A MESSAGE-PASSING MODEL FOR SERVICE ORIENTED COMPUTING*

Diana Allam, Rémi Douence, Hervé Grall, Jean-Claude Royer and Mario Südholt
ASCOLA group; EMN-INRIA, LINA, Département Informatique, École des Mines de Nantes, Nantes, France

Keywords: Service-oriented Computing, Message-passing Model, Type Checking.

Abstract: Web services can be built according to multiple service models and technologies. Although there is a clear need for a model integrating them in multiple real-world contexts, no integrated model does (yet) exist. In this paper, we introduce a model as a foundation for heterogeneous services, in particular, SOAP/WS* and RESTful services. The model abstracts away from service implementations, composes services in a truly concurrent manner and supports asynchronous message passing as well as mobility of typed channels.

1 INTRODUCTION

Web services can be built according to multiple service models, technologies and protocols (Alonso et al., 2004). Despite the different architectures underlying these service models and the applications they target, web service models have similar characteristics and are often used together. In such a heterogeneous system, despite existing security standards, sophisticated attacks regularly emerge (Lindstrom, 2004). Most of these attacks are enabled by weaknesses in web service security policies. A solution to this problem of heterogeneous service compositions can, in principle, be based an message-passing models that mediate between different service models. There are some existing models in the market satisfying this message-passing principle, like the Enterprise Service Bus (ESB) that is available for different service models and technologies. However, these models are not systematically used: frequently, *e.g.,* WS* services call RESTful services directly. Furthermore, ESB-based implementations are not amenable to formal reasoning over properties of service compositions. In order to ensure properties over heterogeneous service compositions, a more general and rigorous abstract model is needed.

Actually, service models can be unified based on a few concepts: distributed agents are built by means of interacting distributed services that are composed in a black-box manner using well-defined service interfaces. These agents provide services while requiring other services that are consumed. Such a global abstraction based on the notion of black-boxes for SOAs provides significant benefits for the management of distributed services: their security properties, in particular, can then be specified through contracts at the interaction level. A formal and unified model can then be used to enforce security properties by means of a security monitor that mediates between an encapsulated agents and the network.

We propose a unified model as a foundation for service compositions that represents services in terms of an abstract pivot language that is independent from the service model and corresponding implementation technology. Once all distributed services are unified, the implementation and reasoning about security properties can be performed at the message exchange level and enforced before the execution of service compositions. In this paper, we motivate and informally define such a unified model (respectively in sections 2 and 3). We then illustrate by an example the conversion of services interfaces conforming to the WS* and RESTful service models into our language (sec. 4). We then illustrate the benefits to the reasoning about service composition properties through a simple application to service type checking (Sec. 5). We show how well-typed services solve problems that might occur because of programmer mistakes or lack of information during runtime service discovery.

The formal account of the model and its application to the verification of several security properties are provided in a companion report (Allam et al., 2011).

*This work has been supported by the CESSA ANR project (see http://cessa.inria.gforge.fr).

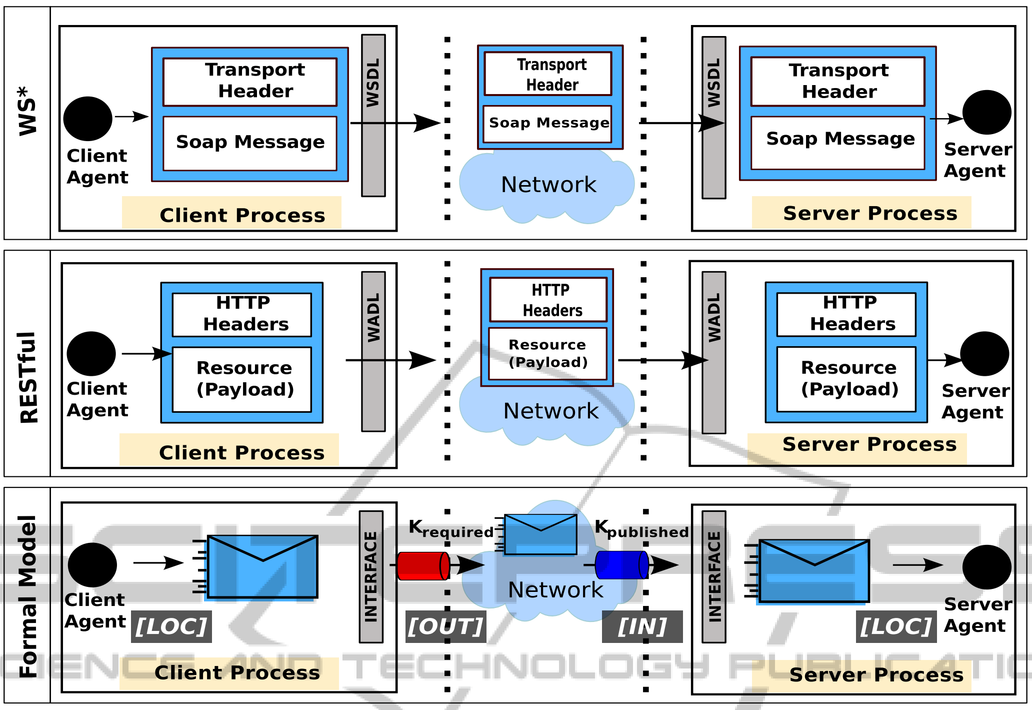

Figure 1: Flight booking request: relationship between WS*/RESTful and our formal model.

2 MOTIVATION BY EXAMPLE

We motivate and illustrate our approach through a running example, a service-based system for flight reservation. We focus our illustration on the two most competing models in web services: SOAP/WS* and RESTful. We consider a two-step scenario: a client agent first searches a flight travel from a source city to a target one at a specific date; in a second step, it receives a list of possible flights, makes its choice and books a flight. We investigate two different implementations for this example, respectively using WS* and RESTful services. Here, we aim to find the main concepts allowing to define a unified abstract model for both technologies: RESTful and WS*. In the following, we discuss three core concepts in light of the flight service example: *message exchange*, *black box agent* and *channel mobility*.

Message Exchange. The first two rows of Figure 1 show message exchanges that are part of a flight booking request. This request may hold a SOAP message over a transport protocol (HTTP, FTP, SMTP or others) governed by the WS* standards or a simple HTTP message wrapping a payload for RESTful services. Thus, an abstract formal model unifying these two technologies must specify a message format compatible with both of them.

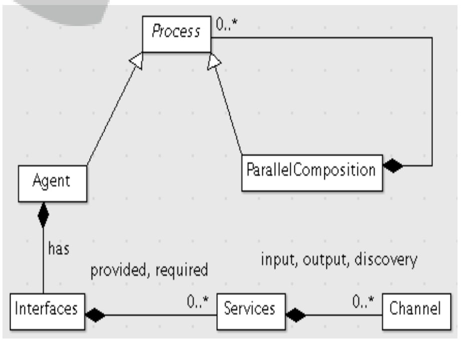

Figure 2: UML diagram for processes.

As illustrated in the bottom row of Figure 1, our abstraction of message exchanges corresponds to three communication rules : (i) [LOC]: abstracts local executions that consume incoming messages and produce messages to be sent over the network; (ii) [OUT]: enables the passage of a message throughout the agent interface to the network; and (iii) [IN]: enables the receipt of a message from the network to the designated agent throughout its interface.

Black Box Agent. The service-based implementation of the flight reservation process requires a whole stack of different layers and business components: data bases and associated access interfaces, business processes that coordinate (*e.g.,* using BPEL) the execution of business components that implement enterprise functionalities and a suitable interfaces must be defined that expose the business processes as services. To specify such an interface, we can use WS-Management WSDL (Web Services Description Language) for the WS* implementation, a method more expressive than a simple WSDL file. Typically interaction contracts based on exchanged SOAP messages are defined using WS-Addressing, WS-Discovery, WS-Policy, WS-Security and others. As for RESTful services, no standard exists that provides such interface descriptions. For our example, we consider a WADL (Web Services Application Language) file.

In accordance with the *black-box principle*, we are interested in an abstraction that hides all the implementation details of web services. We model an elementary process as a composition of an interface and an agent having an internal state which evolves during its execution. The agent abstraction hides all implementation details for the data base access layer and the business process layer. The interface is composed of provided or required services. Each service is a set of channels to receive incoming messages or to send messages over the network. This abstraction is illustrated in Figure 1 for a client and a server on a channel *K* refering to the ”book flight” service.

Channel Mobility. We describe two practical important aspects of this feature.

Request/Reply Mechanism. In the client/server communication of the flight reservation scenario, the server does not initially know the client. Thus for the reply, the server uses the callback information contained in the received message. Concretely, this information could correspond to a source IP address and port over a transport protocol (e.g. HTTP) or an endpoint reference (EPR) in the *ReplyTo* block of a WS-Addressing header for a SOAP message or a specific location in an HTTP message. This way, the callback mechanism can be performed synchronously, usually over one session of the transport protocol, or asynchronously over two distinguished client/server sessions. Our formal model must therefore be expressive enough to represent both synchronous and asynchronous communications. However, since synchronous communication can be implemented with message passing (Lamport and Lynch, 1990), our model only supports asynchronous communication, via one-way channels. A reply channel, corresponding to a callback, must therefore be sent by the client.

Discovery Mechanism. Suppose that the client does not know the location of the flight reservation service while knowing its interface. Thus, at runtime, before invoking the service, the client needs to discover a target location for the service. In case of WS* technology, several techniques can be used for service discovery. The UDDI standard can be used if a centralized registry is appropriate to discover web services. As part of an ad-hoc architecture, the WS-Discovery standard provides endpoint references for the flight service. In the case of a RESTful implementation, search engines, like Google, could be used for getting the root resource URL. Then a RESTful service could be *”self-discovery”* by defining links between resources. Service discovery thus means looking up a set of channels.

3 MODEL OVERVIEW

In this section we give an overview of the concepts of our abstract model for unifying web services technologies. Our model belongs to the class of *message-passing models* (Lamport and Lynch, 1990): agents exchange messages by using a buffer and without sharing memory or without synchronizing the sending and the receiving of messages in a rendez-vous.

Communication is assumed to be completely asynchronous. The buffer used to communicate models the network. We consider it as a finite multiset of messages, with no bound and no order. Actually, a message is defined as some content on a channel. The channel determines the unique target of the message. Initially, coordination is only possible between agents that share a channel, for instance between a server providing a channel and its clients, requiring the same channel. Gradually, agents discover new channels since messages can contain channels: channel mobility makes network topology evolve.

The two main requirements of our model, asynchronous communication and true concurrency, has led us to resort to a chemical model (Berry and Boudol, 1992). Here, we describe the concepts using UML class diagrams and informally present the main semantic rules.

Processes and Agents. Processes are described in Figure 2. This UML diagram describes distributed agents implementing web-services. A process is made from several agents acting in a parallel composition. This concept is represented thanks to a composite pattern in Figure 2. An agent has a name, a state and an interface. The state of agents is kept abstract, in accordance with the black-box principle. Different formalisms, like process algebras or temporal logics, could therefore be used to model agents. An interface declares a set of provided and required services by the agent. A service is a set of channels with *"Input"*, *"Output"* or *"Discovery"* roles. *"Input"* channels correspond to the channels receiving messages from the network. *"Output"* channels correspond to the channels sending messages to the network. *"Discovery"* channels can be communicated to another agent by putting them in the message content in order to be discovered at receipt as *"Output"* channels.

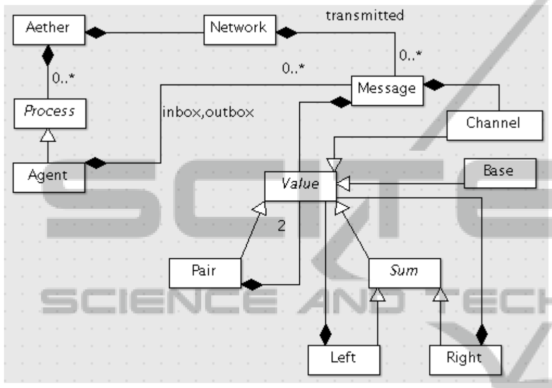

Figure 3: Simplified UML diagram for Aether.

Aether. Processes defined previously are deployed in a chemical solution which we call "Aether". Aether represents the global state of the computing world of processes. The UML diagram of the aether description is depicted in Figure 3. The aether is a set of parallel agents and a network which contains messages in transit. An agent has two mailboxes associated respectively to incoming (outgoing) messages from (to) the network. A message is defined as a value on a channel. As different messages can have exactly the same form (same channel, same content), and, generally, emission ordering is not respected by communications, we use multisets to store messages. During aether's evolution (by reduction), messages are emitted by agents into the network or consumed from the network. We abstract away message contents and strictly consider constructions required with a black box view of web service communications. Our aim is to abstract important constructions in SOAP and HTTP messages, which are the two most common transport formats of messages in web services. In our model, message values carried by channels support base values, mobile channels, pairs of two values and left and right injections associated to disjoint union of values. The two abstract constructions, product and sum, are common in data format like XML and JSON.

Informal Semantics. The formal semantics can be given by using a chemical abstract machine as defined by Berry and Boudol in (Berry and Boudol, 1992). Here we informally describe the three specific rules of the machine, corresponding to reaction rules: see Figure 1.

[LOC]: An agent consumes a multiset, possibly empty, of input messages, produces a multiset, possibly empty, of output messages, creates dynamically new sessions and updates its current state to a new state.

[OUT]: An agent sends message from its outbox over the network via channels. If the sent message contains channels, these channels must be *"Discovery"* channels known by the agent.

[IN]: An agent receives message in its inbox from the network. The message channel must be declared as an input channel by the agent. At receipt there is dynamic discovery of unknown channels. The agent upgrades its declaration of *"Output"* channels by adding all the channels discovered in the content of the message.

4 EXAMPLE REVISITED

We now show how to apply our unified model to the flight reservation scenario of Sec. 2. We discuss, in particular, the relation of the formal model with the WS* and RESTful models. We use a java syntax to represent this scenario in our model.

Processes Initialization. The client/server system of the flight reservation scenario requires the definition of four channels, as declared in the following:

```
// Channel for requesting trips from a
// source to a target city at a specific date
Channel searchTravel = new Channel(searchType);
// Channel for getting a list of trips
Channel getSearchReply = new Channel(searchReplyType);
// Channel for booking requests for a
// specific flight of a client
Channel bookFlight = new Channel(bookingType);
// Channel for getting a booking confirmation
Channel getBookingReply = new Channel(notifReplyType);
```

According to these channels declarations, now we are able to define the server and the client processes.

The server provides the service named "FlightReservation" composed of two *"Input"* channels: `searchTravel` and `bookFlight`. Now, we give a declaration of the server process, according to the UML description given in Sec. 3.

```
// Creation of the service with two
// input channels
Service s = new Service("FlightReservation");
s.addInputChannel(searchTravel);
s.addInputChannel(bookFlight);
// Creation of an interface providing "s"
Interface i = new Interface();
i.provide(s);
// Creation of the server process
// with interface i
Process server = new Agent(i);
```

As for the client process, it requires the service provided by the server. The two channels `searchTravel` and `bookFlight` are then declared as *"Output"* channels. The client must declares the two channels `getSearchReply` and `getBookingReply` for getting replies from requests on `searchTravel` and `bookFlight` respectively. The two channels, `getSearchReply` and `getBookingReply`, are *"Input"* channels. They have also the role of "*Discovery*" channels in the sense that these two channels can be communicated to the server in order to get replies. The client declaration is given in the following:

```
// Creation of the service with two
// output channels and two
// Input/Discovery channels
Service s = new Service("FlightReservation");
s.addOutputChannel(searchTravel);
s.addOutputChannel(bookFlight);
s.addInputDiscoveryChannel(getSearchReply)
s.addInputDiscoveryChannel(getBookingReply)
// Creation of an interface requiring "s"
Interface i = new Interface();
i.require(s);
// Creation of the client process
// with interface i
Process client = new Agent(i);
```

Finally, the global system is seen as a parallel composition of the client and the server processes.

WS*-based Implementation. In case of an implementation based on WS* services, a WSDL file is used with a WS-Management specification as we mentioned in Sec. 2. The resulting WSDL file is used by the client to invoke the services of the flight server. In this file, the flight service is represented by a port linked to a binding description and to an EndPoint Reference (EPR). The EPR associates:

- A URI address that identifies the service, L_{server} = "http://flight-travel-service".
- A port type which defines the two methods: "*searchTravelOp*" and "*bookFlightOp*",
- Properties which may contain the individual properties that are required to identify the entity or the resource being conveyed.
- A service name, *e.g.,* "FlightReservation".
- A Policy defined using WS-Policy.

In our scenario, we consider a policy which requires an EPR in the "*ReplyTo*" block of the WS-Addressing SOAP header for the request messages. We suppose that the two methods "*searchTravelOp*" and "*bookFlightOp*" are request/reply, thus they define an input (InMsg) and an output (OutMsg) messages. Suppose that the client is accessible via a location L_{client}, which is the URI address communicated in the SOAP header *ReplyTo*. According to this brief interface description, we associate channels with their corresponding formal definitions as follows:

- searchTravel is L_{server}.searchTravelOp.InMsg
- bookFlight is L_{server}.bookFlightOp.InMsg
- getSearchReply is L_{client}.searchTravelOp.OutMsg
- getBookingReply is L_{client}.bookFlightOp.OutMsg

Implementation using RESTful Services. The implementation of our service interface is different using the RESTful technology. The flight reservation service is composed from two resources:

- "*flightTravel*" accessible by the URL: $L1_{server}$ = "http://flight/travel"
- "*flightReservation*" accessible by the URL: $L2_{server}$ = "http://flight/reservation"

Each resource has four methods: GET, PUT, POST and DELETE. The GET method for the *flightTravel* resource replaces the "*searchTravelOp*" operation of the previous WSDL example. The POST method for the "*flightReservation*" resource replaces the "*bookFlightOp*" operation defined in the WSDL file. Both of these two methods are request/reply and can be defined using the Web Application Description Language (WADL) proposal for standardization.

Suppose that L_{client} is a source IP address and port over HTTP, we associate the following meaning for our declared channels:

- searchTravel is $L1_{server}$.GET.RequestMsg
- bookFlight is $L2_{server}$.POST.RequestMsg
- getSearchReply is L_{client}.GET.ResponseMsg
- getBookingReply is L_{client}.POST.ResponseMsg

5 APPLICATION TO TYPE CHECKING

As an application of our model to the securization of distributed services, we consider type checking. There is rather few formal work on type checking in the context of web services, a notable exception being the work of Sans and Cervesato (Sans and Cervesato, 2010), which presents an abstract model for web applications. Considering once again the flight reservation scenario, we wish to apply type checking on the client agent.

Scenario and Problems. Let us consider that the client process is implemented by using the `cxf` framework of the Apache foundation framework[2]; the server process is exposed as a RESTful server. As multiple ways exist to call the flight reservation service, there are different implementation cases for the client developer:

(i) The programmer knows the server *"EndPoint"* and he refers to the WADL file provided by the server to generate his code. In this case the programmer of the client agent can type-check his code.

(ii) The programmer knows the server *"EndPoint"* but he does not want to use the WADL file. Instead, he wishes to call the RESTful service by making explicit HTTP invocations, using the HTTP-centric API exposed by `cxf` framework. Thus, the client could send ill-typed HTTP messages; the developer needs to be careful to ensure that only correct messages are sent.

(iii) The programmer does not know the server and he discovers it, potentially at runtime; a discovery problem could occur in this case if the discovered service is sent or modified by a malicious agent. For example, an agent could communicate with the malicious one and believe, erroneously, that the latter one provides a particular service. Such an error gives the opportunity to attackers on the network for hacking web services by simply flooding servers with erroneous messages.

Our Proposed Solution. To avoid all of these issues, we propose to use a framework to represent the client and the server agents in our abstract model, in terms of the description given in Sec. 4[3]. Before execution, the two agent interfaces must be checked in order to exclude typing errors. This check consists in the evaluation whether each required channel has a type compatible with the corresponding provided channel. Moreover, if a discovery mechanism is required at run-time to probe for an *"EndPoint"* reference, an authentication mechanism is required to avoid discovering services with fake information sent by hackers. If the system is not well typed, the implementation generates a set of typing errors. To correct this problem, the developer has two possibilities: 1) he can modify statically his code to correct all the type problems, or 2) type checking can be implemented separately from the implemented agent, thus the check will occur at runtime before emitting the message or at the moment of reception if we allow messages to be emitted by unchecked agents. Such dynamic type checking can be assured, *e.g.,* by a proxy that wraps the local agent.

[2]http://cxf.apache.org

[3]For lack of space we only present a brief informal description of our typing method. A detailed account can be found in (Allam et al., 2011).

6 RELATED WORK

In the following we present related work for black-box models. There is some interesting works, like (Seehusen and Stolen, 2009; Keller et al., 2006). In (Seehusen and Stolen, 2009), authors define a formal and abstract model for services. The semantics are based on a notion of trace which is a sequence of events. The intent of this formalization is to abstract message sequence chart as they are used in UML 2. While in (Keller et al., 2006), authors use the notion of abstract state space to specify the functional descriptions of web services. The authors demonstrate the applicability of the formal model by showing how to define and determine realizability and semantic refinement. Comparing these two models to ours, there is different abstraction interests. Our model contributes mainly in the abstraction of existing standards and is applied to type soundness. While (Seehusen and Stolen, 2009) presents an advanced formalization of some confidentiality issues and (Keller et al., 2006) focuses on a particular functional description without a clear defined syntax. Another interesting work is (Sans and Cervesato, 2010) where authors share with us a general view of the web services interactions but without parallelism and asynchrony. Their work covers code mobility which we do not address here, we are only concerned with the remote procedure call.

7 CONCLUSIONS

In this paper, we have presented an abstract model unifying web service composition in heterogeneous environments, such as those requiring interaction between backend servers that use WS* services and mobile appliances that use RESTful services. We have illustrated the model in the context of a flight booking example. Finally, we have briefly presented an application of the model to type checking. We have shown how to support the detection of security-critical type errors, notably security flaws due to service discovery attacks, an interesting topic for future work.

REFERENCES

Allam, D. et al. (2011). Extension of the service model for security and aspects. Deliverable D1.3, CESSA ANR project, no. 09-SEGI-002-01.

Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004). *Web Services: Concepts, Architectures and Applications*. Springer, Berlin.

Berry, G. and Boudol, G. (1992). The chemical abstract machine. *Theoretical Computer Science*, 96(1).

Keller, U. et al. (2006). On the semantics of functional descriptions of web services. In Sure, Y. and Domingue, J., editors, *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference*, volume 4011 of *LNCS*, pages 605–619. Springer.

Lamport, L. and Lynch, N. (1990). Distributed computing: models and methods. In *Handbook of Theoretical Computer Science vol B*. Elsevier.

Lindstrom, P. (2004). Attacking and defending web services. In *A Spire Research Report*.

Sans, T. and Cervesato, I. (2010). Qwesst for typesafe web programming. 3rd International Workshop on Logics, Agents, and Mobility (LAM'10) Edinburgh, Scotland, http://www.qatar.cmu.edu/ tsans/index.php?page=research.

Seehusen, F. and Stolen, K. (2009). Information flow security, abstraction and composition. *IETF Information and Security*, 3(1):9–33.