Service integration via target-transparent mediation

Mila Dalla Preda*, Maurizio Gabbrielli*, Claudio Guidi†, Jacopo Mauro* and Fabrizio Montesi‡

*Lab. Focus, Department of Computer Science/INRIA, University of Bologna, Italy.
Email: dallapre | gabbri | jmauro @unibo.it
†italianaSoftware srl, Imola, Italy. Email: cguidi@italianasoftware.com
‡IT University of Copenhagen, Denmark. Email: fmontesi@itu.dk

Abstract—**In the context of Service-Oriented Architectures (SOAs), the integration of services is an important aspect that is usually addressed by using specific tools, such as Enterprise Service Bus (ESB). In this paper we propose a framework to perform service integration building on the extension of service interfaces, capturing a class of service integrators that are decoupled from the services they integrate in an SOA. We show how our service integrators can be used in practice by evaluating our approach with Jolie, a service-oriented programming language. Finally, we present how our methodology differs from the standard practice with ESB.**

Keywords-**SOC; SOA; integration of services; interfaces.**

I. INTRODUCTION

Service-Oriented Computing (SOC) is a programming paradigm for distributed systems based upon the composition of services. Services are autonomous computational entities that can be dynamically discovered and invoked.

In the last decade, research and practice on SOC have led to the stratification of service composition into three layers.

The first layer deals with the implementation of the basic computational features of a service, usually obtained through general-purpose programming languages such as Java, C#, or Python. A service is a program deployed at a given location (e.g., a URL) that can be used by external parties to invoke the service and access its functionalities, dubbed *operations*, by means of message passing. The list of the operations provided by a service, dubbed *interface*, is defined in a machine-readable document that reports their respective names and expected message types.

The second layer composes the operations offered by the services in a Service-Oriented Architecture (SOA) into workflows, thus creating more high-level features by reusing existing services. These workflows are still implemented by services, dubbed *orchestrators*. In Web Services, orchestration is usually supported by the WS-BPEL language [1].

The third, and last, layer covers the integration of different SOAs. For instance, two (or more) systems may need some kind of adaptation of their data formats or functionalities in order to cooperate successfully, or they may need some bridging between their respective network infrastructures. This is usually obtained through an Enterprise Service Bus (ESB). In this paper, we focus on this last layer.

From an abstract point of view, the integration of a service S with another system can be obtained through a *mediator* M: an entity that receives the invocations for S from any client C and redirects them to S after performing some adaptations. A mediator can perform different actions, such as protocol conversion or the addition of specific features (e.g., monitoring or authentication). In the sequel, we refer to a service such as S as the *target* of the mediator M.

In this paper we identify two kinds of mediations: *target transparent* and *target dependent*. Target-transparent mediation does not depend on the content of the message expected by S. It is, for instance, the case for authentication since M needs to check only the *additional information* that it introduces (the authentication credentials). Target-dependent mediation, instead, needs to inspect also the original information required by S.

A notable problem of mediation is that the code for integrating two systems is usually ad-hoc. Mediators, or adaptors, are designed to specifically interact with their targets and adapt them towards external systems. Therefore, mediators are hardly reusable in most occasions. This paper starts from the observation that target-transparent mediation seems particularly suited for modularity. Specifically, since a target-transparent mediator works only with its own added information, it should be possible to develop program mediators that are completely decoupled from their targets. In other words, we want to build target-transparent mediators that can be reused with many different targets. To the best of our knowledge, there have been no proposals of tools that specifically deal with this problem.

We show that Jolie [2], a service-oriented programming language, provides a framework for developing such reusable target-transparent mediators simply allowing the developer to easily extend the service interfaces. We report how using Jolie to obtain this objective elicits a programming methodology that is different than that of other existing tools for mediation, which leads to the design of mediators that can be reused even when some parts of the SOA in which they operate change.

II. PROGRAMMING MEDIATORS IN JOLIE

Writing a complex SOA in a cost-effective way requires the knowledge of different languages and ad hoc tools. Indeed, a SOA developer needs to know at least a general programming language, a domain specific language like

WS-BPEL and one, but often more, integration tools. To overcome this limitation, we propose to use an extension of Jolie [2] - a fully-fledged service-oriented programming language released as an open-source project. In this section we briefly recall some basic Jolie constructs and then we show the definition of a mediator in a simple case study[1].

A Jolie program defines a service as composed by two parts: *behaviour* and *deployment*. A behaviour defines the implementation of the functionalities offered by a service. However, these do not deal with how communications are supported: they abstractly refer to *communication ports*, which are to be correctly defined in the deployment part. The latter deals with the actual definition of the necessary information for supporting communications.

The basic deployment primitives are *input ports* and *output ports*, which support input and output communications with other services. Input and output ports are dual concepts and their syntaxes are quite similar. Ports are based upon the three fundamental concepts of *location* and *protocol*, that define the concrete binding information between a Jolie program and other services, and the one of *interface* that defines type information that is expected to be satisfied by the behaviour that receives invocations through the port.

Communication ports require interfaces to be defined. An interface I is a list of request-response and one-way operations. These are functionalities names with their arguments types thay, in the request-response case, return a value.

Aggregation is the basic mechanism offered by Jolie for programming mediation through the creation of proxy services [4]. Aggregation allows to merge the functionalities of one or more services called providers into one service called aggregator that exposes all the interfaces of the aggregated providers. In the presence of aggregation a service consumer C sees the aggregator M as a unique endpoint to exploit the functionalities of the aggregated providers without knowing which provider implements the functionalities exposed by the aggregator. In [4] we have proposed an extension of the mechanism of aggregation, called *smart aggregation*, that allows the aggregator to manipulate the messages before forwarding them to the providers (for example the aggregator could verify the validity of a key). In general, the interface of the aggregator is an extension of the interfaces of the aggregated services, meaning that the aggregator could require some extra arguments that it consumes before forwarding the message to the corresponding aggregated service. The code that specifies the manipulations that an aggregator has to perform before forwarding the messages to the final provider is specified by the so called *courier code*. The courier code differs from the code of the standard service behavior in the fact that it cannot receive inputs. Figure 1 provides a graphical representation of the mechanism of smart aggregation where the incoming messages m_1^+ and

m_2^+ are manipulated by the courier code and then forwarded as m_1 and m_2 to the corresponding providers. We denote the interfaces of the aggregator as I_1^+ and I_2^+, and the ones of the aggregated services as I_1 and I_2 to highlight the fact that the aggregator receives a message m_1^+ or m_2^+ with some extra arguments that it processes before forwarding the messages m_1 and m_2 to the providers.

The concept of interface extension makes smart aggregation *target-transparent*. Indeed, if the target service exposing I_1 is redeployed with a new interface I_1', the aggregator M (implementing mediation) does not need to be modified. This is because the differences between I_1 and I_1^+ have been explicitly encoded as an interface extension. Thus, the interface extension can be automatically reapplied from the execution engine to I_1' in order to obtain the new aggregator interface $I_1'^+$. Even the mediation logic (courier session code) does not need to be changed, since our programming practice forbids to manipulate the content of the message that will be forwarded to the target service: a courier session can only manipulate the data added through the interface extender, and messages can be forwarded to the target services only as black boxes.

Example: Printer System: We consider a company that develops a service for internal usage that does not need nor support authentication since it is accessible only through the local network. Imagine, without loss of generality, that this service allows the printing of jobs. Abstracting from the implementation, let us suppose that the printer has the following interface and input port for accessing it:

```
type PrintRequest:void { .doc:string }
type PrintResponse:void { .jobId:int }

interface PrinterIface {
RequestResponse:
 print(PrintRequest)(PrintResponse),
 del(DelRequest)(DelResponse) }

inputPort PrinterInput {
Location: "socket://192.168.1.25:8000/"
Protocol: sodep     Interfaces: PrinterIface }
```

Afterwards, the company wants to allow for the printer service to be used also through a public network provided the authentication of non local users. This issue can be addressed by defining a service exposing a new input port that aggregates, via the keyword *Aggregates*, the functionalities of the printer. The aggregator deployment is:

```
outputPort Printer {
Location: "socket://192.168.1.25:8000/"
Protocol: sodep     Interfaces: PrinterIface }

inputPort AggregatorInput {
Location: "socket://www.company.com:80/"
Protocol: http     Aggregates: Printer }
```

We assume that the aggregator runs on a machine that has access to both the private network and to the Internet. Notice that the printer and the aggregator use a different protocol to be invoked (i.e. sodep for the printer and http for the aggregator). The Jolie interpreter will automatically

---

[1]The full version of the example can be retrieved at [3].
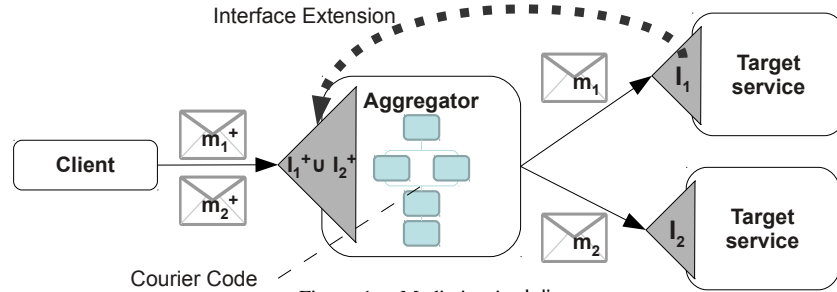
Figure 1. Mediation in Jolie

take care of converting the incoming HTTP messages into the SODEP protocol format. Hence, we can access the aggregator by means of a common web browser. Browsing the following URL would for instance invoke the print operation: http://www.company.com/print?doc=Hello. Now let us suppose that the access to the printer should be restricted by means of an authentication system already developed and deployed. This can be done allowing the aggregator to manipulate the messages before forwarding them to the printer using the smart aggregation [4][2]. Our aggregator code thus changes to the following:

```
type AuthRequest:void
  { .username:string .password:string }

type SessionId:void { .sid:string }

interface AuthenticatorIface {
RequestResponse:
  auth(AuthRequest)(AuthResponse)
    throws AuthFailed(void)
  checkSession(AuthRequest)(SessionId)
    throws InvalidSession(void)}

interface extender SessionIdExtender {
RequestResponse: *(SessionId)(void)
  throws InvalidSession(void)}

outputPort Authenticator {
Location: "socket://192.168.1.20:8080/"
Protocol: sodep    Interfaces: AuthenticatorIface }

outputPort Printer {
Location: "socket://192.168.1.25:8000/"
Protocol: sodep    Interfaces: PrinterIface }

inputPort AggregatorInput {
Location: "socket://www.company.com:80/"
Protocol: http
Aggregates: Authenticator,
  Printer with SessionIdExtender}

courier AggregatorInput {
[interface PrinterIface(request)(response)]
{ chk.sid = request.sid;
  checkSession@Authenticator(chk)(chkRes);
  forward(request)(response) } }
```

The aggregated authentication service Authenticator offers two operations, auth and checkSession. The operation auth allows a client to authenticate by sending a username and password: if the authentication succeeds the invoker receives back a session identifier, otherwise an AuthFailed

[2]Note that the legacy printer and authenticator services do not require modifications.

fault denoting invalid credentials is raised. The operation checkSession is used to check if a session identifier is valid. The session identifier is now required for accessing every operation of the printer service through the aggregator. Indeed the interface of the aggregator is extended by means of the operator SessionIdExtender with the additional information of type SessionId. When a message is received by the aggregator the (courier) code courier AggregatorInput is executed. Specifically, inside the courier code, every request for an operation in the PrinterIface interface is intercepted and its session identifier verified by calling the authenticator service. If the session identifier is found to be invalid, the generated InvalidSession fault from the authenticator will be automatically sent back to the client. Otherwise, the call is successfully forwarded to the printer through the forward primitive. Jolie will automatically take care of removing the additional authentication informations not expected by the printer service exploiting the information given by the interface extender SessionIdExtender.

Observe that our aggregator code is parametric on PrinterIface: if the latter changes, the interface extension will be simply reapplied to the new definition without requiring any change in the aggregator. Modifications to aggregated interfaces may be changing the set of exposed operations or their data types (provided that the new data fields do not have the same names as those introduced by the aggregator through extension). For instance, suppose that the printer service is upgraded to provide a new operation check_status and that the data type of operation print is changed to require also an options parameter. Then, the aggregator above could be simply redeployed as it is and continue working as intended.

The task of integrating different services, or applications in general, has been investigated for a long time in the context of practical, commercial tools using Enterprise Application Integration (EAI) frameworks [5]. Usually these frameworks are implemented by enhancing standard middleware products like the Enterprise Service Bus (ESB) [6]. On the market one can find several ESB mature products developed by leading IT companies like IBM [7], Oracle [8] and Microsoft [9]. However, contrary to what happens for WS-BPEL, there is no standard that establishes what an

ESB should require or provide. An ESB usually is provided with an ad hoc GUI that simplifies the development of the service integration, while avoiding the need to make sweeping changes to the existing applications or data structures. An ESB can be seen as an integration infrastructure that facilitates the interaction between different entities, in a possibly distributed system, through mediation. Service providers and service consumers use the ESB to interact with each other. Indeed the ESB is able to send requests and receive responses to and from the integrated services (invocation). The ESB also handles the message routing, by deciding the final destination of a message based, for example, on the content of the message. Thus, the ESB allows the developer to process messages, handle protocol transformations, filter messages, monitor or log activities, and implement security policies. Hence, an ESB acts as a proxy, implementing all the programmed mediation activities such as message routing and transformation. The proxy also exposes the functionalities of the service providers (possibly augmented with some extra parameters used in the mediation logic). In ESB tools, designers usually implement a mediator M by first (i) importing the interface I_S of the target service, (ii) write the interface I_M to be exposed to external invokers for interacting with S through M, and finally (iii) by creating the intermediate mediation logic through graphical tools that allow for the programming of simple workflows.

The crucial point that we want to observe here is that even if programmers usually start to define I_M by modifying I_S, the ESB tools does not track in any way that I_M is a modification of I_S. Moreover, in the mediation logic the programmer may refer to and manipulate any part of the messages received through I_M. For this reason updates or modifications of the target service S imply modifications of the mediator definition and implementation, thus making the ESB approach *target-dependent*.

Figure 2 provides a graphical representation of an ESB infrastructure: service consumers and service providers are connected to the bus that handles their interactions through the proxy.

Let us compare the process of adding (or modifying) a service provider to an ESB with the process of adding (or modifying) a service provider to an aggregator in Jolie. The operation chain has a lot of similarities. In both cases the first task is to import the interfaces of the service providers. This operation can be intuitively seen as creating a binding that allows the use of the services implementing the basic activities. In ESB this task is done by importing the WSDL description of the service providers. In Jolie instead the binding consists in the creation of output ports. Once the service providers have been imported we have the first and main difference between the ESB and Jolie approach. The second task in the ESB methodology consists in creating or modifying the proxy interface to support or modify existing operations, while in Jolie the interface does not need to

be created from scratch. Existing interfaces can be properly extended to arrange for the presence of new or modified operations. This small difference impacts heavily in the SOA design since being able to extend previous interfaces allows a developer to build an SOA in an incremental way, being more modular and less error prone since the number of repetitive coding tasks decreases. After the definition of the proxy interfaces there is a duality between the ESB and Jolie integration approaches. Indeed, while in the ESB approach before deploying the proxies the developer needs to program their behaviour defining how the messages are mediated, routed, enhanced or filtered, in Jolie the same could be done by defining the service courier code.

Even though ESBs are specifically tailored for service integration, and therefore well suited for the majority of service integration tasks, other approaches are still used. For instance, since the ESB development is a knowledge-intensive task (i.e., even if there exist visual tools to support the implementation phase, an ESB programmer requires a deep understanding of the many parameters provided by the common ESB solutions) and an ESB license is usually expensive , the service integration can be obtained using BPEL or a general-purpose language such as Java. In these cases the programmer may have a deeper control of the message flow, implement non-standard service behaviors, and develop faster and lighter service integration frameworks. However, this advantages comes at the price that a lot of code needs to be written in order to perform the same task that can be easily written using a domain specific language or tools.

Consequently when these approaches are used there is no direct support for service integration operations and therefore a features that allows or ease a developer to propagate the changes of an interface throughout the entire SOA is completely missing.

III. CONCLUSIONS

We have proposed a mechanism based on interface extensions that can be used to develop target-transparent service integration. When target-transparent mediators suit the application scenario, our methodology leads to a simpler and clearer definition of this kind of service integrators than ESB. We envision that this simplicity will improve the cost-effectiveness of the design of SOAs in the future. An initial implementation of this mechanism has been obtained extending the Jolie framework [4]. Through this extension, the methodology described in this paper has been used in production at italianaSoftware [10] to integrate medium-sized SOAs (ranging from about 10 to 40 services). Given our initial results, we deem that interface extension could be considered an interesting feature to add to current ESB frameworks, maybe exploiting the new extension features of WSDL 2.0 [11]. In this sense, the formal semantic specifications of the Jolie language for interface extension [4] would serve as guidelines to reobtain the same results.
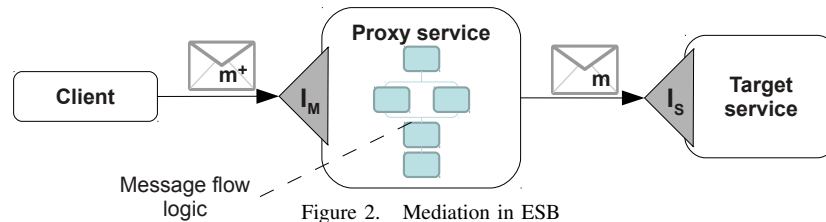
Figure 2. Mediation in ESB

The interface extension mechanism is particularly relevant when considering SOAs which deal with a continuously changing environment and therefore have to address variability aspects [12]. Indeed, also the recent 2.0 version of the WSDL language includes a form of interface extension. Our notion of interface extension however is more general: WSDL 2.0 for instance does not allow the possibility to add arguments to service invocations while in Jolie this is one of the basic building blocks for target transparency.

The lack of the service extension primitives in the ESB solutions prevent the design of complex EAI solutions where different layers of services depend exclusively on the functionalities of lower service levels. The possibility of being able to extend the services will allow the development of more modular and maintainable multi-tier solutions for EAI since with service extensions the propagation of a low level service modification is small or negligible.

Similarities to the approach presented in this paper can be found also with delta models [13], a technique developed for tackling the variability of Software Product Lines. A delta defines how existing code should be integrated, modified, or deleted to define the new features of a Software Product Line. Interface Extension and courier sessions can therefore be seen as a delta for Service-Oriented Computing.

Recently some papers have addressed the problem of a more general, dynamic variability or adaptation of SOAs, and some frameworks based on ESB technologies have been considered in order to address this issue, see for example [14], [15], [16]. We believe that our work on interface extension in Jolie, which can be seen as a static form of variability, can be considered as a first, beneficial step to obtain dynamic adaptable SOA, and we are planning to explore this direction of research in the future. Moreover, we are also investigating a new interface extension-like primitive for removing some operation fields, instead of adding new ones. While adding fields to inputs is usually considered a safe practice in a compositional setting [17], [18], removing fields is not since understanding when it is safe to remove operations, variables or methods is nontrivial [19].

From a more pragmatic point of view we would like to perform an extensive study to determine how much the use of interface extensions could speed up the development of an SOA or reduce its maintainability burden, and evaluate the performances and the scalability of our solution in Jolie with respect to behaviorally equivalent ESB solutions.

REFERENCES

[1] "Web Services Business Process Execution Language Version 2.0: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html."

[2] "JOLIE: Java Orchestration Language Interpreter Engine. Project Website: http://www.jolie-lang.org/."

[3] "Example implementation: http://www.itu.dk/people/fabr/papers/soca2012/example.zip."

[4] M. Dalla Preda, M. Gabbrielli, Claudio, Guidi, J. Mauro, and F. Montesi, "Interface-based service composition with aggregation," in *ESOCC*, 2012.

[5] M. H. Sherif, *Handbook of Enterprise Integration*. Auerbach Publishers, Incorporated, 2009.

[6] D. A. Chappell, *Enterprise Service Bus - Theory in practice*. O'Reilly, 2004.

[7] "WebSphere Process Server. Product Website: http://www-01.ibm.com/software/integration/wps/\#."

[8] "Oracle Enterprise Service Bus. Product Website: http://www.oracle.com/technetwork/middleware/service-bus/overview/index.html."

[9] "BizTalk Server. Product Website: http://www.microsoft.com/biztalk/en/us/default.aspx."

[10] "italianaSoftware Website: http://www.italianasoftware.com."

[11] "Web Services Description Language (WSDL) Version 2.0: http://www.w3.org/TR/wsdl20/."

[12] H. J. La, J. S. Bae, S. H. Chang, and S. D. Kim, "Practical Methods for Adapting Services Using Enterprise Service Bus," in *ICWE*, pp. 53–58, 2007.

[13] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-Oriented Programming of Software Product Lines," in *SPLC*, pp. 77–91, 2010.

[14] L. González and R. Ruggia, "Towards dynamic adaptation within an ESB-based service infrastructure layer," in *MONA '10*, pp. 40–47, 2010.

[15] Y. Liu, M. A. Babar, and I. Gorton, "Middleware architecture evaluation for dependable self-managing systems," in *QoSA '08*, pp. 189–204, 2008.

[16] S. H. Chang, H. J. La, J. S. Bae, W. Y. Jeon, and S. D. Kim, "Design of a Dynamic Composition Handler for ESB-based Services," in *ICEBE*, pp. 287–294, 2007.

[17] B. C. Pierce, *Types and Programming Languages*. MA, USA: MIT Press, 2002.

[18] S. Gay and M. Hole, "Subtyping for session types in the pi calculus," *Acta Informatica*, vol. 42, pp. 191–225, Nov. 2005.

[19] M. Carbone and F. Montesi, "Deadlock-freedom-by-design: Multiparty asynchronous global programming," in *POPL, to appear*, 2013.