit `addIntegersRequest` messages in its current state! Proxies must know the present configuration of their WSs to avoid such faulty behavior. However, WSs encapsulate their states and generally, they do not provide means to share them with clients. By simulating their respective WS behavior, proxies compensate for this: every time a proxy exchanges a message with its WS, its simulation automaton makes a corresponding transition. In this way,

[5] We model stateless WSs with singleton automata.

a proxy can always derive which message exchanges its WS permits, namely from its simulation automaton.[6]

Service Side. The service side of a proxy contains components that facilitate network communication, e.g., using SOAP: a standardized protocol for exchanging structured information in computer networks [17]. To implement this, we use Apache Axis2 [8], because it is a very flexible and easily extensible framework. Moreover, it supports many standards, including WSDL [18] and SOAP [17].

In short, Axis2 provides us the technology for exchanging messages with WSs over a network. Consequently, we had to implement only a connection between Axis2 and the simulation automata in our proxies. The box labeled "WSCU" in Fig. 5 represents the component of the proxy that does this, called *Web Service Communication Unit* (WSCU). Roughly, a WSCU works as follows.

- A WSCU monitors the simulation automaton. If this automaton makes a transition, it registers the data items and the synchronization points involved. Subsequently, it packs these data items—payloads—and synchronization points—message names—into an appropriate message format (using Axis2), e.g., SOAP. Finally, it sends these messages over the network to the actual WS (again using Axis2).
- Concurrently, a WSCU receives messages sent by the actual WS (using Axis2). Subsequently, it unpacks these messages (e.g., removes headers) and writes their payload as data items on the appropriate synchronization points. Importantly, a WSCU forces the simulation automaton to make a corresponding transition. Otherwise, this automaton and the actual WS can diverge.

In our current implementation, proxies of WSs run on the same machine as the circuit orchestrating them. This has a practical reason: typically, we cannot deploy applications on the remote machines on which the WSs run.

Generating Proxies. Previously, we outlined why we need proxies and how they work. Next, we describe how the Proxy Generator generates them. To generate a proxy for a single WS, the Proxy Generator requires two inputs: a WSDL document and a WS *behavior specification* (WSBS). The WSDL document specifies the syntax and technical details of the interface of the WS; the WSBS formally describes its (externally observable) behavior.[7]

To explain in more detail how the Proxy Generator works, suppose a WSDL document `service.wsdl` and a WSBS file `service.wsbs`. (Suppose they describe the same WS.) The Proxy Generator proceeds in three steps.

[6] Currently, we assume that a WS and a simulating automaton *start* and *stay* in sync. Communication errors, for instance, can take the two out of sync, but a proxy can detect such situations and recover or reset itself to reestablish its sync with its respective actual service.

[7] We assume that WS providers publish sufficient information about the externally visible behavior of their WSs to construct a faithful WSBS in some formalism. Note that not only our approach requires such a behavioral description: generally, if WS providers want to enable others to compose the services they provide, this comprises essential information (especially in the case of stateful WSs).

- First, the Proxy Generator parses `service.wsdl` using Axis2 technology; the box labeled "WSDL Parser" in Fig. 4 represents the component involved. (Note that both the Proxy Generator and the generated proxy use Axis2, albeit in different ways.) Subsequently, the Proxy Generator generates a Web Service Communication Unit (WSCU) based on the previous parsing; the box labeled "WSCU Generator" represents the component involved.
- Second, the Proxy Generator parses `service.wsbs`; the box labeled "WSBS Parser" in Fig. 4 represents the component involved. More precisely, this component parses the content of `service.wsbs` to a Constraint Automaton (CA). Subsequently, similar to the Reo Compiler (see Section 4.1), the Proxy Generator generates Java code for the CA resulting from parsing `service.wsbs`. In fact, the boxes labeled "CA2Java Code Generator" in the Reo Compiler and Proxy Generator refer to the same component. Instances of the resulting Java class serve as simulation automata at runtime, as described above.
  The current version of the Proxy Generator handles WSBS files describing CA. We aim to experiment with other languages for specifying WS behavior: in principle, our approach supports any modeling language for which we can devise a mapping to CA. For instance, we can use the tool presented in [5] to automatically translate UML Sequence Diagrams to Reo circuits. Combined with the Reo2CA component in Fig. 4, this yields CA that can serve as WSBSs. We outline this further in Sec. 5.
- Finally, the Proxy Generator combines the generated WSCU and the generated simulation automaton class by adding glue code between them. More concretely, this step yields a Java class `Proxy`. The box named "Java Proxy Code" in Fig. 5 represents this class. Instances of `Proxy` run as proxies, encapsulating the constituent WSCU and simulation automaton.

4.3 Gluing Together Orchestrators and Proxies

We discuss how to combine generated proxies of WSs with a compiled circuit that orchestrates them—we describe the box named "Final Code Merger" in Fig. 4. We start with some notation. Suppose an XML document `conn.xml` specifying a circuit C. Moreover, suppose a set of $\langle$WSDL, WSBS$\rangle$-pairs $\widehat{S}$ representing the set of WSs S that C orchestrates. Let `Conn` denote the Java class the Reo Compiler yields on input of `conn.xml` (i.e., the box named "Java Circuit Code"); let `Proxy` denote the class the Proxy Generator yields on input of some $\langle$`service.wsdl`, `service.wsbs`$\rangle \in \widehat{S}$ (i.e., the box named "Java Proxy Code").

The box named "Final Code Merger" in Fig. 4 represents the component that glues together `Conn` and the `Proxy` of every $\widehat{s} \in \widehat{S}$. This gluing yields a collection of Java classes that, once deployed, orchestrate the WSs in the set S as desired. The "Final Code Merger" produces a gluing Java class that comprises the following activities.

1. Create a synchronization point for each boundary node of C.
2. Create an instance `conn` of the class `Conn` generated by the Reo Compiler on input of `conn.xml`. Moreover, pass the synchronization points created in the

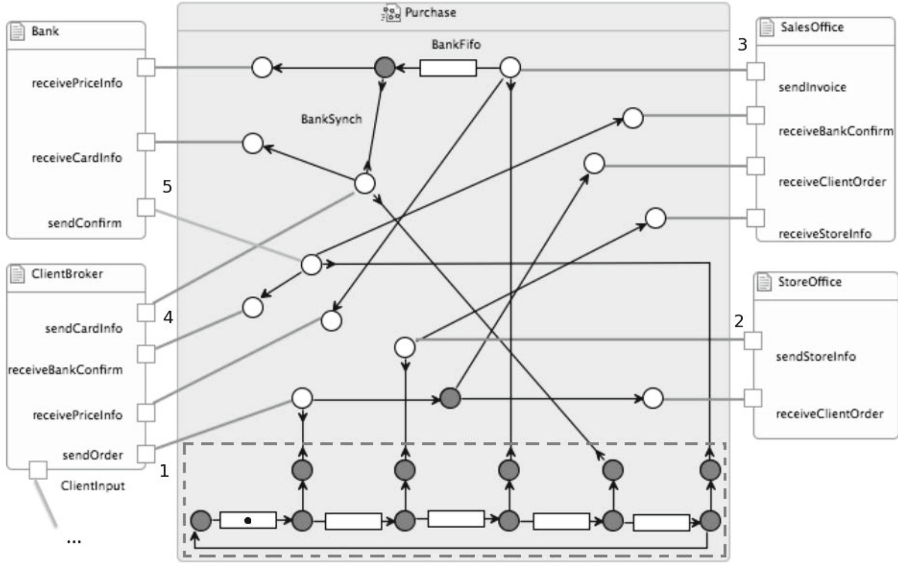


Fig. 7. The sequential coordination of four WSs represented as a Reo circuit: the numbers represent the ordering of the exchanged messages

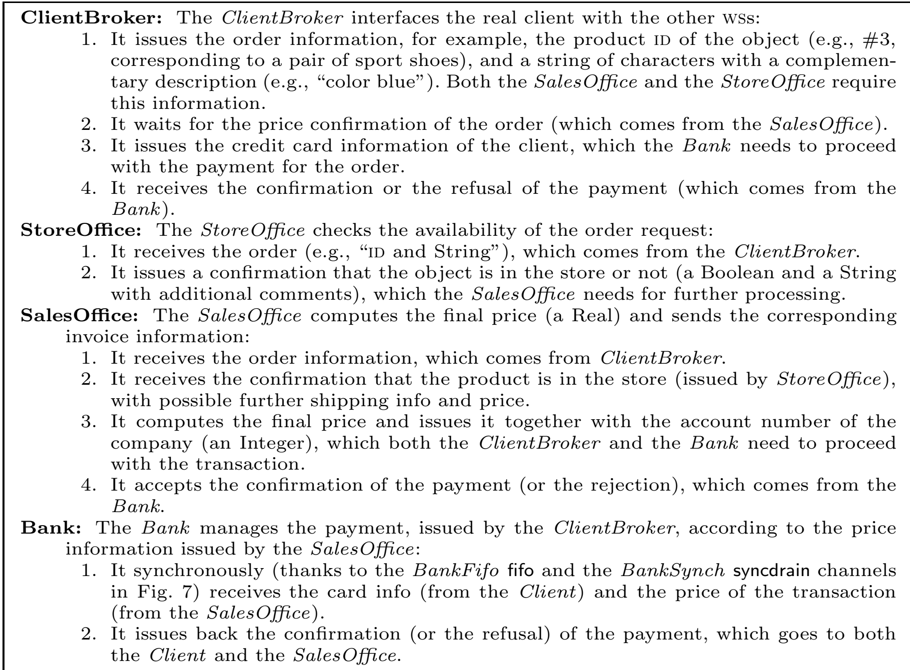
previous step to `conn`. These synchronization points constitute the interface through which proxies communicate with `conn` (see Sec 4.1).

3. Create an instance `proxy` of the class `Proxy` generated by the Proxy Compiler on input of $\widehat{s} = \langle \texttt{service.wsdl}, \texttt{service.wsbs} \rangle$ for each $\widehat{s} \in \widehat{S}$. Importantly, pass to this instance the *appropriate* synchronization points created in step (1). The sharing of synchronization points between `conn` and `proxy` establishes the link between the orchestrator and a WS.

5 Case Studies

In this section, we first present a simple yet nontrivial example of orchestration to familiarize the reader, and then, we extend the Reo circuit of this first example to show a more complex interaction protocol among (almost) the same services of the first example, to show the expressiveness of Reo at full.

In Fig. 7 we can see the complete representation of the circuit for the orchestration of four WSs. The image has been created with the ECT. With the aid of the Reo Compiler (see Sec. 4.1) we generated the *Purchase* circuit describing the interaction among the four WSs named *ClientBroker*, *StoreOffice*, *SalesOffice*, and *Bank*. This scenario implements the classical purchase-online example. The *ClientBroker* service takes care of interfacing the real client to the other services, which deal with: the information about the store (i.e., the *StoreOffice* service), the procedure to prepare the invoice (i.e., the *SalesOffice* service), and the



ClientBroker: The *ClientBroker* interfaces the real client with the other WSs:

1. It issues the order information, for example, the product ID of the object (e.g., #3, corresponding to a pair of sport shoes), and a string of characters with a complementary description (e.g., "color blue"). Both the *SalesOffice* and the *StoreOffice* require this information.
2. It waits for the price confirmation of the order (which comes from the *SalesOffice*).
3. It issues the credit card information of the client, which the *Bank* needs to proceed with the payment for the order.
4. It receives the confirmation or the refusal of the payment (which comes from the *Bank*).

StoreOffice: The *StoreOffice* checks the availability of the order request:

1. It receives the order (e.g., "ID and String"), which comes from the *ClientBroker*.
2. It issues a confirmation that the object is in the store or not (a Boolean and a String with additional comments), which the *SalesOffice* needs for further processing.

SalesOffice: The *SalesOffice* computes the final price (a Real) and sends the corresponding invoice information:

1. It receives the order information, which comes from *ClientBroker*.
2. It receives the confirmation that the product is in the store (issued by *StoreOffice*), with possible further shipping info and price.
3. It computes the final price and issues it together with the account number of the company (an Integer), which both the *ClientBroker* and the *Bank* need to proceed with the transaction.
4. It accepts the confirmation of the payment (or the rejection), which comes from the *Bank*.

Bank: The *Bank* manages the payment, issued by the *ClientBroker*, according to the price information issued by the *SalesOffice*:

1. It synchronously (thanks to the *BankFifo* fifo and the *BankSynch* syncdrain channels in Fig. 7) receives the card info (from the *Client*) and the price of the transaction (from the *SalesOffice*).
2. It issues back the confirmation (or the refusal) of the payment, which goes to both the *Client* and the *SalesOffice*.

Fig. 8. Description of the services in Fig. 7

effective payment management (i.e., the *Bank* service). The complete high-level behavior of these services is described in Fig. 8.

The dashed rectangle in Fig. 7 highlights a Sequencer of the messages (see Sec. 3), i.e., a Reo subcircuit that enforces the correct ordering of the messages exchanged among the WSs. Therefore, the interaction of the WSs is sequential: the sequence consists of five steps, whose ordering is shown in Fig. 7 with ordinal numbers beside the sink ports of the components.[8]

We programmed and deployed the WSs on a server machine, and afterwards we automatically generated their proxies with the help of the Proxy Generator (see Sec. 4.2). We described the four WSs as UML *Sequence Diagrams* as represented in Fig. 9, where labels correspond to the types of exchanged SOAP messages. From this description, we can generate the corresponding CA as described in Sec. 4.2.

In Fig. 10, we show essentially the same purchase interaction as in Fig. 7, albeit with a more complex behavior. The scenario in Fig. 10 is closely derived from [15] and the related circuit is named *ComplexPurchase*. As in the first

[8] The presence of the Sequencer in Fig. 9 may appear redundant as the WSs themselves already impose an ordering on their interactions. However, the sequencing of the messages *is* also part of the protocol among the WSs and should, therefore, be part of the protocol specification—regardless of what the WSs involved do.

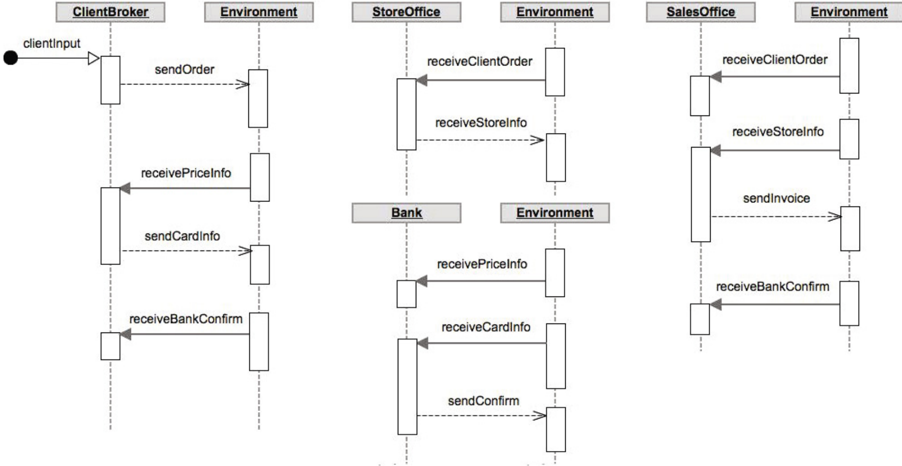


Fig. 9. The UML Sequence Diagrams for the four services in Fig. 7

case study, we represent a *ClientBroker*, a *Bank*, a *SalesOffice*, and a *StoreOffice* service. Additionally, in this second case study, we include a *ShippingOffice*, which accomplishes the task of controlling shipping details, e.g., a shipping fee.

The transaction is initiated by the *ClientBroker*, which on receiving the purchase order message from the customer, initiates parallel flows to handle shipping (*ShippingOffice*), invoicing (*SalesOffice*), and scheduling (*StoreOffice*), concurrently. While some of the activities in the transaction may proceed concurrently, there are control and data dependencies among the services, thus coordination is needed to execute the transaction. For example, to complete the price calculation by the *SalesOffice* service, the shipping price is required. Once the parallel flow is finished, the invoice is delivered to both the *Bank* and the *ClientBroker*. Then, on behalf of the customer, the *ClientBroker* sends the credit card information (which can be accepted or not by the filter channels, see Sec. 3) to the *Bank*, which sends the outcome of the financial transaction back to the *ClientBroker*.

The dashed rectangle in Fig. 7 highlights a Sequencer subcircuit. The high number of fifo channels is due to the fact that the activities of *ShippingOffice*, *StoreOffice*, and *SalesOffice* run in parallel: the fifo channels are used to buffer the messages in case one of these service is not yet ready to accept them.

Note that in this example, we adopt filter channels to check if the format of the message is coherent with the given specifications, e.g., if the expiry date of a credit card is after the current date or not. This shows that Reo circuits can perform such "active" controls. They are not only passive routers of data. One more data-aware action is performed during the final delivery of the invoice and the credit card information to the *Bank*. This is carried out by a *join node*, denoted by $\oplus$ in Fig. 10, which represents a component that accepts data items from all connected sink ends and creates a tuple out of them: all the information is merged into one message only, representing the complete request to the *Bank*.

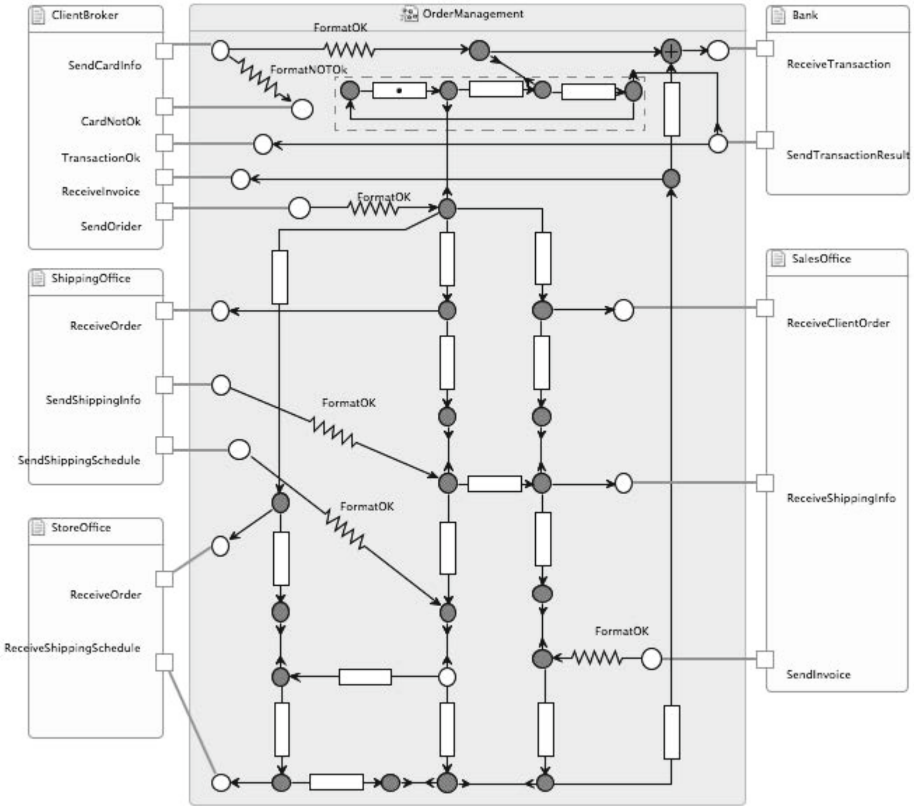


Fig. 10. The example in Fig. 7 with a more complex interaction

6 Conclusion

In this paper, we have shown how to automatically generate an orchestration framework for WSs. The orchestration is defined by using the Reo language, and the generated Java code is used to compose the behavior of the services in a way transparent to the client and all the services. The input of our generation tool consists of the externally observable behavior of each service, the WSDL description file of each service, and the specification of the orchestration as a Reo circuit. From all this information, it is then possible to automate the Java code generation process from a Reo circuit and a proxy (see Sec. 4.2) for each service. This proxy component is in charge of managing the communication between the technology behind the service and the Reo environment.

This research can proceed in the future along different lines. Our first intention is to be able to generate different communication units for the proxy (not only the WSCU, see Sec. 4.2), in order to include other technologies in the orchestration of components and services, e.g., CORBA, RPC, WCF. We intend to have a multi-technology platform to integrate several kinds of third-parties.

Moreover, we would like to study different service behavioral description schemes to generate the code of service proxies according to other kinds of input. A possible choice can be various UML diagrams, e.g., *Activity Diagrams* or *State Machines*.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. MSCS 14(3), 329–366 (2004)
2. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. SCP 61(2), 75–113 (2006)
3. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and Pipelines for Structured Service Programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)
4. Cesari, L., Pugliese, R., Tiezzi, F.: A tool for rapid development of ws-bpel applications. SIGAPP Appl. Comput. Rev. 11(1), 27–40 (2010)
5. Changizi, B., Kokash, N., Arbab, F.: A Unified Toolset for Business Process Model Formalization. In: Proceedings of FESCA 2010 (2010)
6. Decker, G., Kopp, O., Leymann, F., Pfitzner, K., Weske, M.: Modeling Service Choreographies Using BPMN and BPEL4Chor. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 79–93. Springer, Heidelberg (2008)
7. Hull, R., Benedikt, M., Christophides, V., Su, J.: E-services: a look behind the curtain. In: PODS, pp. 1–14. ACM (2003)
8. Jayasinghe, D., Azeez, A.: Apache Axis2 Web Services. Packt Publishing (2011)
9. Jongmans, S.S., Arbab, F.: Overview of Thirty Semantic Formalisms for Reo. SACS 22(1), 201–251 (2012)
10. Kokash, N., Krause, C., de Vink, E.: Reo+mCRL2: A framework for model-checking dataflow in service compositions. FAC 24(2), 187–216 (2012)
11. Meng, S., Arbab, F.: Web Services Choreography and Orchestration in Reo and Constraint Automata. In: Proceedings of SAC 2007, pp. 346–353 (2007)
12. Meng, S., Arbab, F.: QoS-Driven Service Selection and Composition Using Quantitative Constraint Automata. FI 95(1), 103–128 (2009)
13. Meng, S., Arbab, F.: A Model for Web Service Coordination in Long-Running Transactions. In: Proceedings of SOSE 2010, pp. 121–128 (2010)
14. Montesi, F., Guidi, C., Lucchi, R., Zavattaro, G.: JOLIE: a Java Orchestration Language Interpreter Engine. ENTCS 181, 19–33 (2007)
15. Web services business process execution language (2007), `http://docs.oasis-open.org/wsbpel/2.0/`
16. Peltz, C.: Web Services Orchestration and Choreography. IEEE Computer 36(10), 46–52 (2003)
17. Simple Object Access Protocol (2000), `http://www.w3.org/2000/xp/Group/`
18. Web Service Description Language (2001), `http://www.w3.org/TR/wsdl`
19. Web Service Choreography Interface (2002), `http://www.w3.org/TR/wsci/`
20. Web Services Choreography Description Language (2005), `http://www.w3.org/TR/ws-cdl-10/`
21. Zhang, J., Chung, J.-Y., Chang, C., Kim, S.: WS-Net: A Petri-net Based Specification Model for Web Services. In: Proceedings of ICWS 2004, pp. 420–427 (2004)