Middleware-layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability

Valérie Issarny[1], Amel Bennaceur[1], and Yérom-David Bromberg[2]

[1] INRIA, CRI Paris-Rocquencourt, France
[2] LaBRI, University of Bordeaux, France

Abstract. This chapter deals with interoperability among pervasive networked systems, in particular accounting for the heterogeneity of protocols from the application down to the middleware layer, which is mandatory for today's and even more for tomorrow's open and highly heterogeneous networks. The chapter then surveys existing approaches to middleware interoperability, further providing a formal specification so as to allow for rigorous characterization and assessment. In general, existing approaches fail to address interoperability required by today's ubiquitous and heterogeneous networking environments where interaction protocols run by networked systems need to be mediated at both application and middleware layers. To meet such a goal, this chapter introduces the approach that is investigated within the CONNECT project and that deals with the dynamic synthesis of *emergent connectors* that mediate the interaction protocols executed by the networked systems.

Keywords: Interoperability, Middleware, Pervasive networking, Protocol mediation

1 Introduction

As networked systems are becoming increasingly pervasive, they need to compose dynamically with their ever evolving environment according to functionalities they provide and/or request. However, such dynamic composition is greatly challenged by the heterogeneity and autonomy of today's digital systems, which are not designed in concert, but are instead independently developed and deployed within pervasive networking environments. As a result, although networked systems may possibly match from the standpoint of provided and required functionalities, actual behavioral matching is unlikely due to inherent design diversity. Therefore, what is needed for enabling the composition of pervasive networked systems is *emergent connectors* [28], which embed a mediation process so as to adapt the systems' respective interaction behaviors for the sake of coordination.

The notion of *mediator* underlying emergent connectors is not new. It has indeed been investigated since the need for interoperability in distributed systems was identified [23]. However, this was initially a design-time concern, while today's dynamic distributed systems require on-the-fly mediation. On-the-fly protocol mediation has in particular been studied quite extensively in the context of Web services to deal with either dynamic service composition (e.g., [14]) or substitution (e.g., [12]). Still, as in particular investigated in the companion chapter on application-layer connector synthesis [26], existing work on runtime automated mediation concentrates on application-layer protocols, while the heterogeneity of open networked systems may concern both the application and middleware layers.

As surveyed within companion chapter on interoperability in complex distributed systems [6], middleware interoperability solutions have been developed since the early days of middleware. While one-to-one bridging was among the early approaches [40], it evolved into more generic solutions such as Enterprise Service Bus [13], interoperability platforms [21] and transparent interoperability approaches [9, 36]. However, except for the transparent interoperability approaches, most of these solutions rely upon the design-time choice to develop applications using the proposed interoperability solution. Thus, they do not allow for on-the-fly interoperability between networked applications embedding different legacy middleware. Middleware interoperability further needs to cope with the many middleware interaction paradigms that now need to coexist. This includes accessing the same functionality through distinct paradigms (e.g., context-awareness through access to a data-centric sensor network or an RPC-based context server).

As an illustration, consider the simple, yet challenging scenario of photo sharing within a public space such as a stadium, which is also investigated from the standpoint of application-layer connection in [26]. Typically, the target environment allows for both infrastructure-based and ad hoc peer-to-peer photo sharing. In the former implementation, a photo sharing service is provided by the stadium, where only authenticated photographers are able to produce pictures while any spectator may download and even annotate pictures. The peer-to-peer implementation allows for photo download, upload and annotation by any spectator, who are then able to directly share pictures using their handhelds. In both cases, the spectator's handheld would need to embed the appropriate software application, which may not be available due to the handheld's specific platform. Further, the spectator may not be willing to download yet another photo sharing application, i.e., the proprietary implementation offered by the stadium, while one is already available on the handheld. Moreover, while the photo sharing functionality is present in both versions of the photo sharing application, it is unlikely that they feature the very same interface and behavior. In particular, the RPC interaction paradigm suits quite well the infrastructure-based service, while a distributed shared data space is more appropriate for the peer-to-peer version. In general, considering the ever-growing base of content-sharing applications for handhelds, numerous versions of the photo sharing application may be available on the spectators' handhelds, thus calling for appropriate interoperability solutions that mediate interaction protocols from the application down to the middleware layer.

This chapter more specifically concentrates on middleware-layer interoperability, i.e., enabling networked systems that functionally match to be able to coordinate despite running heterogeneous middleware protocols. The next section formalizes the role of middleware in the connection of networked systems, in particular highlighting the inter-play between application- and middleware-layer protocols. Then, Section 3 focuses on *interoperability connectors* introduced in the literature, as surveyed in companion chapter [6]; the behavior of interoperability connectors is formally defined, hence providing a rigorous characterization of their respective features. As presented, interoperability connectors allow overcoming the heterogeneity of middleware protocols as long as the protocols implement the same coordination paradigm, which is too restrictive regarding the objective of enabling emergent connectors. Section 4 paves the way for enabling emergent connectors, i.e., the on-the-fly synthesis of connectors that mediate interaction protocols from the application down to the middleware layer, which builds upon the theory of mediators presented in companion chapter [26]. Finally, Section 5 concludes with perspective for future work towards effecting emergent middleware.

2 Middleware-based Connectors

In the context of distributed systems, a connector abstracts a complex interaction behavior that is facilitated by middleware which provides services to realize this interaction. In particular, middleware overcomes the heterogeneity of the distributed infrastructure by establishing a software layer that homogenizes the infrastructure's diversities using a well-defined and structured distributed programming model [27]. In particular, middleware induces an interaction paradigm for enabling distributed networked systems to coordinate [38].

In the following, we introduce middleware-based connectors using state-of-the-art connector classification [34, 50] (Section 2.1) and define their specification using formal notation(Section 2.2). Then, we describe the mismatches preventing connection among components and introduce the needed mediation to enforce interoperability among them.

2.1 A Classification of Middleware-based Connectors

Based upon the classification of connectors introduced in [34, 50], services provided by middleware are depicted in Figure 1. The *communication* and *coordination* services support the transfer of data and control among components and can be realized by different connector types, each of which defining an *interaction paradigm* such as *procedure call*, *event*, *message-based*, or *data access* connectors. The *adaptor* connector type provides a *conversion* service to support interaction among heterogeneous components while services that *facilitate* interaction among components are achieved using the *distributor* and *arbitrator* connector types. Distributor connectors perform discovery through the identification of interaction paths and subsequent routing of communication and coordination

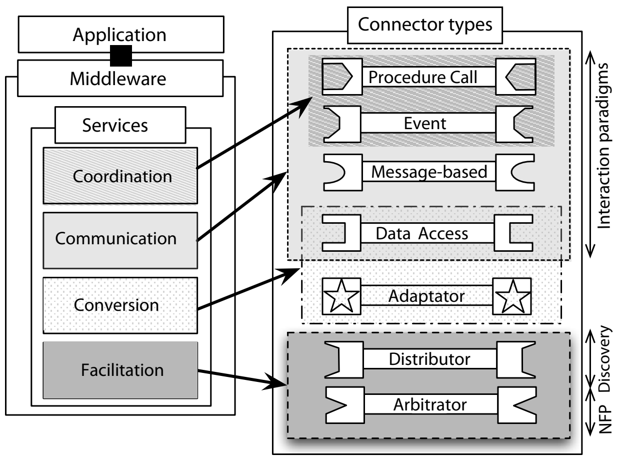

Fig. 1. Middleware-based connector classification

information among components along these paths. Non-functional properties (NFP) are managed by *arbitrator* connectors that streamline system operations, resolve any conflict and redirect the flow of control.

Each connector type is associated with different dimensions (and subdimensions) representing its architectural details. For example, a procedure call connector defines the *Parameters* dimension that is subdivided into *data transfer*, *semantics*, *return value*, and *invocation record* subdimensions. The procedure call connector type is also associated to other dimensions such as *Entry point* associated to two subdimensions, *single* or *multiple*, *Invocation* defining the *implicit* and *explicit* subdimensions, *Synchronicity*, *Cardinality*, and *Accessibility*. The values associated to the various dimensions and subdimensions define a connector implementation, that is, a specific middleware. For example, SOAP[3] (Simple Object Access Protocol), CORBA[4] (Common Object Request Broker Architecture), and RMI[5] (Remote Method Invocation) are specific middleware defining implementations of the procedure call connector type.

2.2 Formalizing Middleware-based Connectors

In order to precisely characterize the role of middleware in the connection of networked systems, this section formalizes middleware-based connectors using FSP [33], as FSP has proven to be a convenient formalism for specifying connectors [47]. In particular, using FSP allows us to exploit the LTSA tool [33] to automate reasoning about the behavior of connectors and connected systems.

[3] http://www.w3.org/TR/soap/
[4] http://www.omg.org/technology/documents/corba_spec_catalog.htm
[5] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

Definitions	
END	Predefined process, denotes the state in which a process successfully terminates
set S	Defines a set of action labels
$[i : S]$	Binds the variable i to a value from S
Primitive Processes (P)	
$a \to P$	Action prefix
$a \to P \| b \to P$	Choice
$P; Q$	Sequential composition
$P(X =' a)$	Parameterized process: P is described using parameter X and modeled for a particular parameter value, $P(a1)$
$P/\{new\_1/old\_1, ..., new\_n/old\_n\}$	Relabeling
$P \backslash \{a_1, a_2, ..., a_n\}$	Hiding
$P + \{a_1, a_2, ..., a_n\}$	Alphabet extension
Composite Processes ($\|P$)	
$P \| Q$	Parallel composition
forall $[i : 1..n]$ $P(i)$	Replicator construct: equivalent to the parallel composition $(P(1)\|...\|P(n))$.
$a : P$	Process labeling

Table 1. FSP syntax overview

FSP notations and semantics. Table 1 provides an overview of the FSP operators, while the interested reader is referred to [33] for further detail. Briefly stated, FSP processes describe actions (events) that occur in sequence, and choices between event sequences. Each process has an alphabet of the events that it is aware of (and either engages in or refuses to engage in). There are two types of processes: *primitive processes* and *composite processes*. Primitive processes are constructed through action prefix, choice, and sequential composition. Composite processes are constructed using parallel composition or process relabeling. When composed in parallel, processes synchronize on shared events: if processes P and Q are composed in parallel as $P||Q$, events that are in the alphabet of only one of the two processes can occur independently of the other process, but an event that is in the alphabets of both processes cannot occur until the two of them are willing to engage in it. The replicator forall is a convenient syntactic construct used to specify parallel composition over a set of processes. Processes can optionally be parameterized and have re-labeling, hiding or extension over their alphabet. A composite process is distinguished from a primitive process by prefixing its definition with $\|$.

A formalization of connectors. According to [1], a connector is defined by a set of *roles* and a *glue* where:

– *roles* (See Figure 2, ❶) specify the expected local behavior of each of the interacting parties.

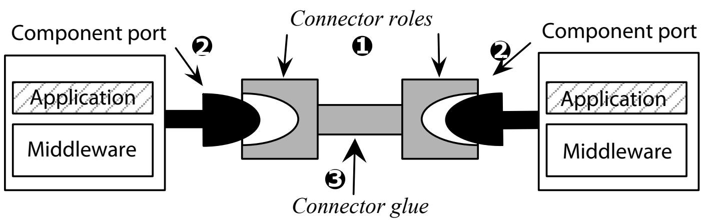

Fig. 2. Components & Connector

– *glue* (See Figure 2, ❸) specifies how the behaviors of these parties are coordinated.

In addition, the interaction protocols of components are specified by *ports* (See Figure 2, ❷).

Then according to [47], roles, glues and ports are specified as FSP processes, which allows assessing architectural matching and thus interoperability. Specifically, a component can be attached to a connector only if its port is *behaviorally compatible* with the connector role it is bound to. Allen and Garlan [1] define behavioral compatibility between a component port and a connector role based on the notion of refinement. Informally, a component port is behaviorally compatible with a connector role if the process specifying the behavior of the former refines the process characterizing the latter. In other words, it should be possible to substitute the role process by the port process.

In our case, we are further interested in characterizing interaction protocols at both application and middleware layers since both of them are sources of heterogeneity. We then define the behavior of a connector as a hierarchical protocol that specifies the behavior of the application-layer interaction protocol in terms of middleware-specific protocols. Building on the work of [47], the behavior of a middleware-layer connector is specified as a parallel FSP process composing: (i) one process for each role of the connector, and (ii) one process for the glue that describes how all roles are bound together. The application-specific behavior is further specified as a process over role processes of the underlying middleware-layer connector.

Example. As an illustration, we have the following FSP-based specification of a SOAP-based connector:

1 Role Client_{SOAP} $= \mathit{SOAP\text{-}RPCCall} \rightarrow \mathit{SOAP\text{-}RPCReceiveReply} \rightarrow \mathsf{Client}_{SOAP}$
2 Role Server_{SOAP} $= \mathit{SOAP\text{-}RPCReceiveCall} \rightarrow \mathit{SOAP\text{-}RPCReply} \rightarrow \mathsf{Server}_{SOAP}$
3 Glue_{SOAP} $= \mathit{SOAP\text{-}RPCCall} \rightarrow \mathit{SOAP\text{-}RPCReceiveCall} \rightarrow \mathsf{Glue}_{SOAP}$
4 $\mid \mathit{SOAP\text{-}RPCReply} \rightarrow \mathit{SOAP\text{-}RPCReceiveReply} \rightarrow \mathsf{Glue}_{SOAP}$
5 $\|\mathsf{Connector}_{SOAP} = \mathsf{Client}_{SOAP} \| \ \mathsf{Glue}_{SOAP} \| \ \mathsf{Server}_{SOAP}$

According to the specification, Client_{SOAP} (Line 1) initiates a request using $\mathit{SOAP\text{-}RPCCall}$, and gets a response through $\mathit{SOAP\text{-}RPCReceiveReply}$. When

Server$_{SOAP}$ (Line 2) gets a request *SOAP-RPCReceiveCall*, it initiates a response *SOAP-RPCReply*. The Glue$_{SOAP}$ coordinates the interaction of the two roles (Lines 3 and 4): a *SOAP-RPCCall* from the Client$_{SOAP}$ is followed by a *SOAP-RPCReceiveCall* to the Server$_{SOAP}$, and a *SOAP-RPCReply* from the Server$_{SOAP}$ is followed by a *SOAP-RPCReceiveReply* to the Client$_{SOAP}$.

Then, different application-layer protocols may be specified using the provided middleware connector. For instance, consider the *Photo Sharing* example discussed in the introduction, Figure 3 gives the FSP specification of the *RPC-SOAP* implementation of infrastructure-based photo sharing. First, we define the SOAP actions that can be performed by the networked systems (Line 1). The behavior of the photo sharing consumer (Lines 3 to 5), producer (Lines 6 to 7), and server (Lines 8 to 12) are specified using this provided set of actions. The photo sharing producer and consumer invoke actions using the Client$_{SOAP}$ process (Lines 14 to 15) while the photo sharing server provides actions using the Server$_{SOAP}$ process(Lines 16 to 17). The Glue$_{SOAP}$ ensures the coordination among all the actions (Lines 18 to 20).

```
 1 //Infrastructure-bade application specification
 2 set SOAP_PhotoSharing_Actions = {uploadPhoto, searchPhoto, downloadPhoto, downloadComment, commentPhoto}
 3 PhotoSharingConsumer          = (req.searchPhoto →P1),
 4 P1                            = (req.downloadPhoto →P1|req.commentPhoto →P1
 5                                 |req.downloadComment →P1 |terminate →END).
 6 PhotoSharingProducer          = (req.uploadPhoto →PhotoSharingProducer
 7                                 |terminate →END).
 8 PhotoSharingServer            = (prov.uploadPhoto →PhotoSharingServer
 9                                 |prov.searchPhoto →PhotoSharingServer
10                                 |prov.downloadPhoto →PhotoSharingServer
11                                 |prov.commentPhoto →PhotoSharingServer
12                                 |prov.downloadComment →PhotoSharingServer|terminate →END).
13 //SOAP middleware Specification
14 Client_SOAP(X =' op)          = (req.[X] →P1|terminate →END),
15 P1                            = (SOAP-RPCCall[X] → SOAP-RPCReceiveReply[X] →Client_SOAP).
16 Server_SOAP(X =' op)          = (prov.[X] →P2 |terminate →END),
17 P2                            = (SOAP-RPCReceiveCall[X] → SOAP-RPCReply[X] →Server_SOAP).
18 Glue_SOAP(X =' op)            = (SOAP-RPCCall[X] →P0 |terminate →END),
19 P0                            = (SOAP-RPCReceiveCall[X] → SOAP-RPCReply[X]
20                                 → SOAP-RPCReceiveReply[X] →Glue_SOAP).
21 //System specification
22 ||SOAP_PhotoSharing           = (PhotoSharingProducer
23                                 ||PhotoSharingConsumer
24                                 ||PhotoSharingServer
25                                 ||(forall [op:SOAP_PhotoSharing_Actions] Server_SOAP(op))
26                                 ||(forall [op:SOAP_PhotoSharing_Actions] Client_SOAP(op))
27                                 ||(forall [op:SOAP_PhotoSharing_Actions] Glue_SOAP(op))).
```

Fig. 3. Infrastructure-based photo sharing

2.3 Connection Mismatches and Related Mediation

Connection mismatches result from different assumptions that components make about connection. Blair et al. [6] define several heterogeneity dimensions generating mismatches (see Figure 4):

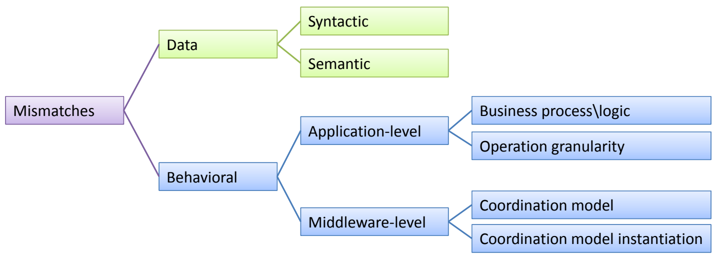

Fig. 4. Classifying mismatches

– *Data heterogeneity:* Networked systems associate different representations (syntax) and meanings (semantics) to their data, which may results in data inconsistencies. Middleware coupled with ontologies play a valuable role in solving both the syntactic and semantic mismatches.
– *Behavioral middleware-level heterogeneity:* While middleware ensures interoperability across languages and network platforms, it only does so for systems using the same middleware. Indeed, a middleware implementation involves a style of interaction by specifying a coordination model and the associated protocol and data format. As a result, systems using different middleware are not able to interoperate.
– *Behavioral application-level heterogeneity:* Different systems may have incompatible business-process logic and disparate interface signatures (e.g., see [26]).

A further dimension of heterogeneity is related to the handling of non-functional properties, which we do not address in this chapter.

In general, networked systems may be connected only in the absence of all of the above heterogeneity dimensions, i.e., networked systems should be behaviorally compatible from application down to middleware layer, and further exchange semantically and syntactically matching data.

However, with networked systems getting increasingly pervasive, one would like to be able to connect networked systems that *semantically match*, despite heterogeneity in the above dimensions. By *semantic matching* [41], we mean that networked systems share a complementary high level goal towards which they need to coordinate although they may possibly run heterogeneous interaction protocols from the application down to the middleware layer.

Still considering the photo sharing example, both the infrastructure-based (see Figure 5A) and the peer-to-peer-based (see Figure 5B) versions of the photo sharing may be implemented over SOAP. Even though, the two systems semantically match and behaviorally match at the middleware-layer, they are not able to interact due to behavioral mismatches at the application layer.

Similarly, the infrastructure-based version of photo sharing may be implemented using two different middleware, such as SOAP (see Figure 6A) and RMI(see Figure 6B). In this case, middleware-layer mismatches prevent the two

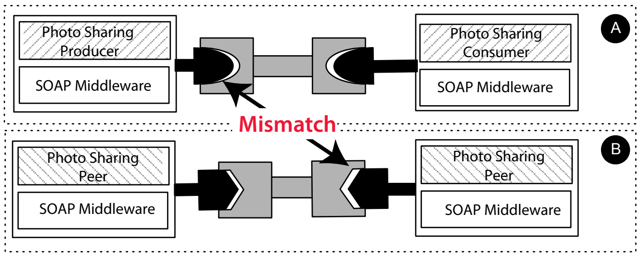

Fig. 5. Application mismatches

networked systems from interacting despite semantic matching and behavioral matching at the application-layer.

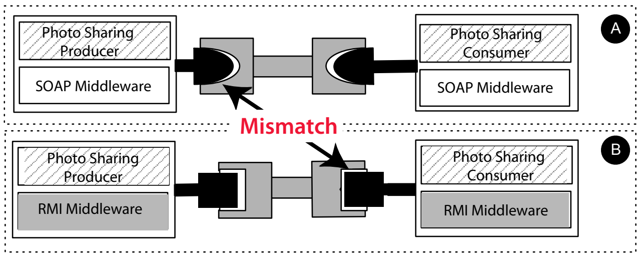

Fig. 6. Middleware mismatches

Under semantic matching of two networked systems, behavioral matchmaking is achieved through the generation of *mediators* that enforce the behavioral compatibility of the networked systems (see Figure 7). The resulting system is called *connected system*.

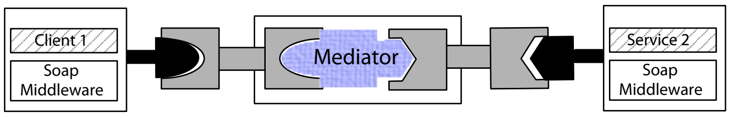

Fig. 7. Connected system

Since mismatches take place at different inter-related layers, mediation becomes a cross-cutting concern that has to be achieved in conjunction at the different system layers, from application down to middleware down to network (see Figure 8). At each layer, many facets (data, interface, and behavior) of heterogeneity should be dealt with. There is a number of existing mediation solutions, each of which solves mismatches related either to applications or to middleware. Indeed, solutions addressing application heterogeneity assume the same middleware whereas solutions achieving middleware interoperability consider the same application atop of it. However, all the dimensions of heterogeneity should be simultaneously addressed in order to guarantee effective interoperability among heterogeneous systems.

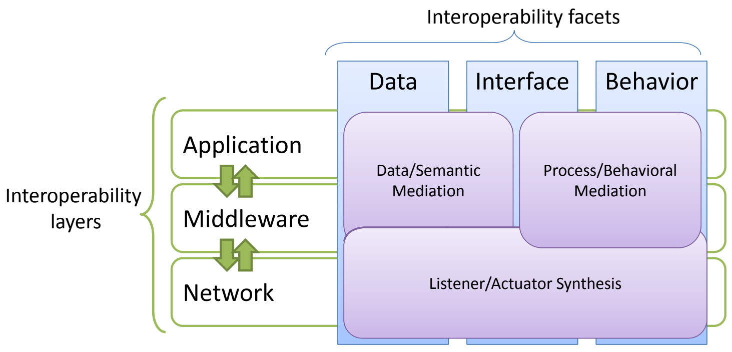

Fig. 8. Mediation

In this chapter, we more specifically concentrate on middleware-layer protocol mediation and its relation with application-layer mediation. As a first step, the next section reviews state-of-the-art solutions to middleware interoperability that is in particular surveyed in [6]. We qualify such solutions as *interoperability connectors*. However, these solutions primarily deal with middleware-level heterogeneity, further assuming connection between components relying on the same interaction paradigm. Section 4 then introduces the solution investigated within the CONNECT project that aims at overcoming both application- and middleware-level heterogeneity, including heterogeneity in the interaction paradigms.

3 Interoperability Connectors

State-of-the-art solutions to interoperability between heterogeneous middleware primarily concentrate on middleware implementing the same interaction paradigm and subdivide into the following categories: *software bridge*, *interoperability platform* and *transparent interoperability* [6]. We review each approach in turn, providing their FSP-based semantics so as to precisely characterize their respective features and further allow for thorough assessment and comparison. In addition, we point out exploitation of the proposed interoperability connectors at the application layer.

3.1 Software Bridges

Bridging assumes *a priori* knowledge of both applications and middleware that have to be made interoperable without any intervention in their code. Particularly, bridging provides a mapping between various interaction protocols. Such a mapping can be either $1 \to 1$, which is *direct bridging*; or $n \to 1 \to m$, which is *indirect bridging*.

Direct bridging. The principle of *direct bridging* is to transform one of the connector roles according the incompatible connector role, as illustrated in Figure 9.

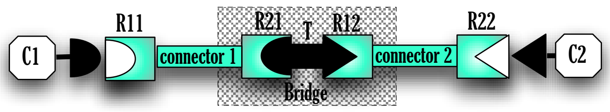

Fig. 9. Direct bridge

Formally, direct bridging is performed as follows (see Figure 10):

1. The glue of each connector is first tagged in order to avoid unwanted event synchronization (tag_1 :Glue_1 and tag_2 :Glue_2, Line 11),
2. A set of predefined transformations, T (Line 7), is applied to the connectors in order to adapt their respective behaviors,
3. The transformations are chained with the tagged glues through the Bridge (Line 5) process.

According to the above, the *direct bridge mediator* specification is defined as:

$$||\mathsf{Direct\_Bridge\_Mediator} = (\mathsf{Bridge}||\mathsf{T}).$$

The developer must thus ensure the correctness of T, in particular ensuring that the bridging actually performs the required mediation without introducing any error. Moreover, a direct bridge must be developed separately for every pair of protocols between which interaction is required. Hence, ensuring interoperability between each pair of n components requires developing $n(n-1)$ mediators. The diversity of protocols that are used in today's networked systems implies that this is a substantial development task.

Practically, middleware direct bridges, such as OrbixCOMet[6] and SOAP2-CORBA[7], ensure interoperability between two fixed middleware implementations (DCOM-CORBA and SOAP-CORBA respectively). Similarly, application software bridges may be introduced to define bridging between application-specific protocols (i.e., overcoming application- and middleware-layer protocols

[6] http://www.iona.com/support/whitepapers/ocomet-wp.pdf
[7] http://soap2corba.sourceforge.net/

```
//Specification of Connector$_1$ & Connector$_2$
1 Role R1$_{i,i\in[1..2]}$ = Specification of Role R1 of Connector$_i$
2 Role R2$_{i,i\in[1..2]}$ = Specification of Role R2 of Connector$_i$
3 Glue$_{i,i\in[1..2]}$     = Specification of the glue of Connector$_i$
4 set I$_{i,i\in[1..2]}$    = Set of events initiated from Role R1$_i$ and R2$_i$
5 Bridge                    = $tag_1.[e_1 : I_1] \rightarrow tag_2.[e_1] \rightarrow$Bridge$|tag_2.[e_2 : I_2] \rightarrow tag_1.[e_2] \rightarrow$Bridge

6 //Specification of the adaptation process
7 T                         = Specification of the required transformations to bridge Connector$_1$ to Connector$_2$

8 //Specification of the direct bridge connector
9 ||C-DBridge               = R1$_1 \| tag_1$ :Glue$_1\|$Bridge $\|$T$\| tag_2$ :Glue$_2 \| R2_2$
```

Fig. 10. Direct bridging specification

heterogeneity). However, implementing a bridge between two networked applications becomes very complex due to the domain-specific and technical knowledge required to realize the mediation.

Indirect bridging. Indirect bridging reduces the development effort associated with software bridges by introducing a common fixed intermediary protocol. This intermediary protocol is represented as a dedicated connector called `Connector`$_{bus}$ (see Figure 11). Then, interoperability is achieved in two steps: (i) the given native middleware protocol taken among n middleware is translated into a common intermediary protocol, (ii) the common intermediary protocol is then translated into the other given native middleware protocol taken among m middleware.

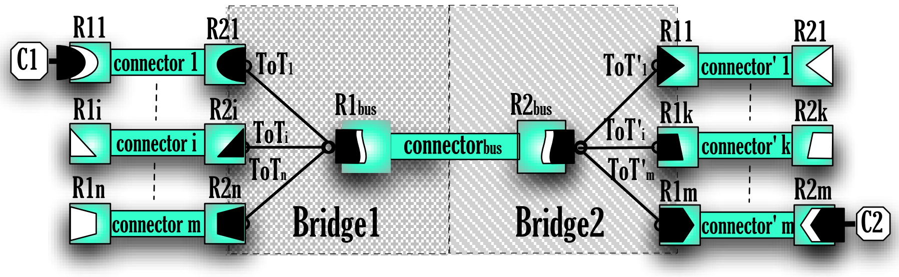

Fig. 11. Indirect bridge

Formally, indirect bridging performs translations back and forth using direct bridges in two steps (see Figure 12):

1. `Connector`$_i$ to `Connector`$_{bus}$ direct bridging through the use of processes $ToT_i \|$Bridge$_i$ (Lines 19 and 26) ($i \in [1..n]$),

2. $\texttt{Connector}_{bus}$ to $\texttt{Connector}'_k$ direct bridging through the use of processes $ToT'_k \| \mathsf{Bridge}'_k$ (lines 22 and 29) ($k \in [1..m]$).

The *indirect bridge mediator* is then specified as:

$$\|\mathsf{Indirect\_Bridge\_Mediator} = (\mathsf{T}_1 \| \mathsf{Bridge}_1 \| \mathsf{Bridge}_2 \| \mathsf{T}_2).$$

```
   //$\texttt{Connector}_{bus}$ specification
 1 Role $\mathsf{R1}_{bus}$        = Specification of Role R1 of $\texttt{Connector}_{bus}$
 2 Role $\mathsf{R2}_{bus}$        = Specification of Role R2 of $\texttt{Connector}_{bus}$
 3 $\mathsf{Glue}_{bus}$           = Specification that describes interactions between Role $\mathsf{R1}_{bus}$ and Role $\mathsf{R2}_{bus}$

 4 //Connectors specification
 5 Role R1                    = $|_{i=1}^{n}(a.glue_i \to \mathsf{R1}_i)$,
 6 $\mathsf{R1}_{i,i\in[1\cdot\cdot n]}$        = $\mathsf{R1}_i$ initial specification as given by $\texttt{Connector}_i | reset \to \mathsf{R1}$
 7 Role R2                    = $|_{k=1}^{m}(b.glue'_k \to \mathsf{R2}_k)$,
 8 $\mathsf{R2}_{k,k\in[1\cdot\cdot m]}$        = $\mathsf{R2}_k$ initial specification as given by $\texttt{Connector}'_k | reset \to \mathsf{R2}$
 9 $\mathsf{Glue}_{i,i\in[1\cdot\cdot n]}$      = Specification that describes interactions between
10                              Roles $\mathsf{R1}_i$ and $\mathsf{R2}_i$
11 $\mathsf{Glue}'_{k,k\in[1\cdot\cdot m]}$     = Specification that describes interactions between
12                              Role $\mathsf{R'1}_k$ and Role $\mathsf{R'2}_k$

13 //Set of events initiated or observed
14 set $\mathsf{I1}_{i,i\in[1\cdot\cdot n]}$    = Set of events initiated from Role $\mathsf{R1}_i$
15 set $\mathsf{O1}_{i,i\in[1\cdot\cdot n]}$    = Set of events observed from Role $\mathsf{R1}_i$
16 set $\mathsf{I2}_{k,k\in[1\cdot\cdot m]}$    = Set of events initiated from Role $\mathsf{R'2}_k$
17 set $\mathsf{O2}_{k,k\in[1\cdot\cdot m]}$    = Set of events observed from Role $\mathsf{R'2}_k$

18 //Specification of the adaptation processes
19 $\mathsf{T}_1$                     = $|_{i=1}^{n}(a.glue_i \to \mathsf{ToT}_i)$,
20 $\mathsf{ToT}_{i,i\in[1\cdot\cdot n]}$       = Specification of the required transformations to bridge $\texttt{Connector}_i$ to $\texttt{Connector}_{bus}$
21                            | $a.reset \to \mathsf{T}_1$
22 $\mathsf{T}_2$                     = $|_{k=1}^{m}(b.glue'_k \to \mathsf{ToT}'_k)$,
23 $\mathsf{ToT}'_{k,k\in[1\cdot\cdot m]}$      = Specification of the required transformations to bridge$\texttt{Connector}_{bus}$ to $\texttt{Connector}'_k$
24                            | $b.reset \to \mathsf{T}_2$

25 //Specification of the bridging processes
26 $\mathsf{Bridge}_1$                = $|_{i=1}^{n}(a.glue_i \to \mathsf{Bridge}_i)$,
27 $\mathsf{Bridge}_{i,i\in[1\cdot\cdot n]}$    = $[e : \mathsf{I1}_i] \to a.tag_i.[e] \to \mathsf{Bridge}_i | a.tag_i.[e : \mathsf{O1}_i] \to [e] \to \mathsf{Bridge}_i$
28                            | $a.reset \to \mathsf{Bridge}_1$
29 $\mathsf{Bridge}_2$                = $|_{k=1}^{m}(b.glue'_k \to \mathsf{Bridge}'_k)$,
30 $\mathsf{Bridge}'_{k,k\in[1\cdot\cdot m]}$   = $[e : \mathsf{I2}_k] \to b.tag_k.[e] \to \mathsf{Bridge}'_k | b.tag_i.[e : \mathsf{O2}_k] \to [e] \to \mathsf{Bridge}'_k$
31                            | $b.reset \to \mathsf{Bridge}_2$

32 //Specification of the indirect bridge connector
33 ||C-IBridge                = $\mathsf{R1} \| \mathsf{T}_1 \|_{i=1}^{n} a.tag_i : \mathsf{Glue}_i \| \mathsf{Bridge}_1 \| \mathsf{Glue}_{bus} \| \mathsf{Bridge}_2 \|_{k=1}^{m} b.tag_k : \mathsf{Glue}'_k \| \mathsf{T}_2 \| \mathsf{R2}$
```

Fig. 12. Indirect bridging specification

Practically, there exist various implementations of indirect bridges such as Enterprise Service Buses (e.g., ARTIX[8]) and MUSDAC [43]. Especially, Enterprise Service Buses (ESBs) have received a lot of attention. An ESB [35] is an

[8] http://web.progress.com/en/sonic/artix-index.html

open standards, message-based, distributed integration infrastructure that provides routing, invocation and mediation services to facilitate the interactions of disparate distributed applications and services.

Compared to direct bridging that requires $n \times m$ direct bridges to allow n components to interact with m, indirect bridging reduces the development effort since $n + m$ bridges have to be manually developed. Nevertheless, it limits the expressiveness of protocols, as some aspects of the relevant protocols may not be compatible with the chosen intermediary protocol.

3.2 Interoperability Platform

To overcome the static nature of software bridging, new approaches that dynamically select the best middleware bridge at a given time and place have emerged. Such solutions, called thereafter *interoperability platforms*, enable networked systems to switch their interaction protocol on-the-fly according to their environment. The principle is to provide a custom interface that abstracts the different interaction protocols used in the environment (see Figure 13).

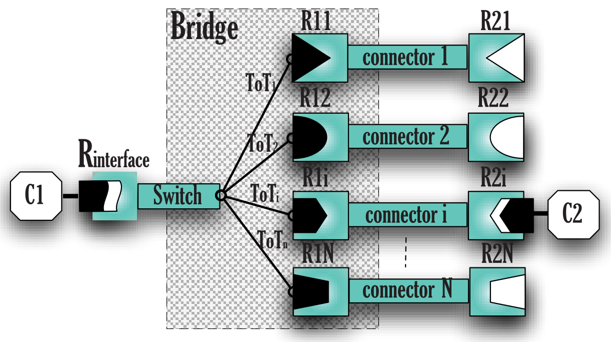

Fig. 13. Interoperability platform

Formally, interoperability is ensured in the following steps (see Figure 14):

1. The common interface, that has to be used by any component to interoperate with its environment is formally specified by a role $R_{interface}$ (Line 2),
2. The Switch process (Line 12) selects the appropriate connector $\texttt{Connector}_i$ among n according to the requirements of the environment,
3. The translation between $R_{interface}$ and $\texttt{Connector}_i$ is achieved in a way similar to direct bridging (Lines 14 to 21).

This leads to the following specification of the *interoperability mediator*:

$$\|\mathsf{Interoperability\_Mediator} = (\mathsf{Switch}\|\mathsf{T}\|\mathsf{Bridge}).$$

```
$//Proprietary\ interface$
Role $R_{interface}$ $= Specification\ of\ the\ bridge\ interface$
Role R2 $= |_{i=1}^{n}(glue_i \rightarrow \mathsf{R2}_i),$
$\mathsf{R2}_{i,i\in[1\cdots n]}$ $= Initial\ specification\ of\ the$ Role R2 of $\mathtt{Connector}_i | reset \rightarrow \mathsf{R2}$
$\mathsf{Glue}_{i,i\in[1\cdots n]}$ $= Specification\ of\ the\ glue\ of$ $\mathtt{Connector}_i$
$//Set\ of\ events\ initiated\ or\ observed$
set $\mathsf{I2}_{i,i\in[1\cdots n]}$ $= Set\ of\ events\ initiated\ from$ Role $\mathsf{R2}_i$
set $\mathsf{O2}_{i,i\in[1\cdots n]}$ $= Set\ of\ events\ observed\ from$ Role $\mathsf{R2}_i$
set $\mathsf{I}_{interface}$ $= Set\ of\ events\ initiated\ from$ Role $\mathsf{R}_{interface}$
set $\mathsf{O}_{interface}$ $= Set\ of\ events\ observed\ from$ Role $\mathsf{R}_{interface}$

$//Switch\ process$
Switch $= (election \rightarrow reset \rightarrow \mathsf{Switch}\ |_{i=1}^{n} election \rightarrow glue_i \rightarrow \mathsf{Switch})\backslash\{election\}$

$//Specification\ of\ the\ adaptation\ process$
T $= |_{i=1}^{n}(glue_i \rightarrow \mathsf{ToT}_i),$
$\mathsf{ToT}_{i,i\in[1\cdots n]}$ $= Specification\ of\ the\ required\ transformations\ to\ bridge$ $\mathsf{R}_{interface}$ $to$ $\mathtt{Connector}_i$
                              $|\ reset \rightarrow \mathsf{T}$

$//Specification\ of\ the\ bridging\ process$
Bridge $= |_{i=1}^{n}(glue_i \rightarrow \mathsf{Bridge}_i),$
$\mathsf{Bridge}_{i,i\in[1\cdots n]}$ $= [e : \mathsf{R}_{interface}] \rightarrow tag_i.[e] \rightarrow \mathsf{Bridge}_i | tag_i.[e : \mathsf{O}_{interface}] \rightarrow [e] \rightarrow \mathsf{Bridge}_i$
                              $|\ [e : \mathsf{I2}_i] \rightarrow tag_i.[e] \rightarrow \mathsf{Bridge}_i | tag_i.[e : \mathsf{O2}_i] \rightarrow [e] \rightarrow \mathsf{Bridge}_i$
                              $|\ reset \rightarrow \mathsf{Bridge}$

$//Specification\ of\ the\ interoperability\ platform\ connector$
$\|\mathsf{C\text{-}InteropPlatforms} = \mathsf{R}_{interface}\|\mathsf{Switch}\|\mathsf{T}\|\mathsf{Bridge}\|_{i=1}^{n} tag_i : \mathsf{Glue}_i\|\mathsf{R}_2$
```

Fig. 14. Interoperability platform specification

Practically, middleware-level interoperability platforms, such as UIC [44] and ReMMoC [21], allow the development of applications independently from the underlying protocol. They select the most appropriate communication protocol according to the context. Many applications, however, have not been developed using such middleware interface and cannot be modified because their source codes are not available. From the perspective of application-layer protocols, the common interface is in general a domain-specific standard that several components and services comply with. However, compliance to the same interface does not necessarily imply behavioral compatibility and mediators have to be used in order to guarantee behavioral compatibility as well [15].

3.3 Transparent Interoperability

Unlike indirect bridging, transparent interoperability solutions do not rely on a fixed common protocol anymore but rather synthesize the common protocol dynamically based on the interaction behavior of communicating parties. We are more specifically interested in *dynamic protocol translation* [7]. This approach is based on concepts taken from the theory of protocol projection [30]. The theory enables mapping incompatible protocols to an image protocol (see Figure 15), which has proven effective to reason about conversions and semantic equivalence among heterogeneous protocols [7]. In particular, an image protocol abstracts incompatibilities among protocols to exclusively consider their similarities. Further, by generating an image protocol on-the-fly, it is possible to provide a dynamic semantic correspondence among heterogeneous middleware protocols.

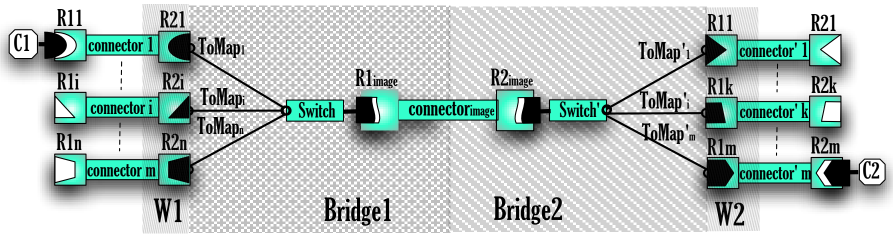

Fig. 15. Transparent interoperability

Formally, a projection function f is used to synthesize an image protocol, that is the greatest common denominator between a pair of protocols (see Figure 16). Interoperability is then performed in the following steps:

1. The glue of all the connectors are tagged in order to avoid unwanted event synchronization,
2. One connector is dynamically chosen among n (m) connectors based on the context/environment through the Switch (Switch′) process: $\texttt{Connector}_i$ ($\texttt{Connector}'_k$) (Lines 19 and 20),
3. W_1 (W_2) (Lines 22 to 25) are then used to synchronize tagged glues with their respective roles depending on the selected connector,
4. The strength of the approach lies in M_1 and M_2 processes (Lines 26 to 32) that are used to define the semantics of the events. To do so, the projection function (f) is used to establish the semantic equivalence between events: $f(e_1) = f(e_2)$ iff e_1 and e_2 have the same semantics,
5. Bridge_1 and Bridge_2 (Lines 34 to 41) tag/untag the projected events in order to allow M_1 and M_2 to synchronize.

This leads to the following specification of the *transparent mediator*:

$$\|\mathsf{Transparent\_Mediator} = (\mathsf{Switch}\ \|\mathsf{W}\ _1\|\mathsf{M}\ _1\|\mathsf{Bridge}_1\|\mathsf{Bridge}_2\|\mathsf{M}\ _2\|\ \mathsf{W}_2\|\mathsf{Switch}').$$

Practically, the INDISS [9] and NEMESYS [7] middleware implement the dynamic protocol translation approach for service discovery and interaction protocol (assuming the same application atop) respectively. uMiddle [36], OSDA [31], SeDiM [19] are other middleware-level implementations of the transparent interoperability approach. Regarding the application layer, there is a substantial piece of work on transparent interoperability at the application layer assuming the use of Semantic Web technologies. OWL-S [51] exploit Semantic Web ontologies to

```
//Connectors specification
Role R1                  $= |_{i=1}^{n}(a.glue_i \rightarrow \mathsf{R1}_i),$
$\mathsf{R1}_{i,i\in[1\cdot\cdot n]}$   $= \mathsf{R1}_i$ Initial specification as given by $\mathtt{Connector}_i|reset \rightarrow \mathsf{R1}$
Role $R2$                $= |_{k=1}^{n}(b.glue_k \rightarrow \mathsf{R2}_k),$
$\mathsf{R2}_{k,k\in[1\cdot\cdot n]}$   $= \mathsf{R2}_k$ Initial specification as given by $\mathtt{Connector'}_k|reset \rightarrow \mathsf{R2}$
$\mathsf{Glue}_{i,i\in[1\cdot\cdot n]}$  $=$ Specification that describes interactions between Role $\mathsf{R1}_i$ and Role $\mathsf{R2}_i$
$\mathsf{Glue'}_{k,k\in[1\cdot\cdot m]}$ $=$ specification that describes interactions between Role $\mathsf{R'1}_k$ and Role $\mathsf{R'2}_k$

//Definition of set of events
set $\mathsf{I1}_{i,i\in[1\cdot\cdot n]}$   $=$ Set of events initiated from Role $\mathsf{R1}_i$
set $\mathsf{O1}_{i,i\in[1\cdot\cdot n]}$   $=$ Set of events observed from Role $\mathsf{R1}_i$
set $\mathsf{I2}_{k,k\in[1\cdot\cdot m]}$   $=$ Set of events initiated from Role $R'2_k$
set $\mathsf{O2}_{k,k\in[1\cdot\cdot m]}$   $=$ Set of events observed from Role $R'2_k$
set $\mathsf{E}_{i,i\in[1\cdot\cdot n]}$    $= \alpha\mathsf{R1}_i \cap \alpha\mathsf{Glue}_i$
set $\mathsf{E}_{k,k\in[1\cdot\cdot m]}$    $= \alpha\mathrm{R2}_k \cap \alpha\mathsf{Glue'}_k$
set $\sum_{\mathsf{E1}_n}$             $= \cup_{i=1}^{n}\mathsf{E1}_i$
set $\sum_{\mathsf{E2}_m}$             $= \cup_{k=1}^{m}\mathsf{E2}_k$
set $\sum_{\mathsf{O1}_n}$             $= \cup_{i=1}^{n}\mathsf{O1}_i$
set $\sum_{\mathsf{O2}_m}$             $= \cup_{k=1}^{m}\mathsf{O2}_k$
Switch                   $= (a.election \rightarrow a.reset \rightarrow \mathsf{Switch}|_{i=1}^{n}a.election \rightarrow a.glue_i \rightarrow \mathsf{Switch})\backslash\{a.election\}$
Switch'                  $= (b.election \rightarrow b.reset \rightarrow \mathsf{Switch'}|_{k=1}^{m}b.election \rightarrow b.glue'_k \rightarrow \mathsf{Switch'})\backslash\{b.election\}$

//Specification of processes for the image protocol generation
$\mathsf{W}_1$                    $= |_{i=1}^{n}(a.glue_i \rightarrow ToGlue_i),$
$\mathsf{ToGlue}_{i,i\in[1\cdot\cdot n]}$  $= [e:\mathsf{I1}_i] \rightarrow a.tag_i.[e] \rightarrow \mathsf{ToGlue}_i\ |a.tag_i.[e:\mathsf{O1}_i] \rightarrow [e] \rightarrow \mathsf{ToGlue}_i\ |a.reset \rightarrow \mathsf{W}_1$
$\mathsf{W}_2$                    $= |_{k=1}^{n}(b.glue'_k \rightarrow \mathsf{ToGlue}'_k),$
$\mathsf{ToGlue}'_{k,k\in[1\cdot\cdot m]}$ $= [e:\mathsf{I2}_k] \rightarrow b.tag_k.[e] \rightarrow \mathsf{ToGlue}'_k\ |b.tag_k.[e:\mathsf{O2}_k] \rightarrow [e] \rightarrow \mathsf{ToGlue}'_k\ |b.reset \rightarrow \mathsf{W}_2$
$\mathsf{M}_1$                    $= |_{i=1}^{n}(a.glue_i \rightarrow \mathsf{ToMap}_i),$
$\mathsf{ToMap}_{i,i\in[1\cdot\cdot n]}$   $= a.tag_i.[e:\mathsf{I1}_i] \rightarrow a.tag_i.f(e) \rightarrow \mathsf{ToMap}_i$
                         $|\ a.tag_i.f(e:\sum_{\mathsf{O1}_n}) \rightarrow a.tag_i.[e:\mathsf{O1}_i] \rightarrow \mathsf{ToMap}_i|a.reset \rightarrow \mathsf{M}_1$
$\mathsf{M}_2$                    $= |_{k=1}^{n}(b.glue'_k \rightarrow \mathsf{ToMap}'_k),$
$\mathsf{ToMap}'_{k,k\in[1\cdot\cdot m]}$  $= b.tag_k.[e:\mathsf{I2}_k] \rightarrow b.tag_k.f(e) \rightarrow \mathsf{ToMap}'_k$
                         $|\ b.tag_k.f(e:\sum_{\mathsf{O2}_m}) \rightarrow b.tag_k.[e:\mathsf{O2}_k] \rightarrow \mathsf{ToMap}'_k$
                         $|\ b.reset \rightarrow \mathsf{M}_2$

//Specification of the bridging processes
$\mathsf{Bridge}_1$               $= |_{i=1}^{n}(a.glue_i \rightarrow \mathsf{ToBridge}_i),$
$\mathsf{ToBridge}_{i,i\in[1\cdot\cdot n]}$ $= a.tag_i.f\left(e_2:\sum_{\mathsf{E2}_k}\right) \rightarrow f(e_2) \rightarrow \mathsf{ToBridge}_i$
                         $|\ f(e_1:\sum_{\mathsf{E1}_n}) \rightarrow a.tag_i.f(e_1) \rightarrow \mathsf{ToBridge}_i$
                         $|\ a.reset \rightarrow \mathsf{Bridge}_1$
$\mathsf{Bridge}_2$               $= |_{k=1}^{m}(b.glue'_k \rightarrow \mathsf{ToBridge}'_k),$
$\mathsf{ToBridge}'_{k,k\in[1\cdot\cdot m]}$ $= b.tag_k.f\left(e_1:\sum_{E1_n}\right) \rightarrow f(e_1) \rightarrow \mathsf{ToBridge}'_k$
                         $|\ f(e_2:\sum_{\mathsf{E2}_m}) \rightarrow b.tag_k.f(e_2) \rightarrow \mathsf{ToBridge}'_k$
                         $|\ b.reset \rightarrow \mathsf{Bridge}_2$

//Specification of the transparent interoperability connector
$\|\mathsf{C\text{-}Transparent\_Interop} = \mathsf{R1}\|\ \mathsf{Switch}\ \|_{i=1}^{n}a.tag_i:\mathsf{Glue}_i/\{f(r:\alpha\mathsf{Glue}_i)/[r]\}\|\ \mathsf{W}\ _1\|\ \mathsf{M}\ _1\|\mathsf{Bridge}_1\|\mathsf{Bridge}_2$
                         $\|\mathsf{M}\ _2\|\ \mathsf{W}\ _2\|_{k=1}^{m}b.tag_k:\mathsf{Glue}_k/\{f(r:\alpha\mathsf{Glue}_k)/[r]\}\|\mathsf{Switch}\ '\|\mathsf{R2}$
```

Fig. 16. Transparent interoperability specification

enrich descriptions of services in order to enhance service discovery and composition using semantic matching. Web Service Modeling Ontology (WSMO)[9] introduces mediators as the core of a conceptual model treating heterogeneity of Semantic Web Services. In particular, it addresses both data and behavioral mediation.

As briefly surveyed in this section, tremendous work exists on the development of concrete interoperability solutions to overcome protocol heterogeneity and in particular middleware protocol heterogeneity. However, these solutions focus on a single protocol layer, while the connection of pervasive networked systems requires dealing with protocol heterogeneity at both application and middleware layers. In addition, middleware heterogeneity is in general overcome for middleware protocols implementing the same interaction paradigms while the increasing heterogeneity of the networked devices now calls for connecting systems relying on different interaction paradigms.

4 Emergent Connector Synthesis

Towards overcoming the increasing heterogeneity of today's pervasive networking environments, this section introduces a model-based approach to the synthesis of emergent connectors, which builds upon the theory of mediators introduced for application-layer protocols in [46] and further surveyed in companion chapter [26]. An emergent connector allows two networked systems that complementary provide/require the same functionality to coordinate although they possibly execute different protocols. This then requires adequate modeling of networked systems to enable reasoning about their semantic and behavioral compatibility/matching (Section 4.1), which in particular relies on the definition of ontologies conceptualizing middleware and application functions (Section 4.2). Briefly stated, two networked systems are considered to be *semantically matching* if they respectively require and provide a matching high-level functionality, which is characterized by ontology concepts. Then, assessing whether the two networked systems are *behaviorally compatible* relies on analyzing whether the protocols associated with the realization of the given functionality may be adapted so that they can successfully coordinate. The resulting adaptation then defines the mediator to be implemented by the emergent connector. As illustrated by the rich literature on protocol conversion (e.g., [11]), different compatibility relations may be defined. They primarily differ according to their complexity and conversely proportional flexibility. In order to lower the complexity of emergent connectors, we perform protocol mediation according to known mapping between the networked systems' actions, which is inferred from their ontology-based semantics. In addition, protocol mediation is composed according to the basic mediation patterns known from the literature (Section 4.3), while concrete connectors handle actual middleware message translation (Section 4.4). Finally, our work takes inspiration from extensive literature in the area of protocol medi-

[9] http://www.wsmo.org/

ation and middleware interoperability; our contribution primarily lies in dealing with mediation from application down to the middleware layer (Section 4.5).

4.1 Modeling Networked Systems towards On-the-fly Connection

A basic assumption of on-the-fly connection of networked systems is that systems advertise their presence in the network(s) they join. This is now common in pervasive networks and supported by a number of resource discovery protocols [53]. Still, a crucial question is which description of resources should be advertised, which ranges from simple (attribute, value) pairs as with SLP[10] to advanced ontology-based interface specification [3].

In our work, resource description shall enable networked systems to compose according to the high-level functionalities they provide and/or require in the network, despite heterogeneity in the protocols associated with the implementation of the functionality. In other words, networked systems must advertise the *high-level functionalities* they provide and/or consume to be able to meet according to the matching of their respective functionalities. Building upon Semantic Web Services, we call such functionalities *capabilities* and we say that networked systems *semantically match* when a networked system requires a capability that matches a capability provided by the other. Then, in accordance with the definition of connectors discussed in Section 2, connection between semantically matching networked systems requires precise characterization of the protocols associated with the realization of capabilities, where protocols are defined as processes over the networked system's observable actions. Observable actions are typically specified as part of the system's *interface signature* while the modeling of protocols relies on some concurrent language and may be advertised by the system or be possibly learned. Last but not least, the semantics of observable actions need to be rigorously defined in order to assign the same meaning to actions in any environment, for which we exploit ontologies.

The following paragraphs further define the notions of *capability*, *interface signature*, and c*capability protocol*.

Capability. Using the terminology of the Semantic Web Services area[11], a *capability* denotes a high-level functionality provided or required from the networked environment. Concretely, a capability is specified as a tuple:

$$Capability =< Type, \mathcal{C}, I, O >$$

where:

– *Type* stands for required (noted *Req*), provided (noted *Prov*) or required and provided (noted *Req_Prov*) capability. A provided capability denotes a capability offered in the network while a required one is to be consumed. A required and provided capability is then both consumed and offered by the networked system, as common in peer-to-peer systems.

[10] http://www.openslp.org/
[11] http://www.ai.sri.com/daml/services/owl-s/

- $\mathcal{C}$ gives the semantics of the capability in terms of an ontology concept;
- I (resp. O) specifies the set of inputs (resp. outputs) of the capability, which is defined as a tuple $< i_1, ..., i_n >$ (resp. $< o_1, ..., o_m >$) with $i_{l=[1..n]}$ (resp. $o_{l=[1..m]}$) being an ontology concept.

and where the ontology concepts are defined by a domain-specific ontology referred to in the networked system's interface. As an illustration, the capability of the photo sharing consumer application is defined as:

<Req, Photo-Sharing_Consumer, Comment, Photo>

where the meaning of concepts is direct from the given names (see further Section 4.2 for the definition of the ontology).

Interface signature. The interface signature of a networked system specifies the set of observable actions that the system executes to interact with other systems. In particular, networked systems implement advertised capabilities as protocols over observable actions that are defined in their interfaces. Usually, the interface signature abstracts the specific middleware functions that the system calls to carry out actions in the network. However, this is due to the fact that existing interface definition languages are closely tight to a specific middleware solution, while we target pervasive networking environments hosting heterogeneous middleware solutions. The specification of an action should then be enriched with the one of the middleware function that is specifically used to carry out that action; indeed, an observable action in an open pervasive network is the conjunction of an application-layer with a middleware-layer function. Middleware functions then need to be unambiguously characterized, which leads us to introduce a middleware ontology that defines key concepts associated with state-of-the-art middleware API, as presented in the next section.

Given the above, the interface of a networked system is defined as a set of actions where each action is described as a tuple: $< m_f, a, I, O >$, where: m_f denotes a middleware function; a denotes the application action; I (resp. O) denotes the set of inputs (resp. outputs) of the action. Moreover, as detailed in Section 4.2, the tuple elements are ontology concepts so that their semantics may be reasoned upon.

As an illustration, Figure 17[12] gives the interface signatures associated with the infrastructure-based implementation of photo sharing. The interfaces refer to ontology concepts from the middleware and application-specific domains of the target scenario; however, this does not prevent general understanding of the signatures given the self-explanatory naming of concepts. Three interface signatures are introduced, which are respectively associated with the *producer*, *consumer* and *server* networked systems. The definition of the systems' actions specify the associated SOAP functions, i.e., the client-side application actions are invoked though SOAP middleware using the *SOAP-RPCCall* function followed

[12] As defined in the next section, *photoFile* and *photoComment* include photoID.

by the *SOAP-RPCReceiveReply* function, while they are processed on the server side using the two functions *SOAP-RPCReceiveCall* and *SOAP-RPCReply*. The specific applications actions are rather straightforward from the informal sketch of the scenario introduced in Section 1. For instance, the producer invokes the server operations *Authenticate* and *UploadPhoto* for authentication and photo upload, respectively. The consumer may possibly search for, download or comment photos, or download comments. Finally, the actions of the photo sharing server are complementary to the client actions.

$$
\begin{array}{l}
\mathsf{Interface}_{photo\_sharing\_producer} = \{\\
\quad < \textit{SOAP-RPCCall}, \textit{Authenticate}, < \textit{login} >, \emptyset >,\\
\quad < \textit{SOAP-RPCReceiveReply}, \textit{Authenticate}, \emptyset, < \textit{authenticationToken} >>,\\
\quad < \textit{SOAP-RPCCall}, \textit{UploadPhoto}, < \textit{photo} >, \emptyset >\\
\quad < \textit{SOAP-RPCReceiveReply}, \textit{UploadPhoto}, \emptyset, < \textit{acknowledgment} >>\\
\}\\
\mathsf{Interface}_{photo\_sharing\_consumer} = \{\\
\quad < \textit{SOAP-RPCCall}, \textit{SearchPhotos}, < \textit{photoMetadata} >, \emptyset >,\\
\quad < \textit{SOAP-RPCReceiveReply}, \textit{SearchPhotos}, \emptyset, < \textit{photoMetadataList} >>,\\
\quad < \textit{SOAP-RPCCall}, \textit{DownloadPhoto}, < \textit{photoID} >, \emptyset >,\\
\quad < \textit{SOAP-RPCReceiveReply}, \textit{DownloadPhoto}, \emptyset, < \textit{photoFile} >>,\\
\quad < \textit{SOAP-RPCCall}, \textit{DownloadComment}, < \textit{photoID} >, \emptyset >,\\
\quad < \textit{SOAP-RPCReceiveReply}, \textit{DownloadComment}, \emptyset, < \textit{photoComment} >>,\\
\quad < \textit{SOAP-RPCCall}, \textit{CommentPhoto}, < \textit{photoComment} >, \emptyset >\\
\quad < \textit{SOAP-RPCReceiveReply}, \textit{CommentPhoto}, \emptyset, < \textit{acknowledgment} >>\\
\}\\
\mathsf{Interface}_{photo\_sharing\_server} = \{\\
\quad < \textit{SOAP-RPCReceiveCall}, \textit{Authenticate}, < \textit{login} >, \emptyset >,\\
\quad < \textit{SOAP-RPCReply}, \textit{Authenticate}, \emptyset, < \textit{authenticationToken} >>,\\
\quad < \textit{SOAP-RPCReceiveCall}, \textit{UploadPhoto}, < \textit{photo} >, \emptyset >,\\
\quad < \textit{SOAP-RPCReply}, \textit{UploadPhoto}, \emptyset, < \textit{acknowledgment} >>,\\
\quad < \textit{SOAP-RPCReceiveCall}, \textit{SearchPhotos}, < \textit{photoMetadata} >, \emptyset >,\\
\quad < \textit{SOAP-RPCReply}, \textit{SearchPhotos}, \emptyset, < \textit{photoMetadataList} >>,\\
\quad < \textit{SOAP-RPCReceiveCall}, \textit{DownloadPhoto}, < \textit{photoID} >, \emptyset >,\\
\quad < \textit{SOAP-RPCReply}, \textit{DownloadPhoto}, \emptyset, < \textit{photoFile} >>,\\
\quad < \textit{SOAP-RPCReceiveCall}, \textit{DownloadComment}, < \textit{photoID} >, \emptyset >,\\
\quad < \textit{SOAP-RPCReply}, \textit{DownloadComment}, \emptyset, < \textit{photoComment} >>,\\
\quad < \textit{SOAP-RPCReceiveCall}, \textit{CommentPhoto}, < \textit{photoComment} >, \emptyset >,\\
\quad < \textit{SOAP-RPCReply}, \textit{CommentPhoto}, \emptyset, < \textit{acknowledgment} >>\\
\}
\end{array}
$$

Fig. 17. Interface signature of infrastructure-based photo sharing

Unlike the infrastructure-based implementation, the peer-to-peer-based photo sharing defines a single interface signature (see Figure 18), as all the peers feature the same capability. The interface further illustrates the naming of actions after domain data types of the application data instead of operations since the actions

$$
\begin{array}{l}
\mathsf{Interface}_{photo\_sharing} = \{ \\
\quad < Out, PhotoMetadata, \emptyset, < photoMetadata >>, \\
\quad < Out, PhotoFile, \emptyset, < photoFile >>, \\
\quad < Rdg, PhotoMetadata, < photoMetadata >, < photoMetadataList >>, \\
\quad < Rd, PhotoFile, < photoID >, < photoFile >>, \\
\quad < Rd, PhotoComment, < photoID >, < photoComment >>, \\
\quad < Out, PhotoComment, \emptyset, < photoComment >>, \\
\quad < In, PhotoComment, < photoID >, < photoComment >>, \\
\quad < Rd, PhotoComment, < photoID >, < photoComment >> \\
\}
\end{array}
$$

Fig. 18. Interface signature of Peer-to-Peer-based photo sharing

are data-centric and are performed through functions of the LIME[13] tuple-space middleware.

Capability protocol. Given the networked system's interface signature, the behavior of the system's capabilities is specified as protocols over the system's actions defined in the interface signature. Such protocols need to be explicitly defined using some concurrent language, as part of the networked system's advertisements. Alternatively, the protocol specification may be learned in a systematic way based on the system's interfaces as investigated in the companion chapter on automata learning [18]. Different languages may be considered for such a specification from formal modeling to programming languages.

Formal languages are a prerequisite for automated reasoning about matching and mediator generation while well-established language from the Web service domain, such as BPEL[14], are easier for developer to deal with. Indeed, BPEL offers many advantages for the definition of processes, among which: (i) the specification of both data and control flows that allow identifying causally independent actions; (ii) the formal specification of BPEL in terms of process algebra that allows abstracting BPEL processes for automated reasoning about protocol matching [20]; and (iii) the rich tool sets coming along with BPEL, which in particular ease process definition by developers. However, same as for the interface signature definition, the language must be generalized to not be only specific to the Web service technology. Precisely, BPEL needs to be enriched so as to support interaction with networked systems using other interaction patterns and protocols than those classically associated with Web services, which can be addressed in a systematic way using the BPEL extension mechanism. Therefore, BPEL may be used by developers to specify the protocol implemented by the networked systems and automatically translated into FSP process algebra.

For illustration, Figure 3 gives the FSP-based specification of the protocols associated with a SOAP-based implementation of the infrastructure-based version of photo sharing application, while Figure 19 introduces the specification of

[13] http://lime.sourceforge.net
[14] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

```
//Peer-to-Peer-based application specification
set Lime_PhotoSharing_Actions = {photoMetadata, photoFile, photoComment}
PhotoSharingPeer              = (req.photoMetadata →Consumer |prov.photoMetadata → Producer),
Producer                      = (prov.photoFile →PhotoSharingPeer),
Consumer                      = (req.photoFile →Consumer |req.photoComment →Consumer
                                 |prov.photoComment →Consumer |req.photoFile →PhotoSharingPeer
                                 |req.photoComment →PhotoSharingPeer
                                 |prov.photoComment → PhotoSharingPeer |terminate →END).

//Lime middleware Specification
Lime_Reader(X =′ tuple)       = (req.[X] →P1),
P1                            = (rd[X] →Lime_Reader |rdp[X] →Lime_Reader
                              | rdg[X] →Lime_Reader |in[X] →Lime_Reader
                              | inp[X] →Lime_Reader |ing[X] →Lime_Reader
                              | terminate → END).
Lime_Writer(X =′ tuple)       = (prov.[X] →P2),
P2                            = (out[X] →Lime_Writer |outp[X] →Lime_Writer
                              | outg[X] →Lime_Writer |terminate →END).
Lime_glue(X =′ tuple)         = (write[X] → P0 |outp[X] → P0 |outg[X] → P0
                              | terminate →END),
P0                            = (rd[X] →P0 |rdp[X] →P0 |rdg[X] →P0
                              | in[X] →Lime_glue |inp[X] →Lime_glue |ing[X] →Lime_glue).

const NumberOfPeers           = 2
||Lime_PhotoSharing           = ( [i : 1..NumberOfPeers]:PhotoSharingPeer
                                 ||(forall [tuple:Lime_PhotoSharing_Actions] Lime_Writer(tuple))
                                 ||(forall [tuple:Lime_PhotoSharing_Actions] Lime_Reader(tuple))
                                 ||(forall [tuple:Lime_PhotoSharing_Actions] Lime_glue(tuple))).
```

Fig. 19. Peer-to-Peer-based photo sharing

a LIME-based implementation of the peer-to-peer version of the photo sharing application. The protocol executed by a LIME-based networked system allows for both production and consumption of photo files. On the other hand, there are different protocols for the producer, consumer and server for the SOAP-based implementation due to the distinctive roles imposed by the service implemented by the photo sharing server. Still, emergent connectors shall enable seamless interaction of the LIME-based photo sharing implementation with systems implementing capabilities of the infrastructure-based photo sharing.

4.2 Ontology for Mediation

Realizing emergent connectors primarily relies on reasoning about capability matching together with identifying matching observable actions among the actions performed by networked systems. Ontologies play a key role in identifying such matching and allow overcoming the inherent heterogeneity of pervasive networked systems. Indeed, “an ontology is a formal, explicit specification of a shared conceptualization” [49]. Such an ontology is then assumed to be shared widely. In addition, work on ontology alignment enables dealing with possible usage of distinct ontologies in the modeling of the different networked systems [17].

Different relations may be defined between ontology concepts. The *subsumption* relation (in general named *is-a*) is essential since it allows, besides equivalence, to match between concepts based on inclusion. Precisely: a concept C *is subsumed by* a concept D in a given ontology O, noted $C \sqsubseteq D$, if in every model of O the set denoted by C is a subset of the set denoted by D [2].

Towards enabling emergent connectors, we introduce a middleware ontology that forms the basis of middleware protocol mediation. In addition, domain-specific application ontologies characterizing application actions serve defining both control- and data-centric concepts.

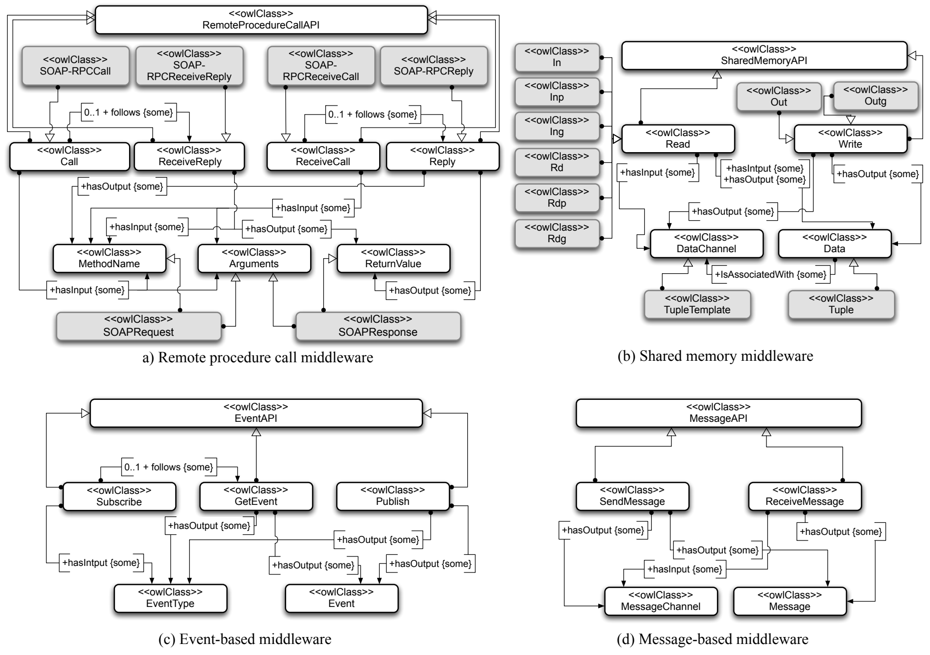

Fig. 20. Middleware ontology

Middleware ontology. As discussed in Section 2.1, state-of-the-art middleware may be categorized according to four *middleware types* regarding provided communication and coordination services [50]: *remote procedure call, shared memory, event-based* and *message-based*. As depicted in Figure 20 and more specifically with concepts defined in white boxes, the proposed middleware ontology is structured around these four categories, which serve as reference enabling to align functions of different middleware solutions. Indeed, the reference middleware ontology can be refined into concepts associated with functions of a specific middleware. This is illustrated in the figure by the grayed boxes that define concepts of the Lime and SOAP-based middleware solutions that we specifically consider in our photo sharing scenario. In addition to the *is-a* relation that is denoted by a white arrow, the middleware ontology introduces a number of customized relations between concepts: *hasOutput* (resp. *hasInput*) to characterize output (resp. input) parameters. We also use relations from best practices in ontology design[15] as illustrated by the *follows* relation that serves defining sequence patterns.

The ontology is given as a set of UML diagrams. In Figure 20.a), the ontology concepts associated with RPC-based middleware include the *Call* function parameterized by the method name and arguments, which must must be followed by the *ReceiveReply* function to receive the result of the call. On the server side, the *ReceiveCall* function to catch an invocation is followed by the execution of the *Reply* function to return the result. The ontologies of functions for shared memory and message-based middleware are rather straightforward. In the former, the shared memory is accessed through *Read*/*Write* functions parameterized by the associated data and corresponding channel (see Figure 20.b). In the latter, messages are exchanged using the *SendMessage* and *ReceiveMessage* functions parameterized by the actual message and related channel (see Figure 20.d). Regarding event-based middleware, events are published using the *Publish* function parameterized by the specific event; while they are consumed through the *GetEvent* function after registering for the specific event type using the *Subscribe* function (see Figure 20.c).

The proposed ontology serves aligning the functions of middleware of the same type through mapping onto the reference functions, which is illustrated for the specific cases of SOAP-based and Lime middleware. Heterogeneity in the underlying implementation may then be overcome using transparent middleware interoperability solutions (see Section 3.3).

A further challenge for emergent connectors in pervasive networking environments is to enable mediation among different middleware types. To enable such mediation, we introduce a further abstraction allowing cross-type alignment of middleware functions. More specifically, according to their semantics, middleware functions may be aligned based on whether they produce or consume an action in the network. We hence define the mapping of middleware functions onto abstract *input* and *output* (denoted by an overbar) actions, which are parameterized by the application action a and associated input I and output O.

The alignment of (possibly sequence of) middleware functions as abstract input and output actions is summarized in Figure 21. The alignment defined for shared memory and message-based middleware functions is rather direct: the *Write* and *SendMessage* functions are mapped onto an output action; while the *Read* and *ReceiveMessage* translate into an input action. Note that *Read* is possibly parameterized with I if the value to be read shall match some constraints, as, e.g., enabled by tuple space middleware. The alignment for the event-based middleware functions is straightforward for *Publish*: publication of an event maps onto an output action. The dual input action is performed by the

[15] http://ontologydesignpatterns.org

GetEvent function, which is preceded by at least one invocation of *Subscribe* on the given event[16]. The semantics of RPC functions follows from the fact that it is the server that produces an application action, although this production is called upon by the client. Then, the output action is defined by the execution of *ReceiveCall* followed by *Reply*, while the dual input action is defined by the *Invoke* function.

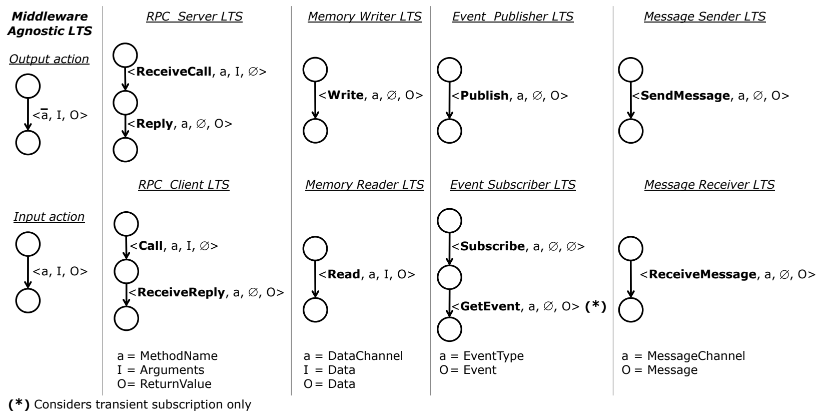

Fig. 21. Middleware alignment

```
Reader(X =' data)   = (req.[X] →P1),
P1                  = (read[X] →Reader |terminate → END).
Writer(X =' data)   = (prov.[X] →P2),
P2                  = (write[X] →Writer |terminate →END).
SM_glue(X =' data)  = (write[X] → P3 |terminate →END),
P3                  = (read[X] →SM_glue).
```

Fig. 22. Shared-memory middleware type specification

The given alignments abstract protocols associated with the realization of capabilities as *middleware-agnostic processes*. As a result, protocols may be matched based purely on their application-specific features. In more detail, middleware-specific functions are abstracted as middleware functions from the reference ontology, which are then translated into input and output actions through the defined alignment. This is illustrated in Figure 22, which gives the FSP-based protocol associated with the peer-to-peer photo sharing implemen-

[16] Note that for the sake of conciseness, the figure depicts only the case where a *Subscribe* is followed by a single *GetEvent*.

$$\begin{array}{ll}
\mathsf{Reader}(X =' data) & = (input[X] \rightarrow \mathsf{Reader} | terminate \rightarrow \mathsf{END}). \\
\mathsf{Writer}(X =' data) & = (output[X] \rightarrow \mathsf{Writer} | terminate \rightarrow \mathsf{END}). \\
\mathsf{SM\_glue}(X =' data) & = (output[X] \rightarrow \mathsf{P}\ | terminate \rightarrow \mathsf{END}), \\
\mathsf{P} & = (input[X] \rightarrow \mathsf{SM\_glue}).
\end{array}$$

Fig. 23. Middleware-agnostic peer-to-peer photo sharing

tation, after abstracting middleware-specific functions into reference functions (see Figure 23) and further aligning onto middleware-agnostic input and output actions. Thanks to the alignment of middleware functions, processes may be matched against the realization of matching application-specific actions whose semantics is given by the associated ontology.

Application-specific ontology. The *subsumption* relation of ontologies serves matching application-specific capabilities and actions against each other. Basically, and as detailed in the next section, a required capability/action matches a provided one if the former is subsumed by the latter.

For illustration, Figure 24 gives an excerpt of the domain-specific ontology associated with our photo sharing scenario, which shows the subsumptions holding among the various concepts defining the interfaces of the networked systems implementing the scenario.

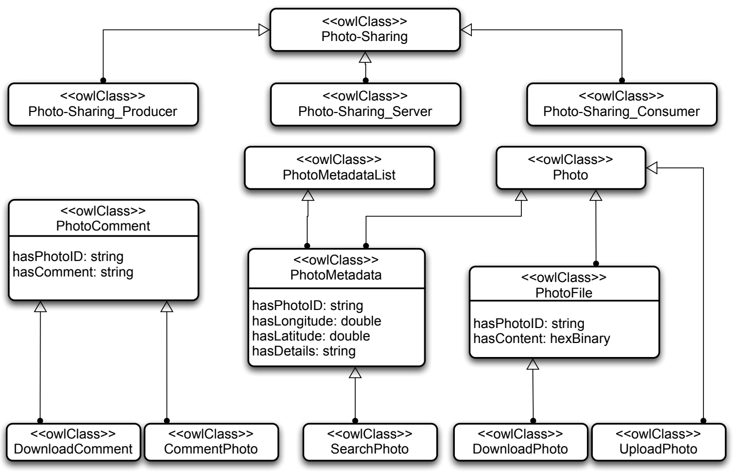
<<owlClass>> Photo-Sharing
<<owlClass>> Photo-Sharing_Producer
<<owlClass>> Photo-Sharing_Server
<<owlClass>> Photo-Sharing_Consumer
<<owlClass>> PhotoMetadataList
<<owlClass>> Photo
<<owlClass>> PhotoComment
hasPhotoID: string
hasComment: string
<<owlClass>> PhotoMetadata
hasPhotoID: string
hasLongitude: double
hasLatitude: double
hasDetails: string
<<owlClass>> PhotoFile
hasPhotoID: string
hasContent: hexBinary
<<owlClass>> DownloadComment
<<owlClass>> CommentPhoto
<<owlClass>> SearchPhoto
<<owlClass>> DownloadPhoto
<<owlClass>> UploadPhoto

Fig. 24. Photo sharing ontology

Note that the application-specific ontology not only describes the semantics and relationships related to data but also to the functionalities and roles of the networked systems, such as *Photo-Sharing_Producer*, *Photo-Sharing_Consumer*, and *Photo-Sharing_Server*. It also defines the semantics of the operations performed on data, such as *UploadPhoto*, *DownloadPhoto*, and *SearchPhoto*. Furthermore, it relates data to operations: data subsumes the operations performed on them. The rationale behind this statement is that by having access to data, any operation could be performed on it. For example *PhotoFile* subsumes *DownloadPhoto* since by providing access to a photo file, one can download it.

Finally, subsumption is not the panacea to reason about semantic relationships between concepts and many other relations such as sequence [16] or part-whole[17] should be specified. We believe that best practices of ontology design and ontology engineering[18] and the use of ontology design patterns[19] may prove very beneficial to automatically discover and reuse semantic relations between concepts.

4.3 Emergent Connectors

Given the models characterizing networked systems that are introduced in Section 4.1 and related ontology definition, emergent connectors are enabled through matching and mapping functions defined over the actions of networked systems. Precisely, if two networked systems implement matching capabilities, then they may possibly coordinate towards the realization of the capability. This is achieved by mapping the respective actions of the systems according to their ontology-based semantics, and then synthesizing the mediator that adapts accordingly the interaction protocols executed by the networked systems.

Capability matching. The first step in identifying the possible matching of two networked systems is to assess whether they respectively provide and require a matching capability. Precisely, and following the definition of semantic matching of capabilities [41], we say that capability $C_R =< Req, \mathcal{C}_R, I_R, O_R >$ semantically matches with capability $C_P =< Prov, \mathcal{C}_P, I_P, O_P >$, noted $C_R \hookrightarrow C_P$, *iff* in the given ontology:

- $\mathcal{C}_R \sqsubseteq \mathcal{C}_P$,
- $I_P \sqsubseteq I_R$ (which is a shorthand notation for subsumption between sets of ontology concepts), and
- $O_R \sqsubseteq O_P$.

Note that a capability C_R of type *Req produces* the inputs I_R and *consumes* the corresponding outputs O_R. In a dual manner, a capability C_P of type *Prov consumes* the inputs I_P and *produces* the corresponding outputs O_P.

In addition, since the capability is related to semantic concepts, we make a similar assumption to that made in the Semantic Web [41], i.e., by specifying $\mathcal{C}_P$ as the functional concept, the provider commits to offering all the functionalities

[17] `http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/index.html`
[18] `http://www.w3.org/2001/sw/BestPractices/OEP/`
[19] `http://ontologydesignpatterns.org`

subsumed by $\mathcal{C}_P$ and output consistent with every concept subsumed by O_P. If this is not the case, then the functionality/output should be restricted to those verifying the above assumption. Similarly, the requester commits to provide any input consistent with the classes that I_R subsumes. However, if the input/output are related to syntactic (XML-based) types and not to semantic concepts, it becomes important to verify the Liskov Substitution Principle (LSP) [32] in the following way:

– $\mathcal{C}_P$ `subtypeOf` $\mathcal{C}_R$, which corresponds to the LSP co-variance rule;
– I_R `subtypeOf` I_P, which corresponds to the LSP contra-variance rule for the outputs; and
– O_P `subtypeOf` O_R, which corresponds to the LSP co-variance rule.

That being the case, if the semantic concept is automatically extracted or learned from the syntactic description, then it should be restricted to the most specific concept. Moreover, since there is a close relation between the semantic concepts and the related syntactic objects, it is required to have specifications or methods enabling transformations between the different concepts and types.

In the case where one capability is required and provided (i.e., of type *Req_Prov*) by a networked system and the other capability is required (resp. provided) by the other networked system, the same condition as above applies considering that the *Req_Prov* capability is considered as being provided and required. For instance, given (1) and (2) below, we have (3):

$$\begin{aligned}
&PhotoSharingConsumer = < Req, Photo - Sharing\_Consumer, < PhotoComment >, < Photo >> && (1)\\
&PhotoSharing = < Req\_Prov, Photo - Sharing, < Photo > \vee < PhotoComment >,\\
&\qquad < Photo, PhotoComment >> && (2)\\
&PhotoSharingConsumer \hookrightarrow PhotoSharing && (3)
\end{aligned}$$

Given capability matching, the emergent connector between the matching networked systems should mediate possible behavioral mismatches in their respective middleware-agnostic interaction protocols. Towards that goal, we build on basic mediation patterns.

Mediation patterns. Possible behavioral mismatches for input actions need to be solved so as to ensure that any input action is synchronized with an output action of the matching networked system with respect to the realization of the capability of interest. On the other hand, the absence of consumption of an output action does not affect the behavior of the networked system as long as deadlock is prevented by the emergent connector at runtime. Still, synthesis of a protocol mediator is known as a computationally hard problem for finite state systems in general [11] and thus requires heuristics to make the problem tractable. Towards that goal, we focus on enabling basic mediation patterns [45] as introduced in the literature for, e.g., Semantic Web Services [48]. We then account for basic mediation patterns as follows:

- **Ordering mismatch:** This concerns the re-ordering of actions so that networked systems may indeed coordinate. In the case of BPEL specification, causally independent actions may be identified through data-flow analysis, hence enabling to introduce concurrency among actions and thus supporting acceptable re-ordering.
- **Extra output action (or missing input action):** As discussed above, extra output actions are simply discarded from the standpoint of behavioral matching. Obviously, the associated concrete mediator should handle any extra synchronous output action to avoid deadlock.
- **Extra input action (or missing output action):** Any input action needs to be mapped to an output action of the matching networked system. However, in this case, there is no such output action that directly maps to the input action. In a first step, we do not handle these mismatches as they would significantly increase the complexity of protocol adaptation.
- **Splitting of actions:** Splitting actions relate to having an action of one system realized by a number of actions of the other. Then, an input action may be split into a number of output actions of the matching networked system if such a relation holds from the domain-specific ontology giving the semantics of actions. On the other hand, we do not deal with the splitting of output actions, which is an area for future work given the complexity it introduces.
- **Merging of actions:** The merging of actions is the dual of splitting from the standpoint of the matching networked system. Then, we only handle the merging of output actions.

Interface mapping. Following the above, interface mapping serves identifying mapping among the actions of the interaction protocols run by the networked systems that should coordinate towards the realization of a given capability.

Let two networked systems that respectively implement the matching capabilities $\mathcal{C}_1$ and $\mathcal{C}_2$. Let further $\mathcal{I}_{\mathcal{C}_1}$ (resp. $\mathcal{I}_{\mathcal{C}_2}$) be the set of middleware-agnostic actions executed by the protocol realizing $\mathcal{C}_1$ (resp. $\mathcal{C}_2$); $\mathcal{I}_{\mathcal{C}_1}$ and $\mathcal{I}_{\mathcal{C}_2}$ are then subsets of the actions defined in the networked systems' interfaces, which are further made middleware-agnostic according to the alignment defined in Section 4.2. We introduce the function $Map_I(\mathcal{I}_{\mathcal{C}_1}, \mathcal{I}_{\mathcal{C}_2})$ which identifies the set of all possible mappings of all the input actions of $I_{\mathcal{C}_1}$ (resp. $I_{\mathcal{C}_2}$) with actions of $\mathcal{I}_{\mathcal{C}_2}$ (resp. $\mathcal{I}_{\mathcal{C}_1}$), according to the semantics of actions. Formally:

$$\begin{aligned} Map_I(\mathcal{I}_{\mathcal{C}_1}, \mathcal{I}_{\mathcal{C}_2}) = & \bigcup_{<a,I,O>\in\mathcal{I}_{\mathcal{C}_1}} \{< a, I, O > \mapsto map(< a, I, O >, \mathcal{I}_{\mathcal{C}_2})\} \bigcup \\ & \bigcup_{<a',I',O'>\in\mathcal{I}_{\mathcal{C}_2}} \{< a', I', O' > \mapsto map(< a', I', O' >, \mathcal{I}_{\mathcal{C}_1})\} \end{aligned}$$

where:

$$\begin{aligned} map(< a, I_a, O_a >, \mathcal{I}) = \{ & << \bar{b}_i, I_i, O_i > \in \mathcal{I} >_{i=1..n} \mid \\ & \quad a \sqsubseteq \cup_i \{b_i\} \\ & \wedge\ I_{i\leq n} \sqsubseteq (\cup_{j<i}\{O_j\}) \cup \{I_a\} \\ & \wedge\ O_a \sqsubseteq (\cup_{j<i}\{O_j\}) \cup \{I_a\} \\ & \} \end{aligned}$$

and

$$\forall seq_1 \in map(< a, I_a, O_a >, \mathcal{I}),\ \nexists seq_2 \in map(< a, I_a, O_a >, \mathcal{I}) | seq_2 \prec seq_1$$

where $\prec$ denotes the inclusion of sequences. In the above definition, the ordering of actions given by the sequence follows from the sequencing of actions in the protocol realizing the capability. The definition is further given in the absence of concurrent actions to simplify the notations, while the generalization to concurrent actions is rather direct.

As an illustration, we give below the interface mapping between the *PhotoSharingConsumer* and *PhotoSharing* capabilities. All the input actions of *PhotoSharingConsumer* have a corresponding output action in *PhotoSharing*. On the other hand, the input actions of *PhotoSharing* associated with the production of photos do not have matching output actions in *PhotoSharingConsumer*. As a result, we support the adaptation of protocols for interaction between *PhotoSharingConsumer* and *PhotoSharing* regarding the consumption of photos by the former only, as further discussed in the next section.

Map(Interface'$_{photo\_sharing\_consumer}$,Interface'$_{photo\_sharing}$)= {
$< SearchPhotos, photoMetadata, photoMetadataList >$
$\mapsto \{<< \overline{PhotoMetadata}, \emptyset, photoMetadata >>\}$,
$< DownloadPhoto, photoID, photoFile >$
$\mapsto \{<< \overline{PhotoFile}, \emptyset, photoFile >>\}$,
$< CommentPhoto, photoComment, acknowledgment >$
$\mapsto \{<< \overline{PhotoComment}, \emptyset, photoComment >>\}$,
$< DownloadComment, photoID, photoComment >$
$\mapsto \{<< \overline{PhotoComment}, \emptyset, photoComment >>\}$,
$< PhotoComment, photoID, photoComment > \mapsto \emptyset$,
$< PhotoMetadata, photoMetadata, photoMetadataList > \mapsto \emptyset$,
$< PhotoFile, photoID, photoFile > \mapsto \emptyset$
}

Mediator synthesis. Given interface mappings returned by Map_I, we need to identify whether the protocols associated with the matching capabilities may indeed coordinate, i.e., the concurrent execution of the two protocols successfully terminates. However, in a first step , we assume that it exists a single mapping for each input action. Formally, let:

$$\mathcal{I}'_1 = \{\alpha_i = \langle a_i, I_{a_i}, O_{a_i}\rangle\}_{i=1..n} \cup \left\{\overline{\beta_j} = \left\langle \overline{b_j}, I_{b_j}, O_{b_j}\right\rangle\right\}_{j=1..m}$$

be the abstract interface associated with required capability $\mathcal{C}_1$, and:

$$\mathcal{I}'_2 = \left\{\alpha'_{i'} = \left\langle a'_{i'}, I_{a_{i'}}, O_{a_{i'}}\right\rangle\right\}_{i'=1..n'} \cup \left\{\overline{\beta'_{j'}} = \left\langle \overline{b'_{j'}}, I_{b'_{j'}}, O_{b'_{j'}}\right\rangle\right\}_{j'=1..m'}$$

be the abstract interface associated with provided capability $\mathcal{C}_2$.

From $Map_I(\mathcal{I}'_1, \mathcal{I}'_2)$, we have:

$$\forall \alpha_{i=1..n} \in \mathcal{I}'_1 : \alpha_i \mapsto \left\langle \overline{\beta'_1}, ..., \overline{\beta'_n}\right\rangle \ | \ \beta'_j \in \mathcal{I}'_2$$

We then define the processes $\mathsf{M}_{\alpha_{i=1..n}}$ that deal with the splitting/merging of $\mathcal{C}_1$ actions by allowing the synchronization of each input action $\alpha_{i=1..n}$ with its corresponding output actions:

$$\mathsf{M}_{\alpha_{i=1..n}} = \beta'_1 \rightarrow ... \rightarrow \beta'_n \rightarrow \overline{\alpha_i} \rightarrow \mathsf{M}_{\alpha_{i=1..n}}$$

We further define the processes $\mathsf{M}_{\beta'_{j'=1..k'}}$ for any extra output action $\beta'_{j'} \in \mathcal{I}'_2$ that is not required by any input action $\alpha_{i=1..n} \in \mathcal{I}'_1$, as follows:

$$\mathsf{M}_{\beta'_{j'=1..k'}} = \beta'_{j'=1..k'} \rightarrow \mathsf{M}_{\beta'_{j'=1..k'}}$$

We define similarly $\mathsf{M}_{\alpha'_{i'=1..n'}}$ and $\mathsf{M}_{\beta_{j=1..k}}$ for $\mathcal{C}_2$.

A process P_1 associated with capability $\mathcal{C}_1$ *behaviorally matches* a process P_2 associated with capability $\mathcal{C}_2$ under $Map(\mathcal{I}'_1, \mathcal{I}'_2)$, noted $P_1 \hookrightarrow_{\mathcal{P}} P_2$, iff

$$P_1 \underset{i=1..n}{||} \mathsf{M}_{\alpha_{i=1..n}} \underset{j'=1..k'}{||} \mathsf{M}_{\beta'_{j'=1..k'}} \preceq P_2 \underset{i'=1..n'}{||} \mathsf{M}_{\alpha'_{i'=1..n'}} \underset{j=1..k}{||} \mathsf{M}_{\beta_{j=1..k}}$$

where $\preceq$ refers to trace refinement as defined in [24] and guarantees that *mediated* P_1 can safely communicate with *mediated* P_2.

Applying the above definition, we can check that:

$$P_{photo\_sharing\_consumer} \hookrightarrow_{\mathcal{P}} P_{photo\_sharing}$$

Consequently, the *emergent connector mediator* is defined as follows:

$$||\mathsf{Emergent\_Connector\_Mediator} =$$
$$\left(\underset{i=1..n}{||} \mathsf{M}_{\alpha_{i=1..n}}\right) || \left(\underset{j'=1..k'}{||} \mathsf{M}_{\beta'_{j'=1..k'}}\right) || \left(\underset{i'=1..n'}{||} \mathsf{M}_{\alpha'_{i'=1..n'}}\right) || \left(\underset{j=1..k}{||} \mathsf{M}_{\beta_{j=1..k}}\right)$$

4.4 From Abstract to Concrete Emergent Connectors

Once the model of the emergent connector has been synthesized, it needs to be transformed into a concrete software artifact. The concretization is threefold:

1. Parsing the network messages in order to generate the corresponding actions; this parsing is performed by a *Listener* specific to each middleware implementation (see Figure 25A).
2. Generating the code corresponding to the mediator (see Figure 25B).
3. Composing the abstract actions in order to generate the corresponding network message; this is the role of an *Actuator* specific to each middleware implementation (see Figure 25C).

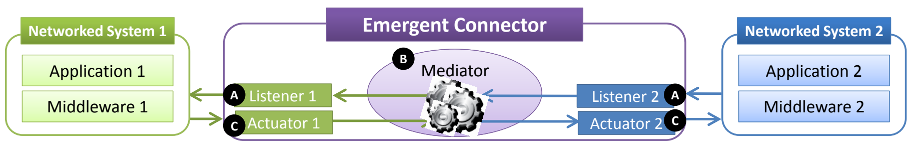

Fig. 25. Concretizing the mediator

Towards the above, we adopt results in the area of synthesis of concrete middleware protocols. Indeed, these last years two main approaches, z2z [10] and Starlink [8], have emerged to synthesize middleware, which acts as gateways to translate one protocol to another. More precisely, these approaches have instantiated the direct bridge concepts, as it provides a high degree of expressiveness and does not require modifications to existing applications. Both z2z and Starlink are based on similar concepts (see Figure 26a,b): they provide an optimized run-time system, and facilities for describing network protocol behaviors, message structures, and translation logics. Such facilities come from the fact that they rely on a high-level definition language that hides low level network details and highlights only key properties of protocols. Hence, to get a generated gateway between two heterogeneous protocols, developers must write specifications consisting of: (i) a protocol specification, describing how the protocols interact with the network, (ii) a message specification, describing the structure of message requests and responses, and (iii) a translation specification, describing how to translate messages among protocols (See Figure 26, ❶,❷). These specifications enable to generate software components such as *listeners*, *actuators* and *mediators* that are plugged into a runtime system to form, from a formal point of view, a *direct bridging* connector (as introduced in Section 3.1). *Listeners*, and *actuators* enable respectively to extract required informations relevant to the interacting parties, and to generate extracted informations in an adequate format according to protocols being used. The *mediator* applies the required translation logic to resolve mismatches between protocols.

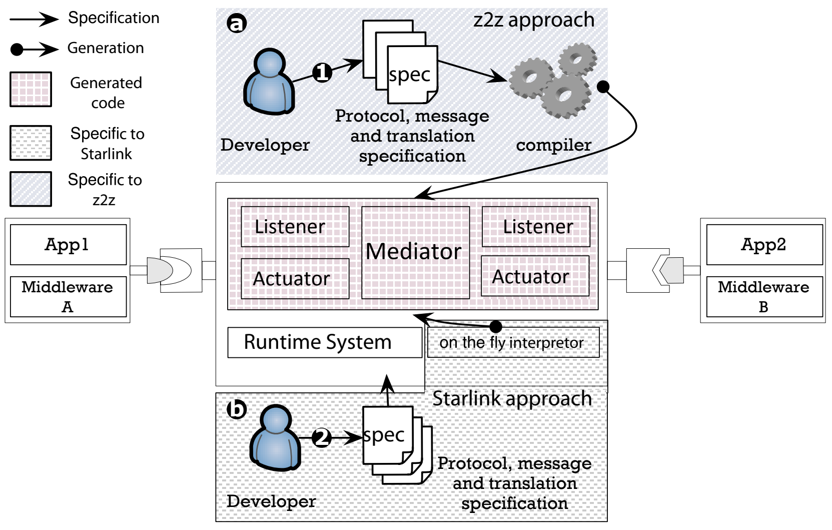

Fig. 26. z2z and Starlink approaches to synthesize middleware

Although z2z and Starlink are closed together in their design, they differ strongly in the way code plugged into the runtime is generated. With z2z, generated gateways are statically built. Hence, once such gateways are deployed in one environment, it is not anymore possible to alter afterwards the translation being processed. Consequently, in environments where systems are composed dynamically, interoperability can not be guaranteed. In general, z2z targets environments where gateways need to be embedded in resource constraint devices with performances in mind. Specifications in z2z are expressed in a C-like language and are compiled at design time. The z2z compiler relies on advanced compilation strategies to perform static verifications at the specification level and to produce highly optimized native code dedicated to the translation between two specific protocols. On the contrary, Starlink is designed with both dynamicity and genericity in mind. Specifications in Starlink are processed dynamically at runtime, and the code plugged into the runtime is done on the fly according to protocols currently used in the environment (See Figure 26b). To this end, compared to z2z, the Starlink runtime embeds both generic parsers and composers that are customized dynamically according to the specifications being used. It is important to note that in Starlink, specifications are interpreted and not compiled as in z2z. Hence, it has a potential impact on performance.

4.5 Related Work

Protocol interoperability has been the focus of significant research since the early days of networking. This has initially led to the study of systematic approaches to protocol conversion (i.e., synthesizing a mediator that adapts the two interacting protocols that need to interoperate) based on formal methods as surveyed in [11]. Existing approaches may in particular be classified into two categories depending on whether: (i) they are bottom-up, heuristic-based, or (ii) top-down, algorithmic-based. In the former case, the conversion system derives from some given protocol, which may either be inferred from the semantic correspondence between the messages of the interacting protocols [30] or correspond to the reference protocol associated with the service to be realized through protocol interaction [39]. In the latter case, protocol conversion is considered as finding the quotient between the two interacting protocols. Then, if protocols are specified as finite-state systems, an algorithm computing the quotient is possible but the problem is computationally hard since it requires an exhaustive search of possibilities [11]. Then, the advantage of the bottom-up approach is its efficiency but at the expense of: (i) requiring the message mapping or reference protocol to be given and further (ii) not identifying a converter in all cases. On the other hand, the bottom-up approach will always compute a converter if it exists given the knowledge about the semantics of messages, but at the expense of significant complexity. This has led to the further development of formal approaches to protocol conversion so as to improve the performance of proposed algorithms [29]. Our work extensively builds on these formal foundations, adopting a bottom-up approach in the form of interface mapping. However, unlike the work of [30], our interface mapping is systematically inferred, thanks to the use of ontologies. In addition, while the proposed formal approaches pave the way for rigorous reasoning about protocol compatibility and conversion, they are mostly theoretical, dealing with simple messages (e.g., absence of parameters).

More practical treatment of protocol conversion is addressed in [52], which focuses on the adaptation of component protocols for object-oriented systems. The solution is top-down in that the synthesis of the mediator requires the mapping of messages to be given. By further concentrating on practical application, the authors have primarily targeted an efficient algorithm for protocol conversion, leading to a number of constraining assumptions such as synchronous communication. In general, the approach is quite restrictive in the mediation patterns that it supports by not buffering messages and thus preventing the handling of the merging/splitting or re-ordering of messages in general. Then, while our solution relates to this specific proposal, it is more general by dealing with more complex mediation patterns and further inferring message mapping from the ontology-based specification of interfaces. Our solution further defines protocol compatibility by in particular requiring that any input action (message reception) has a corresponding (set of) output action(s), while the definition of [52] requires the reverse. Our approach then enforces networked systems to coordinate so as to update their states as needed, based on input from the environment.

More recently, with the emergence of Web services and advocated universal interoperability, the research community has been investigating how to actually support service substitution so as to enable interoperability with different implementations (e.g., due to evolution or provision by different vendors) of a service. While early work has focused on semi-automated, design-time approaches [37, 42], latest work concentrates on automated, run-time solutions [12]. Our work closely relates to the latest effort, sharing the exploitation of ontology to reason about interface mapping and the further synthesis of protocol converter behaviors according to such mapping, using model checking [12]. However, our work goes one step further by not being tight to the specific Web service domain but instead considering highly heterogeneous pervasive environments where networked systems may build upon diverse middleware technologies and hence protocols.

Our work also closely relates to significant effort from the semantic Web service domain and in particular the WSMO (Web Service Modeling Ontology) initiative that defines mediation as a first class entity for Web service modeling towards supporting service composition. The resulting Web service mediation architecture highlights the various mediations levels that are required for systems to interoperate in a highly open network [48]: data level, functional level, and process level. This has in particular led to elicit base patterns for process mediation together with supporting algorithms [14, 51]. However, as for the above-mentioned work on Web service adaptation, mediation is focused on the upper application layer, ignoring possible mismatches in the lower protocol layers. In other words, work from the Web service arena so far concentrates on interoperability among networked systems from the same technology domain. However, pervasive networks will increasingly be populated by highly heterogeneous systems, spanning, e.g., from systems for sensing/actuating to enterprise information systems. As a result, systems run disparate middleware protocols that need to be reconciled on the fly.

The issue of middleware interoperability has deserved a great deal of attention since the emergence of middleware. Solutions were initially dealing with diverging implementations of the same middleware specification and then evolved to address interoperability among different middleware solutions, acknowledging the diversity of systems populating the increasingly complex distributed systems of systems. As already discussed, one-to-one bridging was among the early approaches [40] and then evolved into more generic solutions such as Enterprise Service Bus [13], interoperability platforms [21] and transparent interoperability approaches [9, 36]. Our work takes inspiration from the latest transparent interoperability approach, which is itself based on early protocol conversion approaches. Indeed, protocol conversion appears the most flexible approach as it does not constrain the behavior of networked systems. Then, our overall contribution comes from the comprehensive handling of protocol conversion, from the application down to the middleware layers, which have so far been tackled in isolation. In addition, existing work on middleware-layer protocol conversion focuses on interoperability between middleware solutions implementing the same interaction paradigm. On the other hand, our approach allows for interoperability among networked systems based upon heterogeneous middleware paradigms, which is crucial for the increasingly heterogeneous pervasive networking environment.

5 Conclusion

The need to deal with the existence of different protocols that perform the same function is not new and has been the focus of tremendous work since the 80s, leading to the study of protocol mediation from both theoretical and practical perspectives. However, while this could be initially considered as a transitory state of affairs, the increasing pervasiveness of networking technologies together with the continuous evolution of information and communication technologies make protocol interoperability a continuous research challenge. As a matter of fact, networked systems now need to compose on the fly while overcoming protocol mismatches from the application down to the middleware layer. Towards that goal, this paper has discussed the foundations of *emergent connectors*, which adapt the protocols run by networked systems that implement a matching functionality but possibly mismatch from the standpoint of associated application protocol and even middleware technology used for interactions. Enabling emergent connectors specifically lies in the appropriate modeling of the networked systems' high-level functionalities and related protocols, for which we exploit ontologies so as to enable unambiguous specification. Compared to related work that deals with either automated protocol conversion/mediation or middleware interoperability, our contribution lies in comprehensively dealing with both the application and middleware layers. In addition, through the alignment of middleware concepts, we are able to deal with interoperability between networked systems relying on heterogeneous middleware paradigms.

While this paper has surveyed the overall model-based approach enabling emergent connectors, it comes along with concrete enablers to be deployed in the network for actual enactment of the connectors [4], as studied in companion chapter on the CONNECT architecture [22]. Enablers include *universal discovery*, which in particular implements the matching and mapping relations discussed in this paper, so as to enable networked systems to meet and compose on the fly. However, it should be acknowledged that most legacy systems do not advertise interfaces like the ones needed by emergent connectors but instead advertise simple interface signatures, as common with today's middleware. This leads the CONNECT project to investigate *learning enablers* so as to enable automated learning of interaction protocols [5, 25] as well as inference of capabilities from interface signatures. Furthermore, while universal discovery enables networked systems to compose abstractly through the proposed model-based approach to emergent connection, concrete connectors need to be instantiated, which concretize the proposed model-based protocol conversion according to actual middleware protocols and application actions. Concretization of mediation processes is in particular investigated based on the exploitation of domain-specific languages as defined in Section 4.4. Preliminary prototypes of the CONNECT enablers are being implemented and will be shortly released on the CONNECT Web site [28].

Acknoledgment. This work is partially supported by the FP7 ICT FET IP Project CONNECT. The authors would also like to thank their colleagues Nikolaos Georgantas, Rachid Saadi, Romina Spalazzese and Paola Inverardi for useful discussion about this work.

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6(3) (1997)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook. Cambridge University Press (2003)
3. Ben Mokhtar, S., Preuveneers, D., Georgantas, N., Issarny, V., Berbers, Y.: EASY: Efficient semantic service discovery in pervasive computing environments with QoS and context support. Journal of Systems and Software 81(5) (2008)
4. Bennaceur, A., Blair, G.S., Chauvel, F., Huang, G., Georgantas, N., Grace, P., Howar, F., Inverardi, P., Issarny, V., Paolucci, M., Pathak, A., Spalazzese, R., Steffen, B., Souville, B.: Towards an architecture for runtime interoperability. In: Proceedings of ISoLA (2) (2010)
5. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic synthesis of behavior protocols for composable web-services. In: Proceedings of ESEC/SIGSOFT FSE (2009)
6. Blair, G., Paolucci, M., Grace, P., Georgantas, N.: Interoperability in complex distributed systems. In: Bernardo, M., Issarny, V. (eds.) SFM-11: 11th International

School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems. Springer Verlag (2011)

7. Bromberg, Y.D.: Solutions to middleware heterogeneity in open networked environment. Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelynes (2006)
8. Bromberg, Y.D., Grace, P., Réveillère, L.: Starlink: runtime interoperability between heterogeneous middleware protocols. In: Proceedings of ICDCS 2011. IEEE Computer Society (2011)
9. Bromberg, Y.D., Issarny, V.: INDISS: Interoperable discovery system for networked services. In: Proceedings of Middleware (2005)
10. Bromberg, Y.D., Réveillère, L., Lawall, J.L., Muller, G.: Automatic generation of network protocol gateways. In: Proceedings of Middleware (2009)
11. Calvert, K.L., Lam, S.S.: Formal methods for protocol conversion. IEEE Journal on Selected Areas in Communications 8(1) (1990)
12. Cavallaro, L., Nitto, E.D., Pradella, M.: An automatic approach to enable replacement of conversational services. In: Proceedings of ICSOC/ServiceWave (2009)
13. Chappell, D.A.: Enterprise Service Bus. O'Reilly (2004)
14. Cimpian, E., Mocan, A.: WSMX process mediation based on choreographies. In: Proceedings of Business Process Management Workshop (2005)
15. Denaro, G., Pezzè, M., Tosi, D.: Ensuring interoperable service-oriented systems through engineered self-healing. In: Proceedings of ESEC/SIGSOFT FSE (2009)
16. Drummond, N., Rector, A.L., Stevens, R., Moulton, G., Horridge, M., Wang, H., Seidenberg, J.: Putting OWL in order: Patterns for sequences in OWL. In: Proceedings of OWLED (2006)
17. Euzenat, J., Shvaiko, P.: Ontology matching. Springer-Verlag, Heidelberg (DE) (2007)
18. Falk Howar, Maik Merten, J.N., Steffen, B.: Introduction to automata learning. In: Bernardo, M., Issarny, V. (eds.) SFM-11: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems. Springer Verlag (2011)
19. Flores-Cortés, C.A., Blair, G.S., Grace, P.: An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. IEEE Distributed Systems Online 8(7) (2007)
20. Foster, H., Uchitel, S., Magee, J., Kramer, J.: LTSA-WS: a tool for model-based verification of web service compositions and choreography. In: Proceedings of ICSE (2006)
21. Grace, P., Blair, G.S., Samuel, S.: ReMMoC: A reflective middleware to support mobile client interoperability. In: Proceedings of CoopIS/DOA/ODBASE (2003)
22. Grace, P., Georgantas, N., Bennaceur, A., Blair, G., Chauvel, F., Issarny, V., Paolucci, M., Saadi, R., Souville, B., Sykes, D.: The connect architecture. In: Bernardo, M., Issarny, V. (eds.) SFM-11: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems. Springer Verlag (2011)
23. Green, P., J.: Protocol conversion. IEEE Transactions on Communications 34(3) (Mar 1986)
24. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM (CACM) 21(8) (1978)
25. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On handling data in automata learning - considerations from the connect perspective. In: Proceedings of ISoLA (2) (2010)

26. Inverardi, P., Spalazzese, R., Tivoli, M.: Application-layer connector synthesis. In: Bernardo, M., Issarny, V. (eds.) SFM-11: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems. Springer Verlag (2011)
27. Issarny, V., Caporuscio, M., Georgantas, N.: A Perspective on the Future of Middleware-based Software Engineering. In: Proceedings of FOSE 2007 (2007)
28. Issarny, V., Steffen, B., Jonsson, B., Blair, G., Grace, P., Kwiatkowska, M., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In: Proceedings of the 14th ICECCS (2009)
29. Kumar, R., Nelvagal, S., Marcus, S.I.: A discrete event systems approach for protocol conversion. Discrete Event Dynamic Systems 7 (June 1997)
30. Lam, S.S.: Protocol conversion. IEEE Transaction Software Engineering 14(9) (1988)
31. Limam, N., Ziembicki, J., Ahmed, R., Iraqi, Y., Li, T., Boutaba, R., Cuervo, F.: Osda: Open service discovery architecture for efficient cross-domain service provisioning. Computer Communications 30(3) (2007)
32. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. on Prog. Lang. and Syst. (1994)
33. Magee, J., Kramer, J.: Concurrency : State models and Java programs. Hoboken (N.J.) : Wiley (2006)
34. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proceedings of ICSE (2000)
35. Menge, F.: Enterprise Service Bus. In: Free and open source software conference (2007)
36. Nakazawa, J., Tokuda, H., Edwards, W.K., Ramachandran, U.: A bridging framework for universal interoperability in pervasive systems. In: Proceedings of ICDCS (2006)
37. Nezhad, H.R.M., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proceedings of WWW (2007)
38. Nitto, E.D., Rosenblum, D.S.: Exploiting adls to specify architectural styles induced by middleware infrastructures. In: Proceedings of ICSE (1999)
39. Okumura, K.: A formal protocol conversion method. In: Proceedings of SIGCOMM (1986)
40. (OMG): COM/CORBA interworking specification Part A & B (1997)
41. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.: Semantic matching of web services capabilities. In: Proceedings of ISWC (2002)
42. Ponnekanti, S., Fox, A.: Interoperability among independently evolving Web services. In: Proceedings of Middleware (2004)
43. Raverdy, P.G., Issarny, V., Chibout, R., de La Chapelle, A.: A multi-protocol approach to service discovery and access in pervasive environments. In: Proceedings of MobiQuitous. IEEE Computer Society (2006)
44. Román, M., Campbell, R.H., Kon, F.: Reflective middleware: From your desk to your hand. IEEE Distributed Systems Online 2(5) (2001)
45. Spalazzese, R., Inverardi, P.: Mediating Connector Patterns for Components Interoperability. In: Proceedings of ECSA (2010)
46. Spalazzese, R., Inverardi, P., Issarny, V.: Towards a formalization of mediating connectors for on the fly interoperability. In: Proceedings of WICSA/ECSA (2009)
47. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: Proceedings of ICSE (2003)

48. Stollberg, M., Cimpian, E., Mocan, A., Fensel, D.: A semantic web mediation architecture. In: Proceedings of CSWWS (2006)
49. Studer, R., Benjamins, V.R., Fensel, D.: Knowledge engineering. Data & Knowledge Engineering (1998)
50. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software architecture : foundations, theory, and practice. Hoboken (N.J.) : Wiley (2009)
51. Vaculín, R., Sycara, K.P.: Towards automatic mediation of OWL-S process models. In: Proceedings of ICWS (2007)
52. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19(2) (1997)
53. Zhu, F., Mutka, M., Ni, L.: Service discovery in pervasive computing environments. Pervasive Computing (2005)